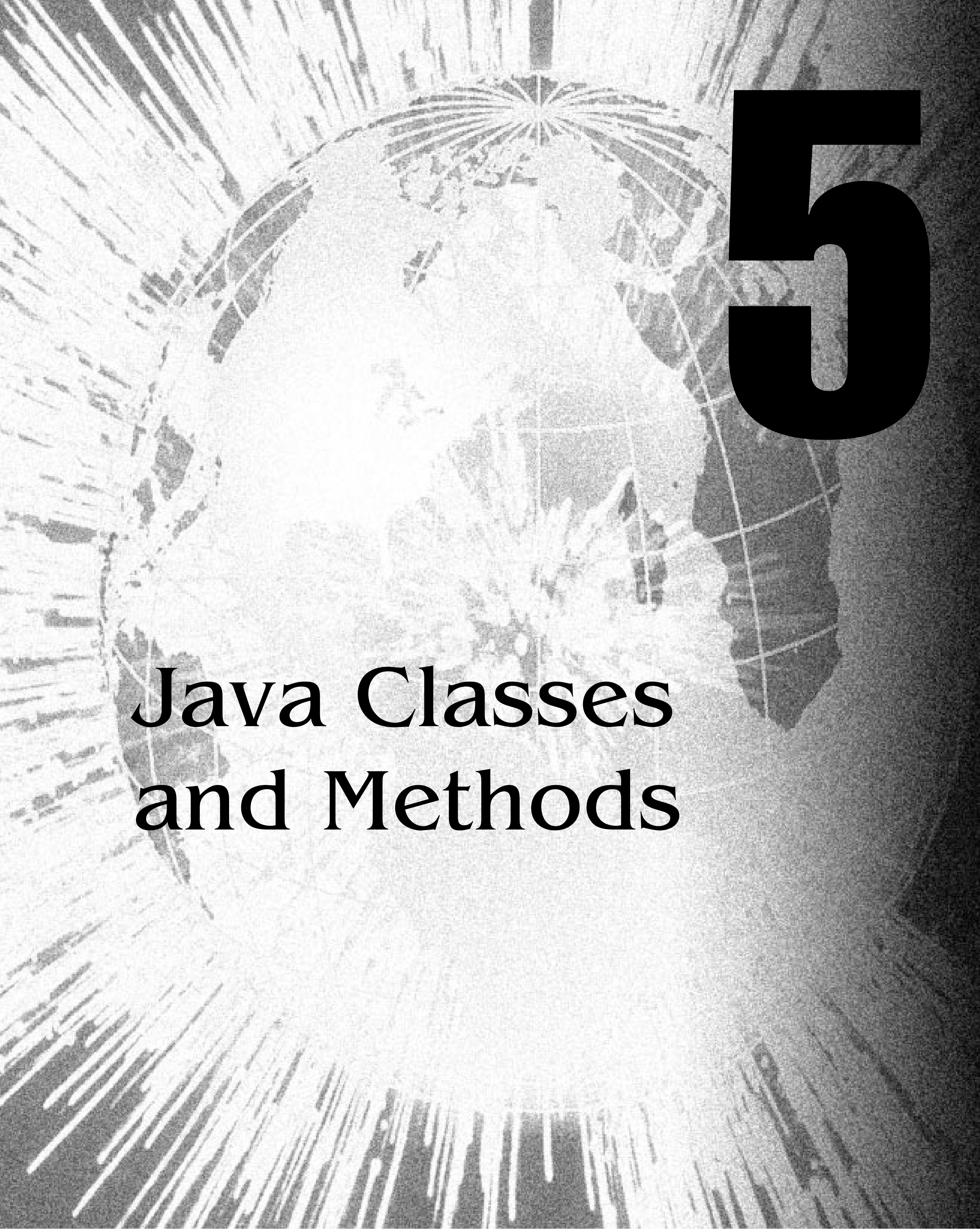
JAVA PROGRAMMING LANGUAGE HANDBOOK

Anthony Potts

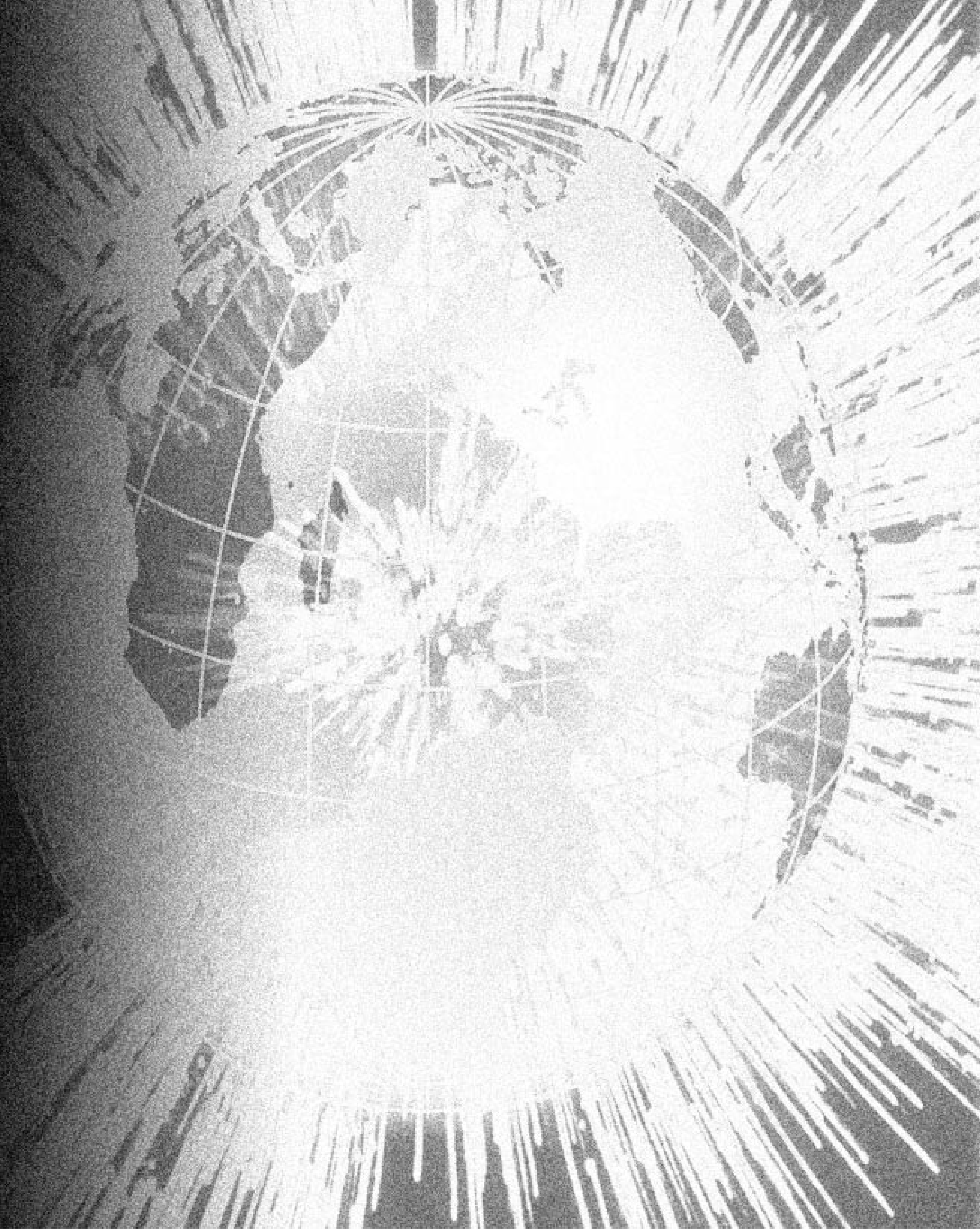
David H. Friedel, Jr.

CORIOLIS GROUP BOOKS



5

Java Classes and Methods



Java Classes and Methods

5

Classes are the key Java components that give the language its object-oriented personality.

If you have some experience programming in a language like C++, you are probably familiar with the power and flexibility that classes provide. They are ideal for plugging general information into a template-like structure for reusing over and over. For example, if you are developing an interactive drawing package, you could create standard classes for some of the fundamental drawing operations and then use those classes to create more sophisticated drawing tasks. If you are new to the world of object-oriented programming (OOP), you'll soon discover that classes are the essential building blocks for writing OOP applications. At first glance, Java classes look and operate like C++ classes; but there are some key differences which we'll address in this chapter.

We'll start by looking at the basics of classes. You'll quickly learn how classes are defined and used to derive other classes. The second half of the chapter covers *methods*—the components used to breathe life into classes.

Understanding Classes

In traditional structured programming languages like C or Pascal, everything revolves around the concepts of algorithms and data structures. The algorithms are kept separate from the data structures, and they operate on the data to perform actions and results. To help divide programming tasks into separate units, components like functions and procedures are defined. The problem with this programming paradigm is that it doesn't allow you to easily create code that can be reused and expanded to create other code.

To solve this problem, object-oriented programming languages like Smalltalk and C++ were created. These languages introduced powerful components called *classes* so that programmers could combine functions (operations) and data under one roof. This is a technique called *encapsulation* in the world of object-oriented programming. Every language that uses classes defines them in a slightly different way; however, the basics concepts for using them remain the same. The main advantages of classes are:

- They can be used to define abstract data types
- Data is protected or hidden inside a class so other classes cannot access it
- Classes can be used to derive other classes
- New classes derived from existing classes can inherit the data and methods already defined—a concept called *inheritance.*

As you'll learn in this chapter, the techniques for defining and using Java classes are adapted from techniques found in the C++ language. In some cases, the Java syntax will look very similar to C++ syntax, but in other cases you'll find a number of differences, including new keywords that have been added to Java for declaring classes and methods; restrictions, such as the elimination of pointers; and different scoping rules that determine how classes can be used in an application.

Declaring a Class

If you recall from Chapter 2, we created a class named **TickerTape,** which controlled how text scrolled across the screen. Let's take a step back and look at the full declaration used to define classes in Java:

```
[Doc Comment] [Modifier] class Identifier
[extends Superclassname]
[implements Interfaces] {
   ClassBody;
}
```

Of course, keep in mind that you won't always use all of the clauses, such as *Doc Comment*, *Modifier*, **extends**, and so on. For example, here's an example of the world's smallest class definition:

```
class Atom_ant {
   int a = 1;
}
```

This class has an identifier, **Atom_ant**, and a body, **int a = 1;**. Of course, don't try to compile this at home as is because it will only result in an error. Why? Well, even though it is a valid class, it is not capable of standing on its own. (You would need to set it up as an applet or a main program to make it work.)

A class declaration provides all of the information about a class including its internal data (*variables*) and functions (*methods*) to be interpreted by the Java compiler. In addition, class declarations provide:

- Programmer comments
- Specifications of the other classes that may reference the class
- Specifications of the superclass the class belongs to (the class's parent)
- Specifications of the methods the class can call

Using a Class

Before we move on and look at all of the other components used to declare classes, let's return to our simple class declaration to see how classes are used in Java programs. Once a class has been declared, you need to use it to create an object. This process is called making an "instance of" a class. In a Java program it requires two steps. First, you declare an object variable using a syntax that looks just like a variable declaration, except the class name is used instead of the name of a primitive data type. For example, this statement would use the **Atom_ant** class we defined earlier to declare an object from the class definition:

```
Atom_ant crazyant;
```

Once the object has been declared, in this case **crazyant**, you then create an instance of it in a Java application by using the **new** operator:

```
crazyant = new Atom_ant();
```

Now the object **crazyant** can access all of the components in a **Atom_ant** class, thus making it an instance of an **Atom_ant** class. To see how this works in context, let's expand our example:

```
class Atom_ant {  // Simple class
   int a = 1;
}
```

```
public class Bug {
   int i = 10;
   Atom_ant crazyant;  // Declare an object

   public static void main (String args[]) {
      // Create an instance of Atom_ant called crazyant
      crazyant = new Atom_ant();
      System.out.println("There are " + bug.i + " bugs here but only " +
        crazyant.i + " atom ant.");
   }
```

The output produced by this example would be:

```
There are 10 bugs here but only 1 atom ant.
```

The main class, **Bug**, creates an instance of the **Atom_ant** class—the **crazyant** object. Then it uses the object to access the data member, **a**, which is assigned a value in the **Atom_ant** class. Notice that the dot operator (.) is used to access a member of a class.

Java Note

Object Declaration Time Saver

In Java, you can both declare an object variable and create an instance all in one statement. Here's an example of how it is done:

```
Atom_ant crazyant = new Atom_ant();
```

Notice that the class **Atom_ant** is used to declare the object variable **crazyant** and then the **new** operator is used to create an instance of **Atom_ant**.

Components of a Class Declaration

Let's look at the components of the class declaration in a little more detail. As you recall from our first example, the only really necessary part of a class declaration is its name or *identifier*. However, whenever you need to reference your class in your program to reuse it, you'll need to reference it by its *fully qualified name*. This name is the package name, or group of classes from which it came, followed by the identifier. For example, if *Atom_ant* is the class name and it belongs to a package named *molecule*, its fully qualified name would be *molecule.Atom_ant*.

Documentation Comment

The *Doc Comment* clause of the class declaration is provided as an aid to help other programmers who might need to use your class. It allows you to write your documentation while you're writing the code. The comments you include as part of this clause can easily be converted into easy to read HTML pages. However, keep in mind that your HTML pages will only be as good as your comments. (To brush up on how to write comments for Java programs, make sure you read Chapter 3.)

Let's look at an example to see how the *Doc Comment* feature works. This class definition

```
/**
* Atom ant is the world's smallest super hero,
  so we gave him a class by himself.
* @author Dave Friedel
*/
class Atom_ant {
   int i = 1;
}
```

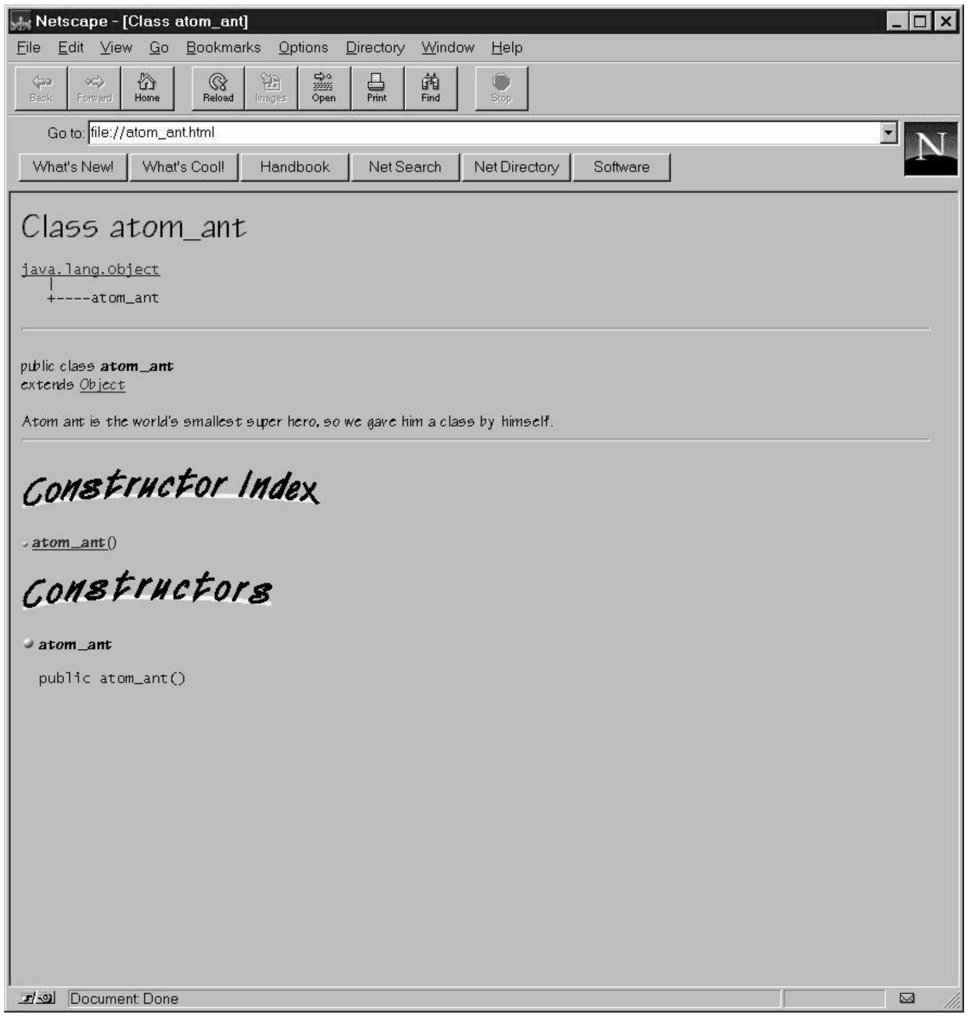
uses *Doc Comment* style comments to produce the HTML page shown in Figure 5.1. Notice how the comments are formatted and used to document the class. In this case, **Atom_ant** is a subclass under the **java.lang.Object** class—the default parent for all classes.

In case you're wondering, the **@author** notation is a special type of comment tag that allows you to personalize your class. These tags are explained in more detail in Chapter 3.

Class Modifiers

Modifiers define the rules for how classes are used in Java applications. They determine how other packages, or classes of other groups can interact with the current class. There are three kinds of modifiers that can be used in a class declaration:

- **public**
- **abstract**
- **final**


Netscape - [Class atom_ant]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
Go to: file://atom_ant.html
What's New! What's Cool! Handbook Net Search Net Directory Software
Document: Done

Class atom_ant

```
java.lang.object
   |
   +----atom_ant
```

public class **atom_ant**
extends Object

Atom ant is the world's smallest super hero, so we gave him a class by himself.

Constructor Index

- **atom_ant**()

Constructors

- **atom_ant**

```
public atom_ant()
```

Figure 5.1
The HTML documentation created for the Atom_ant class.

If you don't use one of these modifiers when declaring a class, Java will automatically decide that only other classes in the current package may access the class. Let's look at how each of these modifiers are used.

PUBLIC CLASS

The **public** modifier is used to define a class that can have the greatest amount of access by other classes. By declaring a class as **public**, you allow all other classes and packages to access its variables, methods, and subclasses. However, *only one public class is allowed in any single Java applet or a single source code file.* You can think of the one public class in an applet as serving the role that the **main**() function does in a C/C++ program.

The source code for an applet must be saved as *ClassName.java*, where *ClassName* is the name of the single public class defined in the applet. Recall that when we created the TickerTape applet in Chapter 2, the single public class was defined as

```
public class TickerTape extends Applet implements Runnable {...
```

and the name of the file was TickerTape.java.

Let's look at another example of how the **public** modifier is used to define a Java class:

```
// Filename: Atom_ant.java
public class Atom_ant {
   public static void main (String args[]) {
      System.out.println("Hello World");
   }
}
```

In this case, **Atom_ant** is the name of the class and the filename for the applet is Atom_ant.java.

Abstract Class

The **abstract** modifier is used to declare classes that serve as a shell or placeholder for implementing methods and variables. When you construct a hierarchy of classes, your top most class will contain the more general data definitions and method implementations that represent your program's features. As you work your way down the class hierarchy, your classes will start to implement more specific data components and operations. As you build your hierarchy, you may need to create more general classes and *defer* the actual implementation to later stages in the class hierarchy. This is where the abstract class comes in. This approach allows you to reference the operations that you need to include without having to restructure your entire hierarchy.

The technique of using abstract classes in Java is commonly referred to as *single inheritance* by C++ programmers. (By the way, limited multiple inheritance techniques can also be implemented in Java by using interfaces. We'll cover this topic in more detail in Chapter 6.)

Any class that is declared as an abstract class must follow certain rules:

- No objects can be instantiated from an abstract class
- Abstract classes must contain at least one declaration of an abstract method or variable
- All abstract methods that are declared in an abstract class must be implemented in one of the subclasses beneath it
- Abstract classes cannot be declared as final or private classes

Let's look at an example of how an abstract class is defined and used to help create other classes:

```
abstract class Quark extends Atom_ant {
   ...
   abstract void abstract_method1();
   abstract void abstract_method2();
   void normal_method();
   ...
}

public class Aparticles extends Quark {
   public void abstract_method1() {
      ... // Definition of the method
   }
}

public class Bparticles extends Quark {
   public void abstract_method2() {
   ... // Definition of the method
   }
}
```

Here, the class **Quark** is declared as an abstract class and it contains two methods that are declared as abstract methods. The subclasses**Aparticles** and **Bparticles** are located beneath the class **Quark** in the hierarchy of classes. Each one defines a method based on one of the abstract methods found in the **Quark** class. A compile-time error would occur if we had failed to define both of the abstract methods in the **Quark** class. All abstract methods must be defined in the subclasses that are derived from abstract classes.

Restrictions in Declaring Abstract Classes

An abstract class cannot be defined as a final class (using the **final** keyword) because the Java compiler will always assume that the abstract class will be used to derive other classes—other subclasses will follow it. (As you'll see in the next section, a final class defines the end of the line for a class hierarchy.) Furthermore, you cannot used a **private** modifier in an abstract class's method declarations because this modifier restricts methods from being used by any other classes except the class they are defined in.

FINAL CLASS

The **final** modifier is used to declare a class that will not be used to derive any other classes. The final class is like the last station on a railway line. By its position in a class hierarchy, a final class cannot have any subclasses beneath it. In **final** class declarations, you cannot use the **extends** clause because the Java compiler always assumes that a final class cannot be extended. Here's an example of what would happen if you tried to declare a final class and then use it in another class declaration:

```
final class Molecule extends Element {
   static String neutron = "molecule";
}

class Atom_ant extends Molecule {
   static String proton = "atom_ant";
}


Compiling...
E:\java\jm\element.java
E:\java\jm\element.java:12: Can't subclass final classes: class
Moleculeclass Atom_ant extends Molecule {      ^1 errorsCompile Ended.
```

In this case, **Molecule** has been defined as a final class. But notice that the second class definition, **Atom_ant**, attempts to use **Molecule** as its parent. The Java compiler catches this illegal declaration and provides the appropriate warning.

Class Identifiers

Each class you define in a Java program must have its own unique identifier. The class's identifier or name directly follows the **class** keyword. The rules for naming classes are the same as those used to name variables. To refresh your memory, identifiers should always begin with a letter of the alphabet, either upper or lower case. The only exception to this rule is the underscore symbol (_) and the dollar sign ($), which may also be used. The rest of the name can be defined using characters, numbers, and some symbols.

Since class names are also used as file names, you need to create names that will not cause problems with your operating system or anyone who will be using your program.

Extending Classes

In most Java applets and programs you write, you will have a number of classes that need to interact each other—in many cases classes will be derived from other classes creating hierarchies. The keyword that handles the work of helping you extend classes and create hierarchies is named appropriately enough, **extends**.

In a class hierarchy, every class must have a parent—except the class that is at the top. The class that serves as a parent to another class is also called the superclass of the class it derives—the class that takes the position immediately above the class. Let's look at an example. As Figure 5.2 indicates, the classes *911*, *944*, and *928* all belong to the superclass *Porsche*. And *Porsche* belongs to the superclass *sportscar*, which in turn belongs to the superclass *automobile*.

When you derive a class from a superclass, it will inherit the superclass's data and methods. (For example, *911* has certain characteristics simply because it is derived from *Porsche*.) To derive a class from a superclass in a class declaration hierarchy, you will need to use the **extend** clause followed by the name of the superclass. If no superclass is defined, the Java compiler assumes that you are deriving a class using Java's top-level superclass named **Object**. Here is an example:

```
public class Element extends Object {
   public static void main() {
     Atom_ant ATOMOBJ = new Atom_ant();
     Molecule MOLEOBJ = new Molecule();
     System.out.println(ATOMOBJ.proton);
```

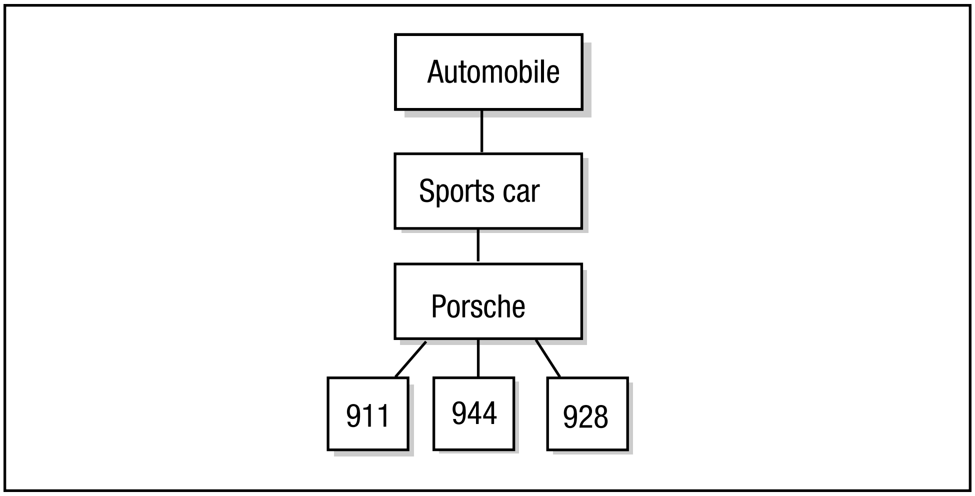


Figure 5.2
A sample class hierarchy.

```
   }
}

class Molecule extends Element {
   static String neutron = "molecule";
}

class Atom_ant extends Molecule {
   static String proton = "atom_ant";
}
```

In this class declaration section, the top-level class defined is **Element**. Notice that it is derived or "extended" from **Object**—the built-in Java class. The first line of the declaration of **Element** could have also been written as

```
public class Element {
...
```

since the Java compiler will assume that a class is automatically derived from the **Object** class if the **extends** clause is omitted. The second class, **Molecule**, is derived from **Element** and the third class, **Atom_ant**, is derived from **Molecule**. As Figure 5.3 shows, both **Molecule** and **Atom_ant** inherit the components of the **Element** class.

Using the implements Clause to Create Class Interfaces

When classes are used to derive other classes, the derived classes can access the data and methods of the classes higher up in the hierarchy chain. Fortunately, Java provides a mechanism called *interfaces* so that classes that are not part of a hierarchy can still access components of other classes. An interface is created for a class by using the **implements** clause. A class can implement as many interfaces as it wishes, but all the interfaces introduced must have all their methods defined in the body of the class implementing it. Thus, all the subclasses that follow from that point on will inherit the methods and variables defined.

Let's develop the **Atom_ant** class we introduced in the previous section to see how an interface can be coded:

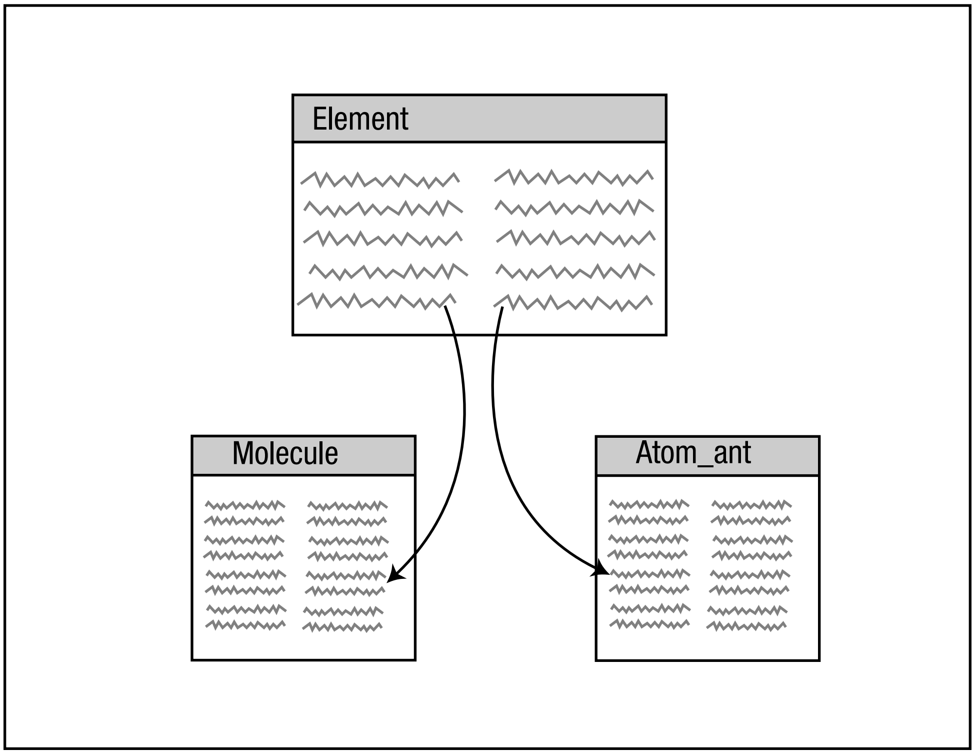


Figure 5.3

Using the **extends** keyword to derive a series of classes.

```
class Atom_ant extends Molecule implements Protons, Neutrons, Electrons {
   static int proton = 45378444;
   void Proton_function() {
      ... // definition of the Proton_function()
   }

   void Neutron_function() {
      ... // definition of the Neutron_function()
   }

   void Electron_function() {
      ... // definition of the Electron_function()
   }
}
```

Here we are making the assumption that the interfaces **Protons**, **Neutrons**, and **Electrons** only have one method declared in each of the interfaces. For example, **Protons** may be set up as follows:

```
Public interface Protons {

   void Proton_function(); // declares the method that will be used
}
```

As you can see, setting up the interface is a two step process. The class where the methods are defined uses the **implements** clause to indicate which interfaces can have access to the methods. Then, **interface** statements are used to declare the method that will be used.

If you recall from Chapter 2, the **TickerTape** class implemented the interface **Runnable** from the package java.lang. The **Runnable** interface has only one method declared in it, which is **run**(). This method is then defined in the class that is implementing it. In this case, the applet TickerTape has defined **run**() to instruct the thread to sleep, call the **setcoord**() method, and rerun the **paint**() method every time the applet calls the **run**() method. This happens in situations where the screen is resized or, in this case, where the applet is instructed to move the text across the screen and the **run**() method is called.

```
// TickerTape Applet

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

```
// TickerTape Class
public class TickerTape extends Applet implements Runnable {
   ...
   public void run() {
      while(ttapeThread != null){ // verifies the thread is still active
         try {Thread.sleep(50);} catch (InterruptedException e){}
         setcoord();  // changes the placement of the text
         repaint();   // repaints the screen by activating the paint()
                      // method
      }
   }
   ...

} // End TickerTape
```

This allows the ability to effectively encapsulate(hide) the classes and all its methods that actually support the **run**() method. Interfaces allow for distinct behaviors, defined by the programmer, to be used without exposing the class(es) to everyone. We'll discuss these techniques in more detail in Chapter 6.

Class Body

The class body contains the code that implements the class. This is where you provide the detail for the actions the class needs to perform (methods) and the data it needs to use (variables). The body can also contain constructors (special methods) and initializers. The basic format for a class body is:

```
{
   Variable-declarations;
   Method-declarations;
 }
```

The variable declarations can be any standard Java declaration (see Chapter 3 and the material presented at the end of this chapter if you need a review). Later in this chapter we'll discuss how methods are declared and used. Here's an example of a class with a body:

```
public class TickerTape extends Applet implements Runnable {
// Beginning of class body
   String inputText;
   String animSpeedString;
   Color color = new Color(255, 255, 255);
   int xpos;
   ...
   // Methods
   public void paint(Graphics g) {
      paintText(osGraphics);
      g.drawImage(im, 0, 0, null);
   }
   ...
// End of Class Body

}
```

Name Space

Every method and variable defined in a class is recorded into an area called a *name space*. This name space is then inherited by the other classes in a class hierarchy which are derived from the class. If a variable or method has been previously defined elsewhere in the structure with the same name, a *shadowing* effect occurs for that level. To access the value of a variable that supersedes the current value, you need to put the prefix clause **super** in front of the variable name. This clause instructs the expression to take the value of the superclass. To access the value of the current variable, you use the prefix **this**. Let's look at an example:

```
public class House extends Object {   static int tvamount = 8;  // Variable
   void main() {
      Room();
   }
}

public class Room extends House {   static int tvamount = 5;   // Variable
   int Child = this.tvamount;  // Child equals 5—same as saying tvamount
   int Parent = super.tvamount;     // Parent equals 8
}
```

In this example the **House** class is derived from the standard **Object** class. Then, the **Room** class is derived from **House**. Now notice that each class defines a variable named **tvamount** and assigns it a value. In the second assignment statement in **Room**, the variable **Child** is assigned the value 5 because **this** is used to access the class's local copy of the **tvamount** variable. In the next assignment statement, notice how **super** is used to access the value **tvamount** was assigned in **House**—the superclass.

Methods

As we've seen, the mechanisms used to implement operations in classes are called *methods*. This terminology is borrowed directly from object-oriented languages like Smalltalk and C++. Methods define the behavior of a class and the objects created from the class. A method can send, receive, and alter information to perform a task in an application. Java requires that every method be defined within a class or interface, unlike C++ where methods (functions) can be implemented outside of classes.

Let's refer to the car class hierarchy we presented earlier in this chapter to get a better understanding of the role methods play. All of the cars we introduced have doors and we could define two methods to operate on these doors: open and close. These same methods could be designed to perform operations on other car components such as windows, trunks, hoods, and so on. A component like a door can be viewed as an object. Of course, a car would be made up of many objects and many methods would be required to process all of the objects. As a programmer, it would be up to you to decide how to arrange the objects you need and what methods must be implemented.

Declaring a Method

If you recall from Chapter 2, our TickerTape applet included a number of methods. The first one defined was the **init**() method, which was responsible for initializing the applet upon loading. Let's take a step back and look at the full declaration used to define a Java method:

```
[Modifier] ReturnType Identifier([ParameterList]) [Throws]
{
   MethodBody;
}
```

The *Modifier* and *Throws* clauses are optional. They are used to specify how the method needs to be accessed and which exceptions should be checked for. (For more information on exceptions and how to catch errors, refer to Chapter 7.)

Components of a Method Declaration

If you were to break down the method declaration, you would find it performs three main tasks:

- It determines who may call the method
- It determines what the method can receive (the parameters)
- It determines how the method returns information

Method Modifiers

Earlier in this chapter, you learned that a set of modifiers are available for defining how classes can be accessed. Methods also can be defined using modifiers, although the method modifiers only affect how methods are used, not the class they are defined in. Java provides eight modifiers for defining methods, but only one modifier from each of the groups listed next may be used in a method declaration. For example, you cannot use a **public** and **private** modifier in the same declaration. Here is the complete set of method modifiers:

- **public**, **protected**, **private**
- **static**
- **abstract**, **final**, **native**, **synchronized**

Keep in mind that it doesn't make sense to use some modifiers in one group with modifiers from another group. For example, a method that is defined using the **private** and **abstract** modifiers contradicts itself. An abstract method is one that requires its actual code to be defined in the subclasses that follow, whereas a private method is one that can only be accessed in the class it is defined in. The rule of thumb when choosing and combining modifiers is that you need to make sure that they are complementary rather than contradictory. If a modifier is not used, the method may be accessed only by the classes that are in the current package.

Public Method

A method declared as public can be accessed by *any* class in the same package. It can also be accessed by other classes from other packages. This modifier gives a method the most freedom.

Protected Method

A method declared as protected can only be used by other classes within the same package. All the subclasses beneath the class the method is defined in may access the method unless shadowing occurs. Shadowing involves naming a method using a name that already exists in a superclass above the class the method is defined in.

Private Method

A method declared as private is one that can only be accessed by the class it is defined in. This modifier gives a method the least amount of freedom.

Static Method

A method declared as static is one that cannot be changed. This type of method is also referred to as a *class method*, because it belongs explicitly to a particular class. When an *instance* of the class that defines the method is created, the static method cannot be altered. For this reason, a static method can refer to any other static methods or variables by name. Limitations of static methods to keep in mind are that they cannot be declared as final, and they cannot be overridden.

Abstract Method

A method declared as abstract is one that must be defined in a subclass of the current class. However, an abstract method must be declared in the current class with a (;) semicolon in place of the method's block of code. Methods that are declared abstract are not required to be implemented in every subclass.

Final Method

A method declared as final is one that ends the hierarchical tree. No methods having the same name can be defined in subclasses that exist below the class that declares the method as final.

Native Method

A method declared as native is one that will be implemented using outside code—code that is written in another language, to be used in conjunction with your current program. This limits you to a specific platform and restricts you from creating Java applets. Native methods are declared by leaving out the method body and placing a semicolon at the end of the method declaration.

Synchronized Method

A method declared as synchronized limits it from being executed by multiple objects at the same time. This is useful when you are creating Java applets and you could have more than one thread running at the same time accessing one central piece of data. If the method is static (e.g., a class method), the whole class would be locked. If you just declare a particular method as synchronized, the object containing the method would only be locked until the method finishes executing.

Return Type of a Method

Any information that is returned from a method is declared as the *return type.* This assures that the information that is returned from a method call will be of the correct type; otherwise, a compile-time error will be generated. If no information will be returned by a method, the **void** keyword should be placed in front of the method name. The different data types that may be returned by methods are covered in Chapter 4.

Parameter Lists for a Method

The parameter list consists of the ordered set of data elements passed to a method. You can pass zero, one, or multiple parameters by listing them between the parentheses, with each type and variable name being separated by a comma. If no parameters are passed, the parentheses should be empty. All variables that are passed become local for that instance of the method. Here's an example of how methods can be declared with and without parameters:

```
public static void MyFirstMethod(String Name, int Number) {
   ...
   // the String variable Name is assigned whatever is passed to it
   // the integer variable Number is assigned whatever is passed to it
   ...
}

public static void MyFirstMethod() {
   ...
   // Nothing is passed  to it.
   ...
}
```

Method Throws

The **throws** clause is used to specify the type of error(s) that will be handled within a method. In effect, it is used to help you set up an automatic error-handler. In the event of an error, the error must be assignable to one of the exceptions in either the **Error**, **RunTimeException**, or **Exception** classes. (These are special classes that Java provides for catching compile-time and run-time errors. We'll cover them in more detail in Chapter 7.) Each method you declare does not need to use the **throws** clause in its declaration, but in the event of an error, the omission of this clause will leave the error handling up to the Java compiler or the Java interpreter. Let's look at an example of how the **throws** clause is used.

In the following method declaration, the Java exception named **ArrayOutOfBoundsException** is specified so that in the event an array range error occurs, the method will know how to handle the error:

```
public class Array_check() {
   String arr[5];

   public static void main(void) throws ArrayOutOfBoundsException {
      int i=0;
      char ch;

      // Specify which code should be tested
      try {
        while (i <= 5) ch = arr[i++];
      }
      // An error has occurred—display a message
      catch {
        System.out.println("Array out of bounds");
      }
   }
 }
```

At some point **main**() will try to access a location outside the legal range of the array **arr**[]. When this happens, an exception will be "thrown" and the **catch** clause will handle it. Also notice the use of the **try** clause which is needed to specify which code in the method should be tested. In our case, we want to check each iteration of the **while** loop.

Method Body

All executable code for Java classes is contained in the body of the methods. Unless a method is declared as abstract, native, or is declared in the body of an interface, the code for the method is placed between a pair of curly braces. This code can be any valid Java statements including variable declarations, assignment statements, method calls, control statements, and so on.

Here's an example of how a basic method is defined:

```
public int SimpleMethod(int Number) {

   // The integer variable Number is assigned whatever is passed to it

   int lowrange = 1;  // Local declarations for the method
   int highrange = 10;

   if (Number <= lowrange) return -1;
   if (Number >= highrange) return 100
      else return 50;
}
```

In this case, the method's name is **SimpleMethod**(). Because it is declared as public, it can be used by any class in the package in which the method is defined. The return type for the method is **int** and it accepts one **int** parameter. The method body contains a few local declarations and a set of if-then decision-making statements.

For a method declared as abstract, native, or one that is declared in an interface, the body is left blank and the declaration is terminated with a semicolon. The bodies are then defined elsewhere depending on how they are declared. Here's an example:

```
abstract class Aparticles extends Quark {

   abstract int abstract_method();  // Defined in the subclasses of the class

   native void native_method ();  // Defined in an external process

   public String normal_method() {
      ... // Definition of the method
   }
}
```

Using the this and super Keywords

To access class variables and methods from within an object, you can reference them by using the keywords **this** and **super**. When the Java compiler encounters the **this** keyword in the body of a method, it knows that you are accessing other methods and variables defined within the scope of the class the method is defined in. On the other hand, variables and methods that are available for accessing in the parent class (superclass) to the current class are referenced using the **super** keyword. Here's an example of how each of these keywords can be used:

```
class Atom_ant extends Molecule {
   int Number;
   ...
}
```

```
class Quark extends Atom_ant {
   int Proton;
   int Neutron;
   String Electon = "Negative attraction";
   ...
   void Count() {
      System.out.println(this.Proton + " is the number of Protons"); // Correct
      System.out.println(Neutron + " is the number of Neutrons"); // Correct
      System.out.println(super.Number + " is the number of Atoms"); // Correct
      System.out.println(Atom_ant.Number + " is the number of Atoms");
        // Correct
      ...
   }
}
```

In this example, this.Proton references the local variable Proton defined in the class Quark. But take a look at the second method call in the Count() method. Here, the variable Neutron, which is also declared in Quark, is referenced without the use of the **this** keyword. What gives? Actually, since both of these variables are defined within Quark, the **this** keyword is not really needed.

As for the two following lines of code, they each reference the **Number** variable declared in the **Atom_ant** class, which serves as the parent to the **Quark** class. Notice that the keyword **super** is placed in front of the variable **Number** to allow it to be accessed. This is the same as using the superclass name in the statement **Atom_ant.Number** to reference the value of **Number**. Superclass names can be referenced further up the hierarchical tree but the **super** keyword can only be used to access class members that reside one level above the current class. If the **Molecule** class contained a variable named **M1**, and we wanted to reference it from the **Quark** class, a statement like this would be required:

```
Proton = Molecule.M1;
```

Here the superclass named **Molecule** is included in the assignment statement. If it was omitted or the **super** keyword was used instead,

```
Proton = super.M1;
```

the Java compiler would return an error because it would try to locate the **M1** variable in the class that is directly above the **Quark** class.

Overloading and Overriding Methods

A method may be declared with multiple declarations, each specifying different types and arguments that may be passed to the method. *The context in which the method is called will determine which actual method code is used* The techniques of using a method's name more than once to define an operation in a class involves overloading and overriding methods. As long as you can define each method having the same name so that it can be distinguished from the others sharing the same name, the Java compiler will not give you an error. The technique for creating overridden methods involves using different parameters (types and numbers) and return types. Methods that are inherited from a superclass may be overridden but the overriding method must provide at least the same access.

Let's look at some examples of how we can override methods:

```
class Atom_ant extends Molecule {
   int Number;
   protected void Count(String Astring, int Number) {

      ...
   }
}

class Quark extends Atom_ant {
   int Proton;
   int Neutron;
   String Electon = "Negative attraction";
   ...
   public void Count(int Number, String Astring) { // Correct
      ...
   }

   protected void Count() {   // Correct
      ...
   }
}
```

Here we've declared two classes: **Atom_ant** and **Quark**. **Atom_ant** serves as the superclass. The method that is overridden is **Count**(). It is first introduced as a protected method in the **Atom_ant** class. Notice that it is declared here as taking two parameters: **Astring** and **Number**. Because **Atom_ant** is declared as a protected method, it is restricted from being called by other classes outside of the package **Atom_ant** is declared in.

The **Quark** class, which is derived from **Atom_ant**, provides two new variations of the **Count**() method, each one being overridden from the base method defined in **Atom_ant**. Notice that each of the overridden methods uses different parameters and/or return types than the original method.

To see how the different versions of the **Count**() method can be called, let's expand the **Quark** class a little:

```
class Atom_ant extends Molecule {
   int Number;
   protected void Count(String Astring, int Number) {
      ...
   }
}

class Quark extends Atom_ant {
   int Proton;
   ...
   public void Count(int Number, String Astring) { // Correct
   ...
   }

   void check() {
      Atom_ant.Count("Hello", 5); //Correct refer to superclass method
      super.Count("GoodBye", 5);  //Correct same as previous
      Molecule.Count("Hello World"); //Correct as long as it exists
      Count(5, "World");             //Correct same as this.Count
   }
}
```

The first two calls to the **Count**() method result in calling the **Count**() method defined in **Atom_ant**. For the third call, we are making the assumption that the class **Molecule**, which **Atom_ant** is derived from, contains a **Count**() method. If it doesn't, a compiler error will occur. The last call to **Count**() accesses the method of the same name defined in **Quark**.

Constructors—The Special Methods

Although constructors are identified as special methods, it is important to distinguish between the two. Methods define an object's behavior in terms of what operations the object can perform. A constructor, on the other hand, determines how an object is initialized by creating a new instance of a class with specified parameters.

Methods and constructors actually differ in three ways. First, constructors do not have their own unique names; they must be assigned the same name as their class name. Second, constructors do not have a return type—Java assumes that the return type for a constructor is **void**. And third, constructors are not inherited by subclasses, as are methods.

To understand how constructors work conceptually, let's return to the car analogy we introduced earlier in this chapter. Each car in our hierarchy represents an object and the blueprint for each car is a class structure. Also recall that operations such as opening and closing car doors were considered to be our methods.

Now, imagine that we have a subclass, called *BodyShop*, which defines the body style for a car. This class could be inserted under the general *car* class in the class hierarchy. An object could be created from this class called *FrameCreation*, which is responsible for making body frames for cars. The process of building a frame could involve first calling a constructor to do the dirty work of "setting up the shop" for building a particular car frame. The manner in which the different classes are defined in the hierarchy will determine what frame a particular car gets at the *BodyShop* from the *FrameCreation* team. (The *FrameCreation* team is responsible for initializing an "object" depending on the information passed to a constructor.)

Now let's assume we have three choices for making body frames:

- 4 Door(*integer*) Falcon(*String*)
- 3 Door(*integer*) Pinto(*String*)
- 2 Door(*integer*) Mustang(*String*), which is the default.

We could just say 2, 3, or 4 doors, but the *FrameCreation* team insists on a certain format for each. The Falcon requires (*integer* Doors, *String* Name), the Pinto requires (*String* Name, *integer* Doors), and the Mustang doesn't require any values (). When you pass these values, known as **types** to the *FrameCreation* team, they immediately know which frame to create, or *initialize*, by the arrangement of the information passed to them (data types and number of parameters). By passing the information in a distinct pattern *FrameCreation(Doors, Name)*, *FrameCreation(Name, Doors)*, or *FrameCreation()* to create an object, we are using a *constructor*.

A constructor is a special method that determines how an object is initalized when created. The constructor is named the same as the class it follows. The code for our example could be written like this:

```
class  FrameCreation extends BodyShop {
   //  ** Initializing the object newcar **
   FrameCreation newcar = FrameCreation(4 , Falcon);

// ** The Beginning of the Constructor **
   FrameCreation {
   // ** An example of Overloading the Constructor **
      FrameCreation(int, String) {
     // Creates the FALCON
   }
// ** An example of Overloading the Constructor **
   FrameCreation(String, int) {
  // Creates the Pinto
   }

   FrameCreation() {   // ** An example of Overloading the Constructor **
  // Creates the Mustang
   }
// ** The End of the Constructor **
}
```

FrameCreation is the constructor, which is declared multiple times—each taking different parameter configurations. When it is called with a configuration (a number, a word), the constructor with the matching configuration is used.

In calling a constructor, you need to disregard the rules for calling methods. Methods are called directly; constructors are called automatically by Java. When you create a new instance of a class, Java will automatically initialize the object's instance variables, and then call the class's constructors and methods. Defining constructors in a class can do several things, including:

- Setting initial values of the instance variables
- Calling methods based on the initial variables
- Calling methods from other objects
- Calculating the initial properties of the object
- Creating an object that has specific properties outlined in the new argument through overloading

Components of a Constructor Declaration

The basic format for declaring a constructor is:

```
[ConstructorModifier] ConstructorIdentifier([ParameterList]) [Throws] {
   ConstructorBody;
}
```

As with the other declarations we've introduced in previous sections, only the identifier and body are necessary. Both the modifier and the throws clause are optional. The identifier is the name of the constructor; however, it is important to remember that the name of the constructor must be the same as the class name it initializes. You may have many constructors (of the same name) in a class, as long as each one takes a different set of parameters. (Because the different constructors in a class must have the same name, the type, number, and order of the parameters being passed are used as the distinguishing factors.) For example, all constructors for a class named**Atom_ant**, must be named **Atom_ant**, and each one must have different set of parameters.

In addition to having a unique declaration from that of a method, a special format is used for calling a constructor:

```
Typename([ParameterList]);
```

The only required element is *Typename*, which names the class containing the constructor declaration. Here's an example of a constructor, with the class **Atom_ant** and a constructor that uses the **new** operator to initialize instance variables:

```
class Atom_ant {
   String  superhero;
   int height;

   Atom_ant(String s, int h) {  // Declare a constructor
      superhero = s;
      height = h;
   }

   void printatom_ant() {
      System.out.print("Up and attam, " + superhero);
      System.out.println("!  The world's only " + height +
        " inch Superhero!");
    }

    public static void main(String args[])  {
       Atom_ant a;
```

```
        a =  new Atom_ant("Atom Ant" , 1); // Call the constructor
        a.printatom_ant();
        System.out.println("------");

        a = new Atom_ant("Grape Ape", 5000);
        a.printatom_ant();
        System.out.println("------");
    }
}
```

The output for this program looks like this:

```
   Up and attam,  Atom Ant!  The world's only 1 inch Superhero!
   ------
   Up and attam, Grape Ape!  The world's only 5000 inch Superhero!
   ------
```

Notice that each constructor call is combined with the **new** operator. This operator is responsible for making sure a new instance of a class is created and assigned to the object variable **a**.

Using Java's Default Constructor

If you decide not to declare a constructor in a class, Java will automatically provide a default constructor that takes no arguments. The default constructor simply calls the superclass constructor **super**() with no arguments and initializes the instance variable. If the superclass does not have a constructor that takes no arguments, you will encounter a compile-time error. You can also set a class's instance variables or call other methods so that an object can be initialized.
Here is an example of a Java class that does not use a constructor but instead allows Java to initialize the class variables:

```
class Atom_ant2 {
   String  superhero;
   int height;
   Boolean villain;
   void printatom_ant() {
      System.out.print("Up and attam, " + superhero);
      System.out.println("!  The world's only " + height +
        " inch Superhero!");
    }

    public static void main(String args[])  {
        Atom_ant2 a;
        a =  new Atom_ant2();
        a.printatom_ant();
        System.out.println("------") ;
    }
}
```

Because no constructor is defined for this example program, the Java compiler will initialize the class variables by assigning them default values. The variable **superhero** is set to null, **height** is initialized to zero, and **villain** is set to false. The variable **a**, in the **main**() method, could have been initialized at the time the constructor was called by substituting the code **a = new Atom_ant2**(); for **Atom_ant2 a = new Atom_ant2**();. Either statement provides an acceptable means of creating an instance of a class—the object **a**. Once this object is in hand, the method **printatom_ant**() can be called.

The output for this program looks like this:

```
Up and attam, The world's only 0 inch Superhero!
   ------
```

Constructor Modifiers

Java provides three modifiers that can be used to define constructors:

- **public**
- **protected**
- **private**

These modifiers have the same restrictions as the modifiers used to declare standard methods. Here is a summary of the guidelines for using modifiers with constructor declarations:

- A constructor that is declared without the use of one of the modifiers may only be called by one of the classes defined in the same package as the constructor declaration.
- A constructor that is declared as public may be called from any class that has the ability to access the class containing the constructor declaration.
- A constructor that is declared as protected may only be called by the subclasses of the class that contains the constructor declaration.
- A constructor that is declared as private may only be called from within the class it is declared in.

Let's look at an example of how each of these modifiers can be used:

```
class Atom_ant2 {
   String  superhero;
   int height;
   String  villain;
   int numberofsuperheros;

   Atom_ant2() {
      this("Dudley Do Right", 60);
   }

   public Atom_ant2(String s, int h) {
      superhero = s;
      height = h;
   }

   protected Atom_ant2(int s, int h) {
      numberofsuperheros = s;
      height = h;
   }

   private Atom_ant2(String s, int h, String v) {
      superhero = s;
      height = h;
      villain = v;
      }

   void printatom_ant() {
      System.out.print("Up and attam, " + superhero);
      System.out.println("!  The world's only " + height +
        " inch Superhero!");
   }
   public static void main(String args[]) {
      Atom_ant2 a;

      a =  new Atom_ant2();
      a.printatom_ant();

      a = new Atom_ant2("Grape Ape", 5000);
      a.printatom_ant();
    }
}

class Molecule_mole extends Atom_ant2 {
   String  superhero;
   int height;
   public static void main(String args[]) {
      Atom_ant2 a;

      a =  new Atom_ant2(); // Compile-time Error
      a.printatom_ant();

      a =  new Atom_ant2("Atom Ant", 1);  // Correct
      a.printatom_ant();

      a =  new Atom_ant2(5, 5); // Correct
      a.printatom_ant();

// Compile-time Error
      a =  new Atom_ant2("Atom Ant", 1 , "Dudley Do Right");
      a.printatom_ant();
    }
}
```

In this example, the **Atom_ant2** class uses constructors with all three of the modifiers: **public**, **protected**, and **private**. In addition, a constructor is declared that does not use a modifier. Notice how the constructors are called from the **Molecule_mole** class. Each constructor type is both defined and called using a different parameter configuration. (This is how the Java compiler knows which constructor to use.)

The first constructor call, **Atom_ant2**(), produces a compiler error because of Java's scoping rules—the declaration of this constructor is outside of the range of the **Molecule_mole** class, and the constructor was not declared as public or protected. Also notice that the call to the fourth constructor produces a compiler error. In this case, the constructor was declared in the **Atom_ant** class as private, which limits the constructor from being called by the class it is declared in.

As this example illustrates, you need to make sure you understand the restrictions that modifiers can place on method declarations. For example, here is an example of a compile-time error you will encounter if you try to access a constructor from another class when its modifier has been declared as private:

```
Compiling...
E:\java\jm\Molecule_mole.java
E:\java\jm\Molecule_mole.java:8: No constructor matching _
  Atom_ant2(java.lang.String, int, java.lang.String) found in class Atom_ant2.
                 a =  new Atom_ant2("Atom ant",5,"Dudley");
^1 error
Compile Ended.
```

Parameter List and Throws Clause

Both the parameter list and throws clause follow the same rules used for declaring and calling methods; after all, a constructor is just a special method. When calling a constructor, different parameter configurations (type of parameters and quantity) can be used as long as you have a matching declaration that uses the same parameter configuration.

Constructor Body

The body of the constructor is essentially the same as the body of a method. The only difference occurs in the first statement. If the constructor is going to call "itself" (an alternate constructor for the same class having the same name) or call the constructor of its superclass, it must do this in the first statement. To access its own class, the **this**() statement is used as a placeholder for the class's identifier. To refer to the class's superclass, the **super**() statement is used. Following each of the clauses are parentheses containing the parameter list to be passed to the constructor, identified by the keyword. Here is an example of how both the **this**() and **super**() statements are used within the constructors defined for **Atom_ant2**:

```
class Atom_ant2 extends Quark {
   String  superhero;
   int height;
   String  villain;
   int numberofsuperheros;

   Atom_ant2() {
    this("Atom Ant", 1);   // Call another Atom_ant2() constructor
   }

   public Atom_ant2(String s, int h) {
      superhero = s;
      height = h;
   }

   Atom_ant2(String s, int h, String v) {
    super(s, h);    // Call the superclass's constructor
   }

   protected Atom_ant2(int s, int h) {
      numberofsuperheros = s;
      height = h;
   }
   synchronized void printatom_ant() {
      System.out.print("Up and attam, " + superhero);
      System.out.println("!  The world's only " + height +
        " inch Superhero!");
  System.out.print("\n-----\n");
   }

   public static void main (String args[ ])  {
      Atom_ant2 a;

      a =  new Atom_ant2();
      a.printatom_ant();
      System.out.println ("------") ;
   }
}
```

When the program runs, the call to **Atom_ant2**() results in the first constructor defined in the **Atom_ant2** class being called. Then, the first constructor calls the second constructor defined in the class. This process is illustrated in Figure 5.4.

In the first constructor, **this**() is used so that the constructor can directly call one of **Atom_ant2**'s other constructors. How does the compiler know which one to use? It looks for a match between the parameters based on **this**("**Atom Ant**", **1**) and one of the other **Atom_ant2**(**...**) constructors. Since the **this**() statement passes a string and an integer, the actual constructor that is called is the second one defined in the **Atom_ant2** class.

In the third constructor declaration, the **super**() statement performs a similar operation except this time it searches the immediate superclass's constructor for

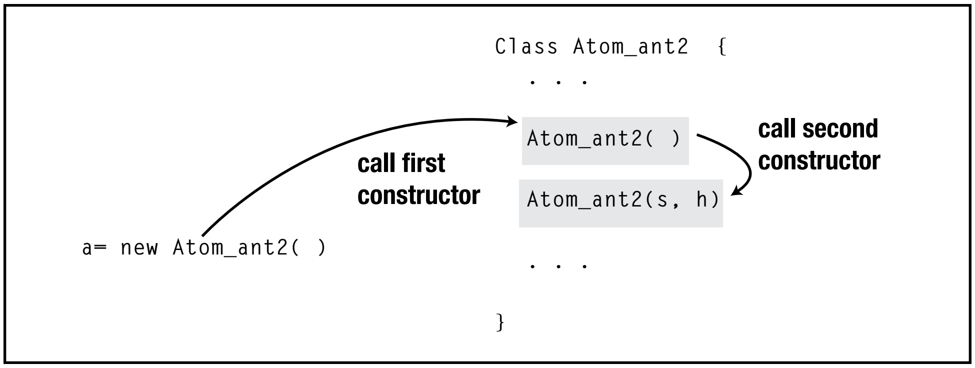


Figure 5.4
The chain of constructor calls in the Atom_ant2 example.

a match. It is important to remember that when using either of these statements, you may not directly call instance variables of the object being created. Furthermore, an instance variable cannot be dependent upon another variable that has not yet been defined, or is defined after it.

Here's an example:

```
class Foo {
   int variableNow = variableLater + 10;
   int variableLater = 20;
}
```

As you can see, **variableNow** is trying to initialize itself before **variableLater** is assigned a value.

Object Creation

There are two ways to create an instance of a class: use a literal, specific to the **String** class or use the **new** operator. The **new** operator is placed in front of the constructor. The parameter list of the constructor determines what constructor is used to create an instance of an object.

```
...
public static void main(String args[])  {
   Atom_ant2 a;

   a =  new Atom_ant2();
   a.printatom_ant() ;
   System.out.println ("------");
}
...
```

Here, the **new** operator initializes **Atom_ant2** with an empty parameter list, initializes the variable to create an instance of the class **Atom_ant2**, and assigns it to **a**.

Variables for Classes

If you've read Chapter 3, you should already be familiar with the basics of declaring variables in Java. Let's refresh your memory: A variable is a named storage location that can hold various values, depending on the data type of the variable. The basic format for declaring a variable is as follows:

```
VariableModifiers Type Indentifier = [VariableInitializer];
```

Only the *Type* and *Identifier* components are necessary. The modifiers are optional.

As with all the identifiers we've used throughout this chapter, the variable identifier simply names the variable. However, you can name any number of variables in the declaration by naming them in the identifier position and separating them with commas. If you decide to declare multiple variables, also realize that the modifiers and *Type* apply to all the variables that are named. For example, in these declarations

```
int paul, david, kelly;
static String henry, diana;
```

the variables **paul**, **david**, and **kelly** are declared as integers, and the variables **henry** and **diana** are declared as static strings.

Variable Modifiers

Java provides seven different modifiers for declaring variables within classes. However, you can only use two of them—one from each group—in a declaration. Also, you can't use two modifiers that contradict each other in the same declaration. The two groups of modifiers are:

- **public**, **protected**, **private**
- **static**, **final**, **transient**, **volatile**

The **public**, **protected**, and **private** modifiers are discussed under the modifiers sections of class, method, and constructors.

Static Modifiers

A static variable is also known as a class variable. This is because there is only one variable of that name, no matter how many instances of the class are created. Here's an example of how the **static** modifier can be used:

```
Atom_ant2() {
   static int Doug = 9;
   this("Atom Ant", 1);
}
```

```
...
public static void main(String args[])  {
   Atom_ant2 a, b, c, d;

   a =  new Atom_ant2();
   b =  new Atom_ant2();
   c =  new Atom_ant2();
   d =  new Atom_ant2();
   a.printatom_ant() ;
   System.out.println("------") ;
}
...
```

Here, no matter how many objects we create, there is exactly one variable **Doug** for every instance of **Atom_ant**().

Final Modifier

When a variable is assigned final, it acts as a constant throughout the instance of the class. They must be declared at time of initialization of the class or method.

Transient Modifier

This is a modifier that has been reserved by Java virtual machine language for low level segments that do not pertain to the persistent state of an object. Other implementations will follow for this modifier in future versions.

Volatile Modifier

These are modifiers that are processed through the multi-processor in an asynchronous manner. The variables are reloaded from and stored to memory every time the variables are used.

The Art of Casting with Classes

When we introduced the fundamental data types in Chapter 3, we showed you how to use casting techniques to convert the values assigned to variables of predefined data types to other data types. For example, in a set of statements like this

```
int i;
short s;
```

```
s = 10;
i = (int) s;
```

the contents of the variable **s**—originally defined to be of the **short** type—is converted to an **int** type by using a cast in the assignment statement. When casting variable types from one to another, no information will be lost as long as the receiver is larger than the provider. Java also allows you to cast instances of a class, known as objects, to instances of other classes. The declaration for an explicit cast to a class is as follows:

```
(Classname)reference
```

The *Classname* is the name of the class you wish to cast to the receiving object. The reference specifies the object that is to receive the cast. When applying a narrowing effect to a class, as you will read about later, this type of cast is required by the Java compiler. Figure 5.5 illustrates this concept.

If a superclass attempts to cast an instance of itself to a subclass beneath it, a runtime error will occur even though this type of cast will be accepted by the Java compiler. The technique of passing object references down a class hierarchy is referred to as *widening*. As a class is located at lower levels in a hierarchy it becomes more specific and thus it contains more information than the classes above it in the hierarchy. Superclasses, on the other hand, are usually more general than the classes beneath them. Conversions that occur when you pass the references up the hierarchy are thus referred to a narrowing because not all the information is passed along to the receiving object. Furthermore, all instance variables of the same name in the receiving object are set to the class variables that are being casted.

Casting an Object vs. Creating an Object

When casting between instances of a class, an object only assumes reference of the class. A new instance of the class is not created; the object merely points to the methods and variables of the casting class. It is important *not* to confuse the process of casting a object with the process of creating an object. Just as you pass the value of a variable through different types (e.g, **int**, **float**, **double**, and so on), you can pass an object through different classes, as long as the class is in the current hierarchy.

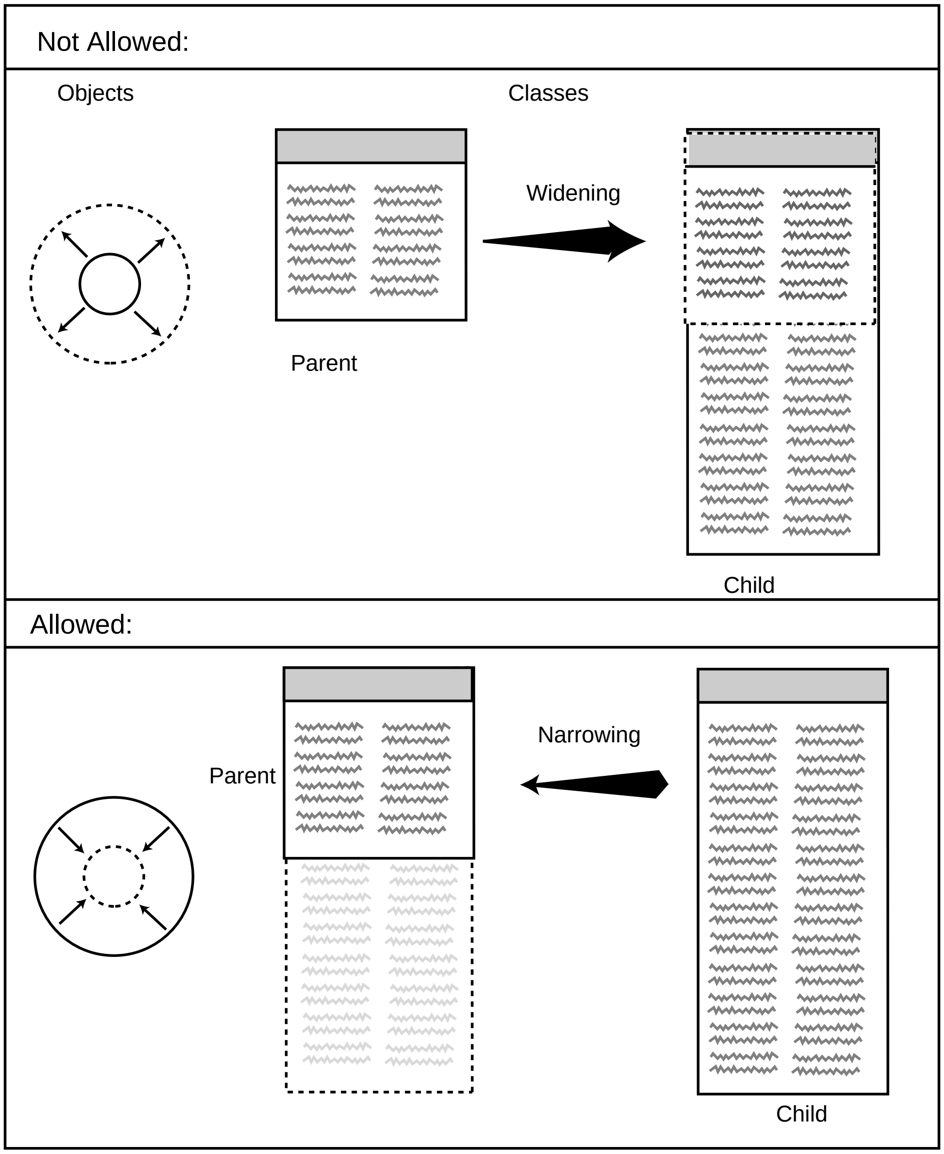
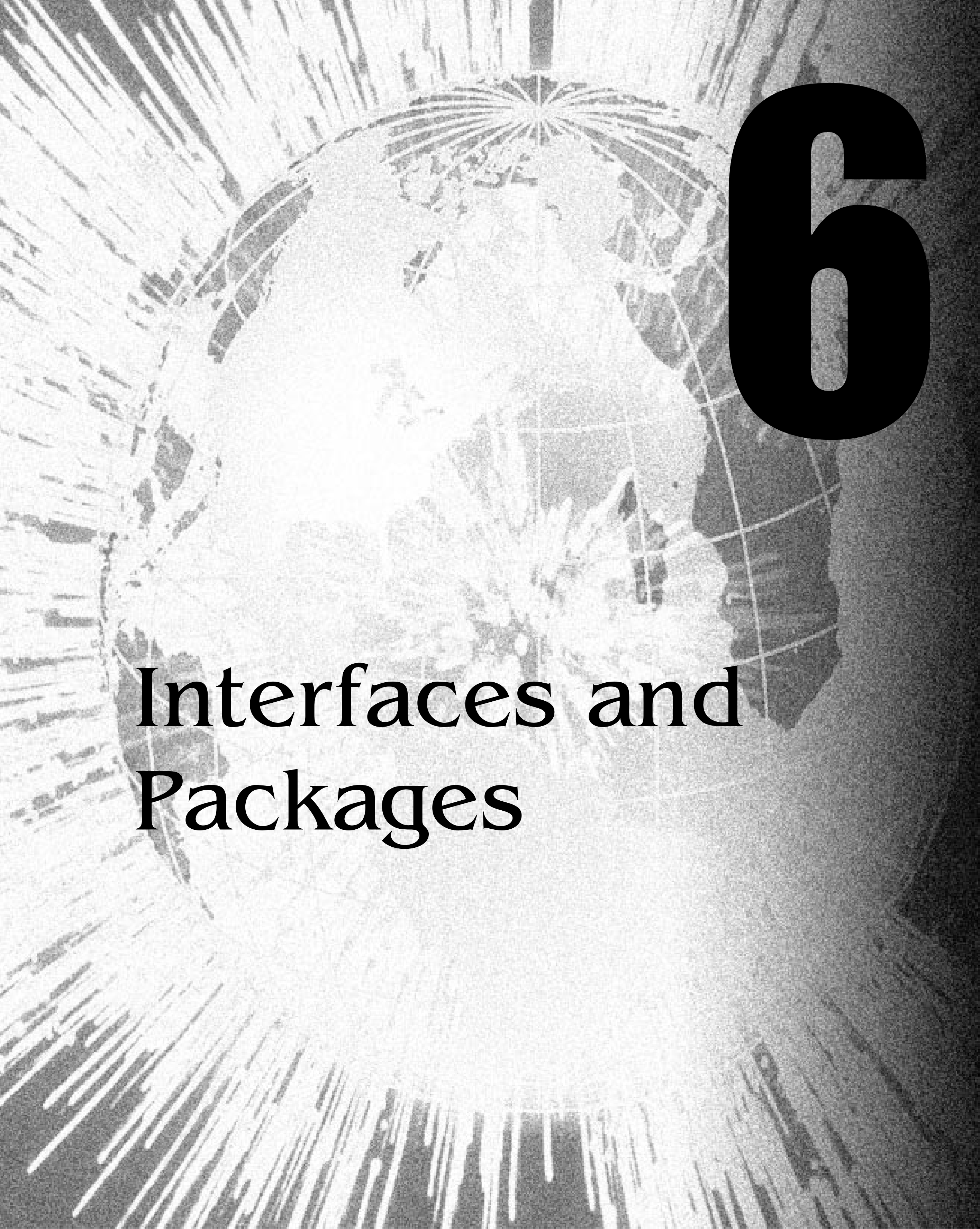


Figure 5.5
Widening and narrowing an instance of a class by using casts.

Here is an example of how you can cast references to objects between class types:

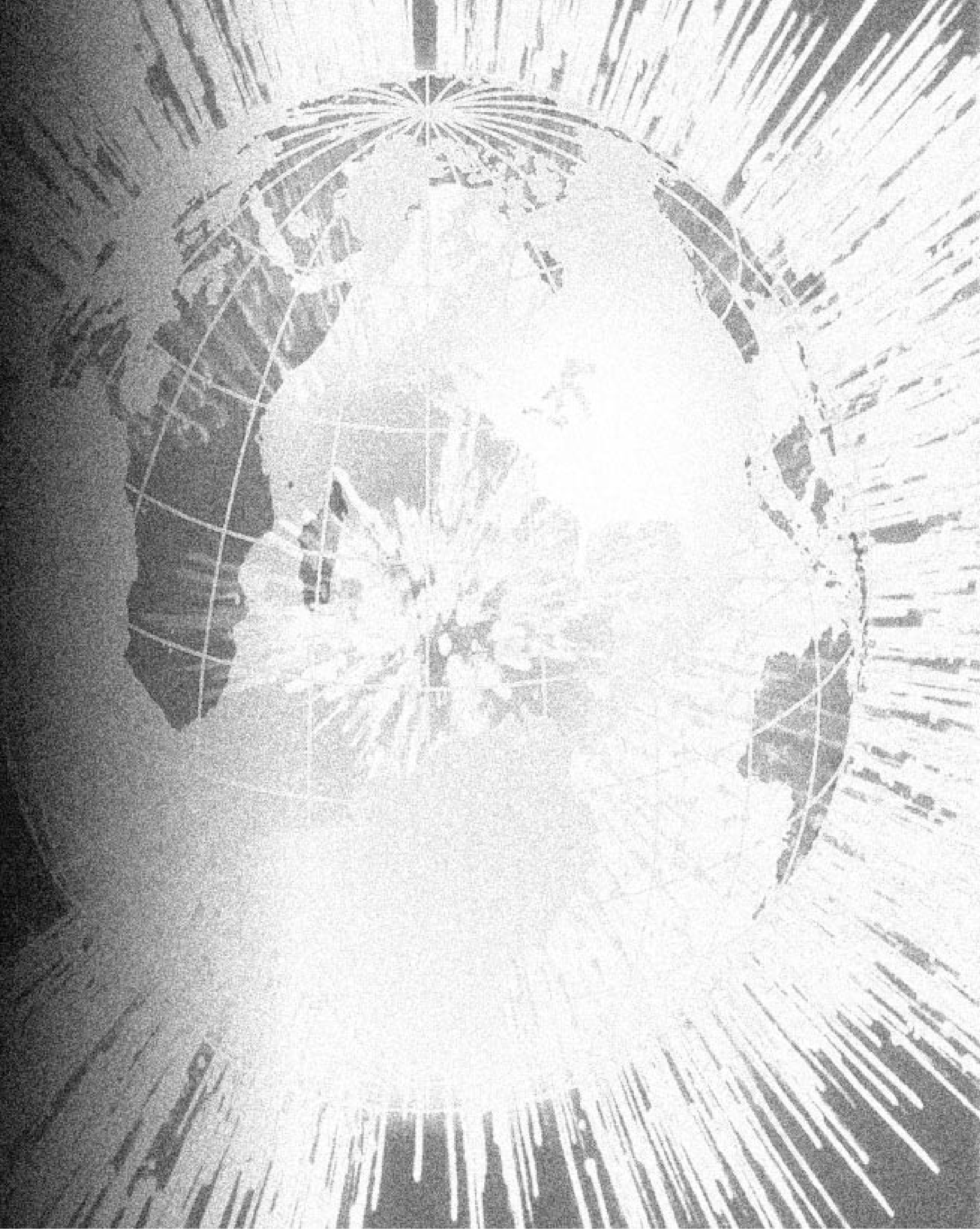
```
public class atom_ant {
   String  superhero = "Atom Ant";
   int height = 10;
   atom_ant() {
   }

   void print() {
      System.out.print (superhero + " is " + height + "\n");
   }

   public static void main(String arg[]) {

      atom_ant a1;
      a1 = new atom_ant();
      a1.print();

      proton_pal p1, p2;
      p1 = (proton_pal) a1; // Runtime error due to casting error
      p1.print();  // Unable to execute because of the previous line

      electron_enemy e1;
      e1 = (electron_enemy) p2; // Compile-time error due to casting to a
                                // sibling class
      e1.print();  // Unable to execute because of the previous line

      atom_ant a2;
      a2 = (atom_ant) p2;
      a2.print();
   }
}

class proton_pal extends atom_ant {

   String  superhero = "Proton Pal";
   int height = 1;

   proton_pal() {
   }

   void print() {
      System.out.print (superhero + " is " + height + "\n");
   }
}

class electron_enemy extends atom_ant{

   String  superhero = "Electron Enemy";
   int height = -1;
   electron_enemy() {
   }

   void print() {
      System.out.print (superhero + " is " + height + "\n");
   }
}
```

Here we've modified our previous **atom_ant** class to illustrate the basics of casting. Notice that two of the casts used will produce a runtime and compile-time error, respectively. (Thus, don't try to compile the code unless you remove the two illegal casts.) The first cast used in the **main**() method, **p1 = (proton_pal) a1**, produces a *widening* effect. Although this statement will compile, it produces a runtime error because the object **a1** cannot be expected to *grow* to accommodate the new variables and methods it references in **proton_pal**. The second casting statement used is a sibling cast: **e1 = (electron_enemy) p2**. It generates a compile-time error because an illegal reference to a *sibling* class, **electron_enemy** is used. This is due to the fact that the classes can have completely different variables and methods not related to each other. The last form of casting that is addressed in the **atom_ant** class produces a *narrowing* effect. In the statement, (**a2 = (atom_ant) p2**), the object **p2** references variables that are defined in the class, **atom_ant**, that is being casted. The reference is then past to the variable **a2**.



6

Interfaces and Packages



6 Interfaces and Packages

If you're ready to move beyond the stages of writing applets and simple standalone applications and applets, you'll find that Java's flexible interfaces and packages provide a welcome relief.

After writing a few applets and applications, you'll probably notice that the directory your classes are written to will start to become obscenely large. This is the downside of the way Java processes classes; but the good news is that Java provides two key features called *interfaces* and *packages* to help you organize your code. We put these two topics in a chapter by themselves instead of covering in detail in the previous chapter to emphasize how important they are. (Many Java books simply lump interfaces and packages in with classes, or they just skim over them—shameful!) As you start to work more with interfaces and packages, you'll discover a myriad of important program design issues that come into play which you'll need to master to use interfaces and packages effectively.

In this chapter you'll learn about:

- The basics of interfaces
- Techniques for implementing interfaces
- The hierarchical structure related to interfaces themselves
- Techniques for using casts with interfaces
- The basics of packages
- Techniques for creating packages
- Techniques for using Java's predefined packages

The underlying goal of this chapter is to help you transition from writing small standalone Java applications and applets to creating classes that can be used over and over. As you start to adopt this style of programming, you'll need the flexibility that interfaces and packages provide.

Understanding Interfaces

An *interface* is a collection of methods and variables that are declared as a unit but they are not implemented until a later stage. Basically this means that the code declarations placed in an interface serve as a shell so that you can create a truly *abstract class*. The goal behind an abstract class is to provide a mechanism so that you can define the *protocols* for a class—how a class should essentially communicate with other classes—early on in the development cycle. The upshot is that when you create your interfaces or abstract classes, you don't have to specify all of the details of how they will be implemented. This is saved for a later stage.

Before we jump in and start writing Java code for declaring interfaces, let's explore a few conceptual examples. The concept of abstract classes and interfaces is tricky to grasp at first. In fact, many experienced object-oriented programmers will tell you that they didn't quite master the concepts until they had written a number of programs. Fortunately, we can help you understand and use the techniques much quicker by providing the background information and conceptual models you'll need to apply them.

The simplest form of an interface involves adding methods and/or variables that are necessary to a particular class, but would disrupt the hierarchy of the class structure you are currently building for an application. If you chose to actually implement these elements in your class, they could limit how you planned to use the class to derive other classes. To make your classes more flexible, you can add interfaces to your classes in your hierarchy early on, so that the interfaces can be used in multiple ways to help construct the "behavior" of other classes that appear elsewhere in your class hierarchy. (If this discussion sounds like we are talking in circles—welcome to the world of interfaces! Hopefully these fine points will start to make sense to you in a moment when we look at a specific example.)

Let's assume that we need to develop an application that processes information about different forms of transportation. Figure 6.1 shows the hierarchy that could be used along with the list of components that could be implemented as interfaces.

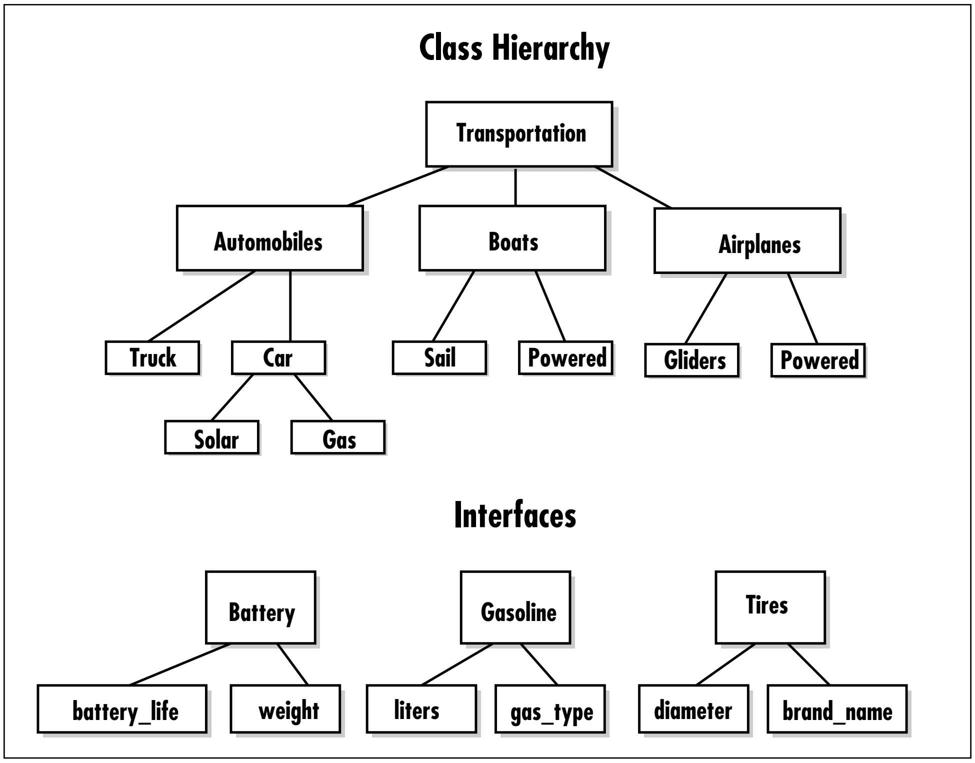


Figure 6.1
The hierarchy of classes for the transportation example.

As with typical class hierarchies, the classes shown in Figure 6.1 become more specific as they appear further down in the hierarchy tree. The interface components are advantageous when you have operations that are to be performed in one section of the hierarchy and not in the other areas. For example, the class *Car* has two subclasses: *Solar* and *Gas*. Let's assume you need to calculate the liters of gas that a gas car will use. You could include the methods and variables for performing this operation in the *Car* superclass, or even better, up two levels in the *Transportation* class, so that the *Powered|Boats* and *Powered|Airplanes* classes could use this code also.

Unfortunately, when you consider the scope of the application and all of the subclasses that inherit this useless information, you'd probably agree that this design approach is flawed. After all, the *Solar|Car* class would never calculate the liters of gas used and neither would the *Sail|Boats* or *Gliders|Airplanes* classes. A class that handles the gas calculating operation would be an incredible pain to incorporate at the *Transportation* level so that it could be designed into the hierarchy, and thus forcing all the subclasses to inherit all of its methods. If we were creating a small application that only required a few classes, this approach could be used. But if you are building an application that uses lots of classes from the beginning or you are expecting to expand the application in the future, this approach could quickly become a programmer's nightmare because limitations could arise from placing such restrictions early on.

In applications that have class hierarchies like our transportation example, interfaces become priceless because they allow us to "mix-in" classes into the application, adding them only where they become absolutely necessary. Another feature that enhances the interface's capabilities is the use of multiple implementations of interfaces per class. For example, in our transportation application, theoretically the *Car* class would be interested in the *Gasoline* interface, but the *Tire* interface could also be of use. An abstract class could incorporate both of these interfaces (the methods and variables that define them) at the *Transportation* level, but the *Boat* class would also be forced to inherit them. The *Boat* class never would have any use for the *Tire*'s methods or variables.

Design Issues with Interfaces

Interfaces will usually fall into a class hierarchy without any problems when you are creating small scale applications. They also help separate the design process from the implementation process because they keep you from having to combine the more abstract design issues with implementation details in one component. They also allow you to derive classes without relying on the more limited technique of *single inheritance*. If you recall from Chapter 5, a single inheritance model requires you to create class hierarchy trees by deriving one class from a single parent or superclass. Each class in the tree is created by using only data and operations that were defined in the levels above the current class.

Interfaces help you build class hierarchies that use more powerful object-oriented techniques like *multiple inheritance*. With interfaces, you can define classes that have multiple parents. You can incorporate interfaces into a hierarchical class tree to include new methods and variables without having to worry about disrupting your current implementation tree.

Declaring an Interface

Let's look at the basic declaration for an interface and then we'll show you the syntax for implementing an interface. After that, we'll introduce some code to illustrate how the transportation example we presented in the previous section could be set up. The basic syntax for declaring an interface looks similar to the syntax used for defining a Java class:

```
public interface InterfaceName {
   StaticVariables;
   AbstractMethods;
}
```

In this case, however, the **class** keyword is not used; the keyword **interface** takes its place. The *InterfaceName* serves as the interface indentifier name and the rules for specifying this name are the same as those used to name classes. The body of the interface declaration simply consists of the declaration of static variables and the names of one or more methods. Here's an example of an interface declaration:

```
public interface Gasoline {
// This variable is  defined as a constant
   public static final int Feet_in_Miles = 7245;

// A Method that is to be defined in a class
   void gas_type(String Name);
// Another method to be defined later
   void liters(int Amount);
}
```

Note that the variable **Feet_in_Miles** is declared as both static and final. This is required because all variables in interfaces *cannot* be changed. This type of declaration essentially turns the variable into a constant. If you leave out the **static** and **final** keywords, Java will force the variable to be declared as a constant. The two methods listed include both the method name and the method's parameter list. The actual code for the method will come when the interface is implemented.

Implementing an Interface

Declaring an interface is only half of the work. At some point, the interface must be implemented. This is accomplished by using the interface definition (or abstract class) to create a class. In a sense, a class can be "derived" using the interface shell. The syntax for implementing an interface is:

```
modifier class Identifier extends Superclass
implements InterfaceName [, InterfaceList ] {
   ClassBody;
}
```

In implementing an interface, you are essentially defining a special type of class. First, the class *modifier* is needed, followed by the **class** keyword. Then, the name of the class is provided. Next, the **extends** keyword is used followed by a superclass name to indicate that the class being defined is derived from a parent class. The **implements** keyword followed by the name of one or more interfaces, tells the Java compiler which interfaces will be used to implement the class. It is important to keep in mind that a class can implement more than one interface.

The class body consists of all of the variables and method definitions for the class. This is where all of the code must be placed for the methods that are listed in the interface declarations that are used. Using the **Gasoline** interface we declared earlier, here is a class called **Gas** that "implements" the **Gasoline** interface:

```
public class Gas extends Car implements Gasoline {
   int Miles;   // Variable declarations
   ...
   void gas_type(String Name) {
   ... // Add code  for this method
   }

   void liters(int Amount) {
   ... // Add code for this method
   }
}
```

Notice that this class is derived from a superclass named **Car**.

Now that we've covered the basics of declaring and implementing an interface, let's return to the transportation example we presented earlier. The first thing we need to do is declare the interfaces for the ones listed in Figure 6.1—**Gasoline**, **Batteries**, and **Tire**:

```
public interface Gasoline {
// This variable is  now a constant
   public static final int Feet_in_Miles = 7245;

// A Method that is to be defined in a calling class
   void gas_type(String Name);
// Another method to be defined later
   void liters(int Amount);
}

public interface Batteries {
// A Method that is to be defined in a calling class
   void battery_life(int Time);
// Another method to be defined later
   void weight(int Amount);
}

public interface Tires {
// A Method that is to be defined in a calling class
   void diameter(int Distance);
// Another method to be defined later
   void brand_name(int Name);
}
```

With these interfaces in hand, we're ready to create the two classes—**Gas** and **Powered**—each one will implement some of the interfaces in different ways. They will also show you how multiple interfaces can be used in a class definition:

```
public class Gas extends Car implements Gasoline, Batteries, Tires {

   int Feet_Traveled;
   int Miles_Traveled = 20;

   Feet_Traveled = Miles_Traveled * Feet_in_Miles;

   public static gas_type(String Name) {
      ... // Any functions that are to be performed with gas_type
      if(Name.equals("Diesel"))
         System.out.println("Ah, good power");
      if(Name.equals("Unleaded"))
         System.out.println("ok power");
      if(Name.equals("Leaded"))
         System.out.println("eh, clogged injectors");
   }

   public static liters(int Amount) {
      ... // Any functions that are to be performed with liters
   }

   public static battery_life(int Time) {
      ... // Any functions that are to be performed with battery_life
   }
   public static weight(int Amount) {
      ... // Any functions that are to be performed with weight
   }

   public static diameter(int Distance) {
      ... // Any functions that are to be performed with diameter
   }

   public static brand_name(int Name) {
      ... // Any functions that are to be performed with brand_name
   }
}

public class Powered extends Boat implements Gasoline, Batteries {

   int Feet_Traveled;
   int Miles_Traveled = 20;

   Feet_Traveled = Miles_Traveled * Feet_in_Miles;

   public static gas_type(String Name) {
      ... // Any functions that are to be performed with gas_type
      if(Name.equals("Diesel"))
         System.out.println("Required");
      if(Name.equals("Unleaded"))
         System.out.println("Not applicable");
      if(Name.equals("Leaded"))
         System.out.println("Not applicable");
   }

   public static liters(int Amount) {
      ... // Any functions that are to be performed with liters
   }

   public static battery_life(int Time) {
      ... // Any functions that are to be performed with battery_life
   }

   public static weight(int Amount) {
      ... // Any functions that are to be preformed with weight
   }
}
```

Notice that the **Gas** class is extended from the superclass **Car** and implements the interfaces **Gasoline**, **Batteries**, and **Tires**. In the class body of **Gas**, the methods declared for these interfaces are coded as well as other variables that the class needs, such as **Feet_Traveled** and **Miles_Traveled**. The **Boat** class, on the other hand, only implements two interfaces: **Gasoline** and **Batteries**. Notice that the **Boat** class implementation for the **gas_type**() method (declared in the **Gasoline** interface) differs from the version implemented in the **Gas** class.

Tips on Using Interfaces

The implements clause lists all of the interfaces that are to be included in the class definition. By referencing the interface, the class implementing it must restate the methods and their definitions in the body of the class. Constructors—the special methods that initialize new objects—may not be included in the interface declaration because interfaces can not instantiate new objects. Interfaces reference an object that is an instance of a class. By doing this they state that the object being referenced includes all the methods in the class that created the object.

The Art of Casting with Interfaces

If you recall from Chapter 5, where we covered casting between class types, we discussed how a cast can be used to change a reference to an object and not the actual object itself. We also showed you how instance variables are created and initialized to reflect the current reference to an object. This occurs when the names of the variable are the same in two classes—the one casting the object and the object the variable references. (If this is beginning to sound Greek to you, refer back to Chapter 5 for a refresher.) When we first introduced casting techniques for classes, this may have seemed to be a negative because a cast can relate to different instances of the same variable name, but actually a cast works to our advantage when used with interfaces.

Let's return to our **Gas** class example to see how we can use casts with interfaces. This time around **Gas** will reference the interfaces **Gasoline**, **Tires**, and **Batteries;** and **Gas** will create objects that reference the interfaces in different ways. Some of the references are correct and some of them will produce compile-time errors. We've included line numbers at the start of each line of code so that you can easily refer to the example in the discussion that follows:

```
1  public class Gas extends Car implements Gasoline, Tires, Batteries {
2
3 Gas     aCar    = makeGasCar();
4 Gasoline  aGasCar= (Gasoline) makeGasCar();    // Use cast
5 Tires     aTireCar  = (Tires) makeGasCar();    // Use cast
6
7   aGasCar.gas_type(Diesel);               // Valid
8   aGasCar.liters(5.8);                    // Valid
9
10  aTireCar.diameter(6.9);                 // Valid
11  aTireCar.gas_type(Unleaded);            // Not Valid
12
13  aCar.gas_type(Diesel);                  // Valid
14  aCar.weight(12.7);                      // Valid
15  aCar.diameter(6.9);                     // Valid
16  aCar.brand_name(Bridgestone);           // Valid
17
18 . . .    // Any functions that you would perform on the Cars created
19}
```

Let's break down what is going on here so that you can better understand some of the important and subtle Java programming techniques that are being used. Our example is only missing one thing that is not shown in the code—a method named **makeGasCar**() that creates and returns an object. Line 3 shows that an object is returned from the **makeGasCar**() method and is named **aCar** of type **Gas**. By assigning the returned value of **makeGasCar**() to an object variable of the type **Gas**, the object inheirits all the methods pertaining to the **Gas** class. This means it acquires all the methods relating to the class, its superclass, and *the interfaces the class implements*. In line 4, we acquire an object from the **makeGasCar**() method, but this time we cast it as type **Gasoline** from the interface **Gasoline**. This means that the object, **aGasCar**, inheirits all the methods that relate to the **Gas** class, its superclass, and *only the methods and variables declared in the interface* ***Gasoline***. As we'll see in a second, this means no methods or variables from the other interfaces are available for the object to reference. The next line does the same as the previous line, but the **Tires** interface is used in place of **Gasoline**.

Lines 7 and 8 both have the object **aGasCar** call the methods **gas_type**() and **liters**(), which were originally declared in the **Gasoline** interface. These method calls are valid because the correct parameters are used and the object **aGasCar** has access to both of these methods because of the cast that was used. In line 10, the **aTireCar** object references the **diameter**() method which is also valid because this object was created using the (**Tires**) cast and the **diameter**() method is declared within the **Tires** interface. But in line 11, the **aTireCar** object tries to call a method that is declared in the **Gasoline** interface. This produces a compile-time error because the object does not implement the interface **Gasoline**. Only the methods declared in the **Tires** interface are available from the object.

In the last section of the **Gas** class, lines 13 through 16, the object **aCar** may call any of the methods available to the interfaces because this object is an instance of the class **Gas** and is not casted to any particular class. This shows you the versatility possible in creating objects using interfaces.

Tips on Implementing Interfaces

If you refer back to our ticker tape applet in Chapter 2, you'll notice that it implements an interface named **Runnable** for the explicit function of moving (actually redrawing) text across the screen. When the applet is loaded into a browser, the browser checks to see if the object **ttapeThread**, which is an instance of the class **Thread** from a package that is imported into our class **TickerTape**, implements the **Runnable** interface. In this case, the browser detects the interface and uses the **run**() method declared in the class **Thread** during the operation of the applet:

```
// TickerTape Class
public class TickerTape extends Applet implements Runnable{
   ...
   // Change coordinates and repaint
   public void run(){
      while(ttapeThread != null){
         try {Thread.sleep(50);} catch (InterruptedException e){}
         setcoord();
         repaint();
      }
   }
   ...
}
```

This is a powerful feature for creating methods and variables in classes that can be set up with interfaces for future use, as long as the interface explains how information will be transferred to and from it. You don't need to allow others access to your original classes.

Java Note

Using the instanceof Operator

To detect if an object implements an interface, you can use the **instanceof** operator. This operator allows you to look at a group of objects to pick out which ones can perform certain operations. Here's an example:

```
if (ttapeThread iinstanceof Runnable) {
((Runnable)ttapeThread).run(); // performs this function only
                              // if the object ttape implements
                               //  the Runnable interface
}
```

In this case the **if** statement checks to see if the object **ttapeThread** is an instance of the **Runnable** interface. If it is, the **run**() method defined in the **Runnable** interface is called.

Creating and Using Packages

As you begin to design and code Java applications and applets that use multiple classes and interfaces, you'll need a way to organize your code so that you can easily update and reuse your growing library of classes and interfaces. Because the Java language is specifically designed to allow you to use classes and interfaces over and over, it's likely that you'll end up getting some of your class and interface names mixed up.

Furthermore, another programmer may design an excellent class that performs operations that you may want to use. Incorporating this class into one of your applications that already uses a number of classes could become difficult, especially if the class name conflicts with the name of a class you are already using. For example, you may have a custom print class named *Print* that performs certain functions for printing to the screen. After you've developed the class, another programmer might provide you with a class having the same name that prints a certain format to a printer that you need to support. You could actually use both of these classes even if they shared the name "Print"; however, they must be packaged in different groups so that the Java compiler can easily determine which one you want to use.

To help us combine classes into unique groups, Java supports the concept of *packages*. A package is essentially a device for grouping classes that you want to be labeled as a unit. You can actually combine any classes that you want into a single group. Usually, classes that share a common goal are combined in a class. For example, if you were creating a set of classes to handle drawing-related functions for a design application, you might create a package called *Draw* and place all of the related classes in this package.

You might have noticed back in Chapter 2 that some of the methods we implemented in the ticker tape applet were borrowed from classes or interfaces belonging to other packages. For example, one of the packages used was the **Applet** package—a package that Java provides, which contains all the necessary classes for creating an applet. A package is introduced to a class by using the **import** keyword in the beginning of a source code file. This will be covered in more detail later in the chapter. As you will see, classes and packages are segregated according to the functions they perform. This reduces the risk of having methods that share the same name interfere with each other. Here is a simple example of how you can implement methods that belong to different packages into a common class:

```
// TickerTape Applet

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

// TickerTape Class
public class TickerTape extends Applet implements Runnable {
   ...
   public void init(){
      ...
   }
   public void start(){
      ...
   }
   public void run(){
  ...
   }
   public void graphics() {
  ...
   }
   public void stop(){
      ...
   }
   ...
} // End TickerTape
```

This is the same applet that was used in Chapter 2. All of the methods declared in this example come from somewhere other than the current class. They have been *overridden* to perform a certain function specific to the operation of this applet. For example, the methods **init**(), **start**(), and **stop**() are defined in the **Applet** class that is contained in the java.applet package. The **run**() method is defined in the **Runnable** interface contained in the package java.lang.

Naming and Referencing Packages

Besides the fact that you may want to repeat a simple class name over and over, you'll want to create packages so that you can distribute your classes to other Java programmers. As with files on your computer, you list the directories in which they are contained to reference them. This creates a "path" for the Java compiler to follow so that it can locate designated classes and interfaces in your packages. Figure 6.2 shows an example of the directory hierarchy used to reference the package java.awt.image.

By convention, the first level of the hierarchy has been reserved for the name of the company that develops it. An example of this is **sun**.audio.AudioData—a

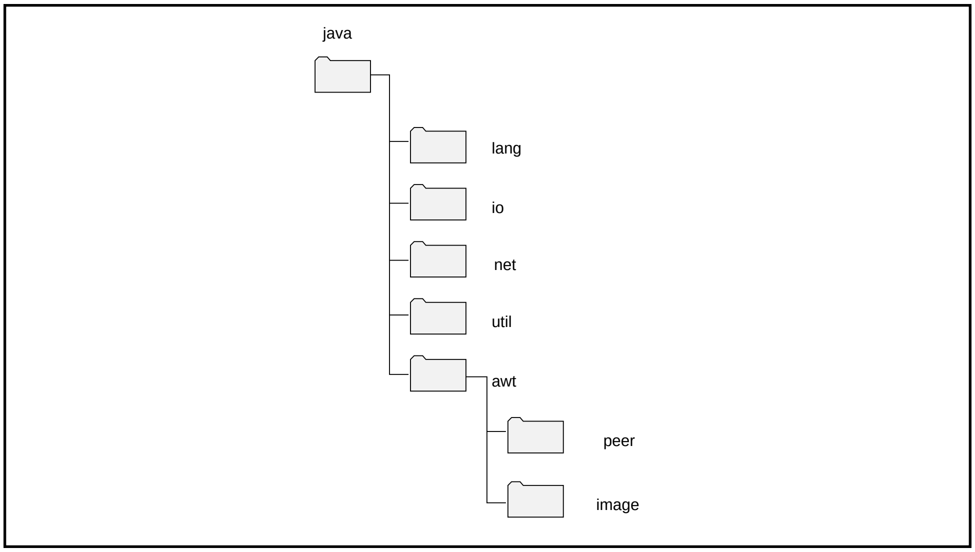


Figure 6.2

A graphical image of the hierarchy of java.awt.image and a call to the (import) java.awt.image on the other side.

package developed by Sun Microsystems. (Of course, as with every programming language, Java provides certain exceptions—one being the guideline for naming and referencing packages. For example, **java**.io.File was developed by Sun Microsystems, but this package is intended to be implemented by other companies as a foundation for building additional I/O packages.) The sections listed beneath the company name reference subdirectories that further specify where the class is located. For example java.**io**.File is a subdirectory that contains classes that relate to the input/output functions of Java. The extension **.class** has been omitted from the reference to the **File** class because interfaces and classes are the only format contained in a package and both end in the **.class** extension.

Uppercase vs. Lowercase Package Names

A specific format should be followed for naming packages and the classes that are contained within them. All package names and the directories that follow them should be specified using lowercase letters. On the other hand, the class and interface names you wish to reference within a package should be specified using an uppercase letter as the first character. This allows other programmers who use your packages to easily determine which components are directory names and which ones are class and interface names.

Declaration for Creating Packages

To create a package, the following statement should be placed at the beginning of a source file that contains a set of class definitions:

```
package PackageName;
```

Each class defined in the source file will automatically be added to the package having the name specified by *PackageName*. The *PackageName* will be created under the subdirectory you have defined in the CLASSPATH variable set in your environment. (The instructions for setting this environment variable are presented in the sidebar, *Setting Your CLASSPATH Environment Variable*.) As an example, assume that you have a source file that contains a set of classes that implement different types of airplanes. These classes could be combined into a single package named **airplanes** by placing the package statement at the beginning of each source file that defines a public class:

```
package airplanes;  // This statement must come first

// Provide source code for Glider class

public class Glider {
   ...   // Class definition
}
// The end of this source file


package airplanes;  // This statement must come first

// Provide source code for Single_engine class

public class Single_engine {
   ...   // Class definition
}
// The end of this source file


package airplanes;  // This statement must come first

// Provide source code for Twin_engine class

public class Twin_engine {
   ...   // Class definition
}
// The end of this source file
```

The actual *PackageName* is extended by the Java compiler by preceding it with the *CLASSPATH*. (Each subdirectory included in the path name is separated by a period.) The nice part is that you don't need to create the path for the package you define yourself; it is generated by the compiler at compile-time automatically.

Java Note

Interfaces and Public Classes

If you recall from Chapter 5, only one public class may be declared in any one source file. Only classes defined as public may be referenced from outside the current package. Otherwise, the classes not defined as public are used to support the public classes in the package.

In another example, if the package **coriolis.books.programming.java** is declared, the directory structure will turn out like this:

```
c:\java\lib\coriolis\books\programming\java
```

Essentially, what the Java compiler does when it encounters a statement like **package coriolis.books.programming.java** is create a new directory structure for *coriolis.books.programming.java* using the directory path specified by the CLASSPATH environment variable. It then places all of the compiled class code defined in the source file in the *java* directory. As the example above illustrates, the *CLASSPATH* would be:

```
c:\java\lib;
```

When the package is later referenced by a Java application, the compiler will know exactly where to look for each class that is referenced in the package.

Java Note

Saving Java Source Code Files

It is wise to save your Java source code in the directories containing your compiled class. This will allow you to later edit your source code if you wish, but more importantly, you won't have to worry about your class definitions being overwritten with identical names in the default directory where you create and save your source code (.java extension). You'll want to save the different versions of your source files because as you create more and more classes, the chance for repeating a class name becomes more common. For example, assume you have a *Spreadsheet* class that contains two classes; one that prints a graph and the other that prints a data sheet. Both classes perform very different operations, but both of them could be assigned the name *Print.class*. In doing so, you must take two steps in generating source code with identical class names because the classes will share the same working directory in most instances. If you placed a statement like this in the beginning of your source code

```
package acme.spreadsheet.graph;
```

the Java compiler would automatically place the *Print.class* in the directory graph but the original source file (Print.java) would still be paced in the working directory. The next step would be to place the source file in the same directory. This is because the next Print.java source file created (for example, the class responsible for printing the data sheet) will be saved in the working directory, causing the old file to be overwritten. If you later need to modify the class file, you will still have the original source code. The next source file should provide the statement

```
package acme.spreadsheet.datasheet;
```

at the beginning. Remember, you are required to manually move the source file to the appropriate directory.

Setting Your CLASSPATH Environment Variable

When your source code is compiled, the CLASSPATH environment variable specifies the default base directory for the packages you create. It also tells the compiler which directory path to search for the classes that are predefined. The order of directories defined by CLASSPATH determines the order in which the Java compiler will search for your classes. When a class is found that meets the requirements of the calling class, the compiler stops searching for a match. You should define the path of the default package that accompanies the Java Development Kit (JDK) and the temporary directory that you work from in this order. Here's an example:

```
CLASSPATH = c:\java\lib;.
```

The period sets the current directory you are compiling from. The first directory listed in the *CLASSPATH* also specifies where *your* package structure will begin.

Using Packages

The one feature that makes the Java language very powerful is that it lets you use the same code (classes) over and over countless times. This is accomplished by referencing classes that are contained in packages. To use classes that have already been created by you or other Java programmers, you need to reference the package(s) the classes are grouped in. You can do this in one of three ways:

- Specify the full package reference each time a class is used that is defined in an outside package. This approach is the most cumbersome and least often used. Here's an example:

```
airplanes.Twin_engine twin = new airplanes.Twin_engine("Beach", 1100);
```

In this case, the object variable **twin** is declared and initialized as an instance of a **Twin_engine** class which is included in the **airplanes** package. With this approach, each time a *Twin_engine* class is accessed, its corresponding package name must also be included.

- Import the actual class needed from the package it is defined in. As an example, we could rewrite the previous example by using this code:

```
import airplanes.Twin_engine;
...
Twin_engine twin = new Twin_engine("Beach", 1100);
```

Notice that once the desired class is imported, the name of the **airplanes** package is not needed to reference the **Twin_engine** class.

- Import all of the classes defined in a package. The syntax for doing this is illustrated with this statement:

```
import airplanes.*;
```

In this case, all of the public classes combined in the airplanes class, such as **Glider**, **Single_engine**, and **Twin_engine**, would be included.

Importing Packages Is Like Including C/C++ Header Files

If you are an experienced C / C++ programmer, you can think of the technique of importing a package as you would the technique of using an include file. Typically, you would use an include file to specify the names of function prototypes you wish to call that are defined in external files.

Every class defined in an external package that you want to reference by a class in your Java application or applet must be called directly or with a wild card (*) in the immediate directory. For example, if you refer back to our ticker tape applet presented in Chapter 2, we called an instance of the class **FontMetrics** that is contained in the java.awt package (directory). The **Applet** class imports the java.awt package with a wild card in the beginning of the code (e.g., **import java.awt.***;). The wild card tells the Java compiler to import *all* of the public classes in the java.awt directory into the **TickerTape** class. The compiler won't, however, import any of the classes that are contained in the peer or image directories beneath java.awt. To include the classes in those directories, you must reference the directories directly (e.g., **import java.awt.peer.*;** or **import java.awt.image.***;).

```
// TickerTape Applet

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

// TickerTape Class
public class TickerTape extends Applet implements Runnable {

   // Draw background and text on buffer image
   public void paintText(Graphics g){
      ...
      FontMetrics fmetrics = g.getFontMetrics();
      ...
   }
}
```

Declaration for Importing Packages

When importing a package into a class, the declaration must appear before any class declarations. The format for declaring a package is as follows:

```
import PackageName;
```

The *PackageName* represents the hierarchy tree separating the directories of the package with decimals. The java.lang package is automatically imported into every class that is created. If you look at the ticker tape applet presented in Chapter 2, you will notice that it does not import the java.lang package but uses many of the classes that are contained in the package. The classes **String**, **Integer**, and **Thread** are just a few of the classes that are called from this package.

```
// TickerTape Class
public class TickerTape extends Applet implements Runnable {
   // Declare Variable
   String inputText;
   String animSpeedString;
   int xpos;
   int fontLength;
   int fontHeight;
   int animSpeed;
   boolean suspended = false;
     . . .
}
```

Standard Java Packages

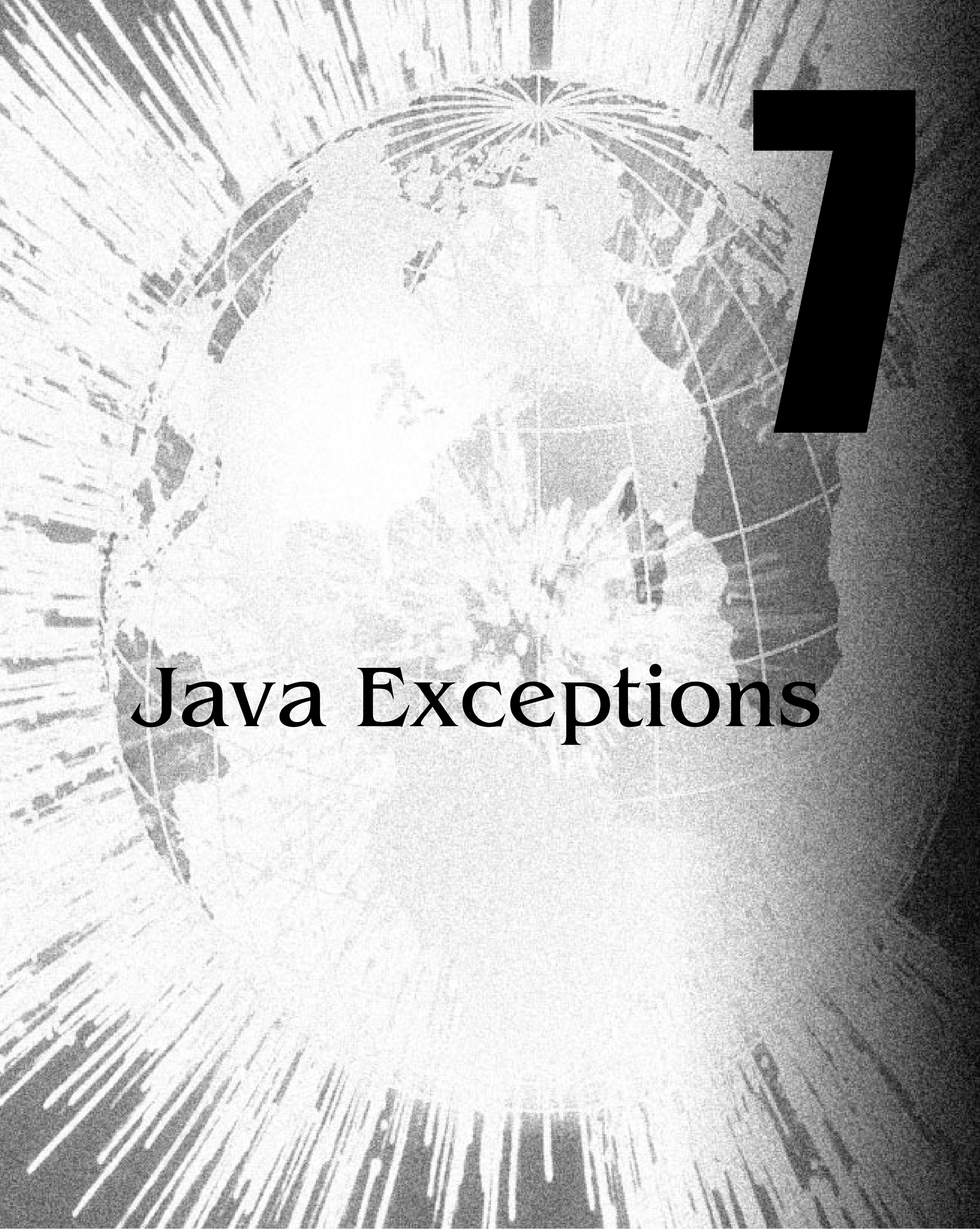
Since we created our first applet in Chapter 2, we have been using packages already defined by other developers including Sun Microsystems. These packages have been arranged by their category of usage. Table 6.1 shows the packages currently being distributed with the JDK.

Hiding Classes Using the Wild Card

We mentioned before that the Java wild card (*) will only allow you to bring in the public classes from an imported package. The benefit of this feature is that you can hide the bulk of your classes that perform support operations for your public classes. Users who use the public classes won't be able to look at the code or directly access the internal support classes.

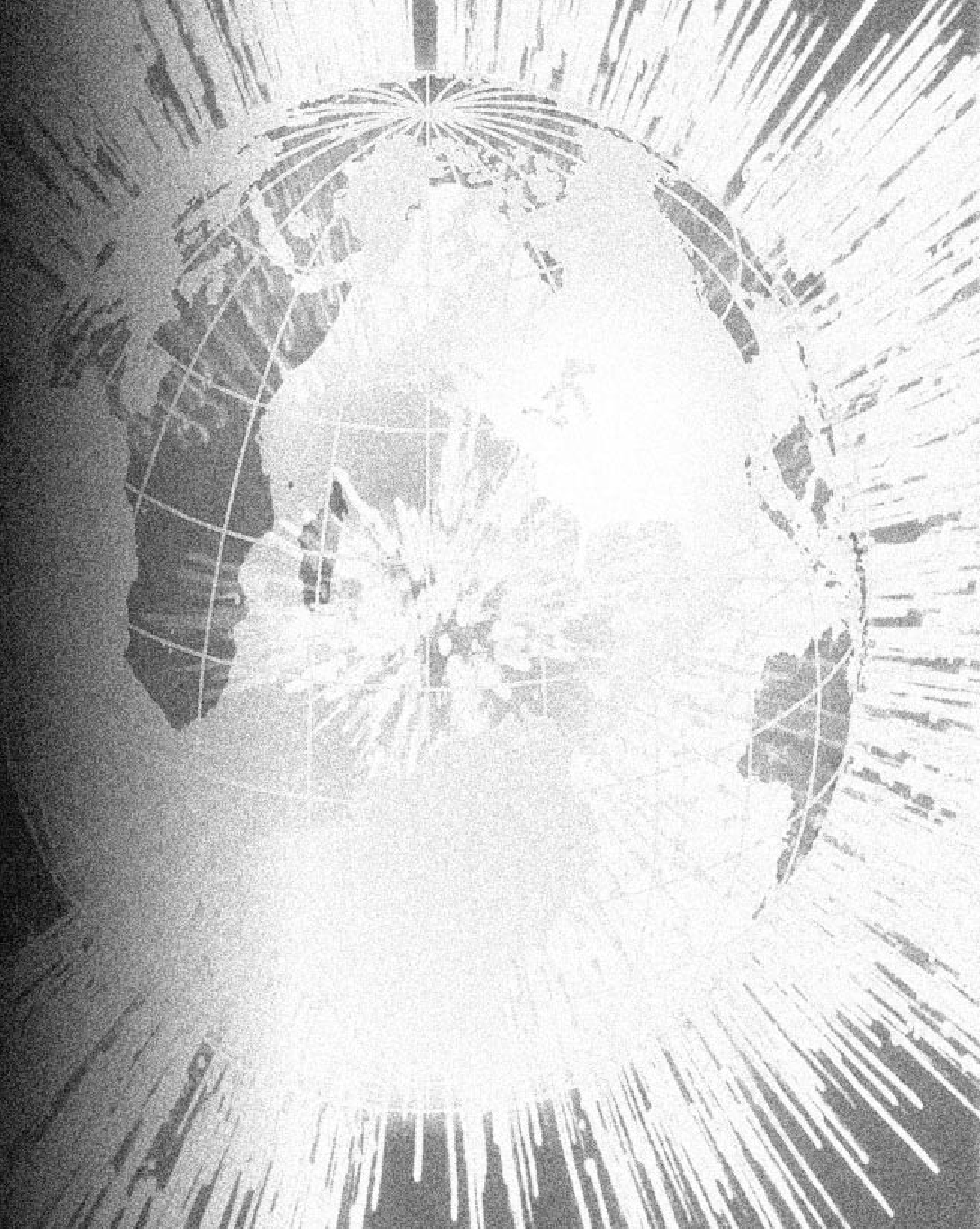
Table 6.1 Packages Distributed with the Java Development Kit

Package	Description
java.lang	Contains essential Java classes for performing basic functions. This package is automatically imported into every class that is created in Java.
java.io	Contains classes used to perform input/output functions to different sources.
java.util	Contains utility classes for items such as tables and vectors.
java.net	Contains classes that aid in connecting over networks. These classes can be used in conjunction with java.io to read/write information to files over a network.
java.awt	Contains classes that let you write platform-independent graphics applications. It includes classes for creating buttons, panels, text boxes, and so on.
java.applet	Contains classes that let you create Java applets that will run within Java-enabled browsers.



7

Java Exceptions



7

Java Exceptions

Are you tired of writing applications that mysteriously crash, leaving the user to give up in frustration? If so, you'll be glad to learn that Java provides a powerful feature called exceptions that automates the work of catching and handling compile-time and runtime errors.

One of the most difficult and time-consuming tasks of developing software involves finding and fixing bugs. Fortunately, Java provides some built-in features that lend a hand in the debugging process. As errors occur in a Java program, and we all know they will, you can use Java exceptions to provide special code for handling them.

Java programs can detect certain errors on their own and instruct the Java runtime system to take some predefined action. If you don't like the default operations that Java performs when it encounters certain errors, you can write your own custom error handling routines.

In this chapter we'll start by explaining the basics of exceptions. Then, we'll show you

- Why exceptions are important
- How to use **try** clauses to setup exceptions
- How to use **catch** clauses to trap exceptions
- When and how to use your own exceptions

Understanding Exceptions

Exceptions catch your errors and handle them gracefully so that your programs can continue to operate. In Java, this process is called *throwing an error*. This type of error handling can greatly benefit both you and the user of your application. After all, nobody likes an application that just crashes out of the blue. Unlike other languages, such as C, C++, and Pascal, where error detection and reporting can often double and even triple the size of an application, Java provides the means to detect and handle errors and at the same time reduce the overall size of your applications. The best part is that error handling in Java replaces the multiple "**if** this occurs **then** do this" statements so often found in programs written in languages like C.

Java's exceptions allow you to effectively code the main sections of your applications without you having to spend too much time writing code to detect and handle potential errors. As you'll learn in this chapter, exceptions create an object when an error occurs. The exception, which is a subclass of the *Throwable* class, *throws* an object, which is passed up through the hierarchy of the calling classes. The object will continue up through the classes until an *exception handler*—a method that deals with the exception—*catches* the object. This process is illustrated in Figure 7.1. If no exception handler is defined, a default exception handler is used to handle the error. This causes the error to be printed to the command line and the program will cease running.

Java Note

Using Java's Throwable Class

For a class to throw an error or catch one, it must be declared as a subclass of the Java **Throwable** class. All of the classes in the java package have incorporated the **Throwable** class in the package. This is why you don't see the **Throwable** class imported at the beginning of Java source code files. Although if you wish to refer to this class, you can directly import it into an application by including the statement:

```
import java.lang.Throwable;
```

Having error checking and error handling features in Java is important because Java programs, especially applets, run in multitasking environments. Often when an applet is executed in a Web browser like Netscape 2, other applets will be running at the same time. Each applet will have its own *thread* that the system

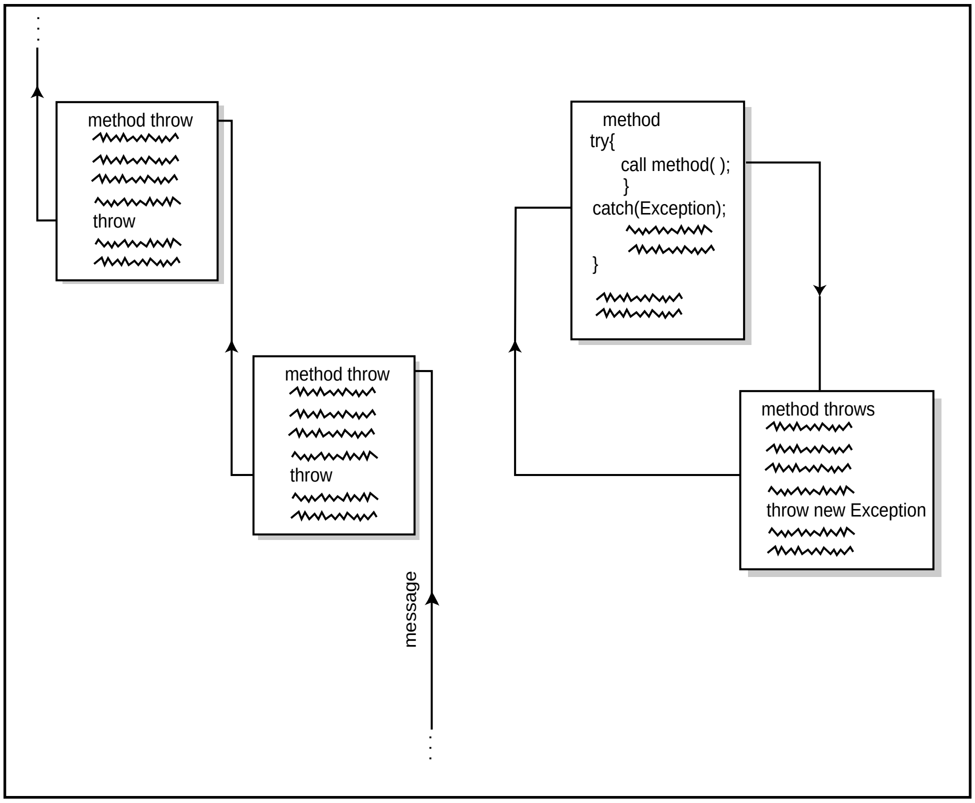


Figure 7.1
The process of throwing and catching an error in Java.

will control. If one applet causes a fatal error, the system could crash. With exceptions, on the other hand, critical errors are caught; and the Java runtime environment will know how to handle each thread that it must manage.

Do You Really Need Exceptions?

Even the smallest program can quickly evolve into a programmer's nightmare when you are trying to locate and fix a troublesome error. Why should you waste your time handling the possible errors that can occur? The fact is, you should only handle errors that you can do something useful with. Grabbing every little error in a program is useless if you do nothing intelligent with them. The best reason to declare exceptions is to provide a means for you and others that follow to understand where your code is having problems during critical operations.

For example, assume you have created a class to write data to a disk file. As your program is running, a number of errors could occur such as your hard disk being full, a file being corrupted, and so on. If you didn't have a way to catch errors like these at some point, the program might crash, leaving the user with nothing except a cryptic error message. Here's a Java program that performs a critical file operation but doesn't provide any error handling:

```
// This program will not compile because an IOException handler is
// expected by the Java compiler
import java.io.*;

public class WriteAFile extends Object {

   WriteAFile(String s) {
      write(s);
   }

   // Writes to a file
   public void write(String s) {        // I/O errors could occur here
      FileOutputStream writeOut = null;
      DataOutputStream dataWrite = null;

      // Begin to Write file out
      writeOut = new FileOutputStream(s);
      dataWrite = new DataOutputStream(writeOut);
      dataWrite.writeChars("This is a Test");
      dataWrite.close();
   }

     // Where execution begins in a stand-alone executable
     public static void main(String args[]) {
        new WriteAFile(args[0]);
     }
}
```

(Actually, this program won't compile just yet because the Java compiler expects to find an exception named *IOException*. We'll explain this in a moment.) The part of the code that could get you into trouble is the **write**() method. This method creates a new file output stream and attempts to write a character string to the stream. If the operation fails for one reason or another, a runtime error would occur, although an exception has not been setup to handle such an error.

To catch potential I/O problems, Java provides a built-in exception called *IOException.* This exception gets "thrown" whenever an I/O error occurs during a transfer to a device, such as a printer, disk drive, and so on. In our sample program, the Java compiler knows you must declare the exception because the**write**() method calls other methods that have declared an *IOException* to be *thrown* to calling methods. To remedy this, we could alter the **write**() method as shown here:

```
 // Begin to Write file out
 try {
   writeOut = new FileOutputStream(s);
   dataWrite = new DataOutputStream(writeOut);
   dataWrite.writeChars("This is a Test");
   dataWrite.close();
}
   catch(IOException e) {
}
```

Notice that two changes have been made. First, the entire block of code has been placed in a **try** { } clause. Essentially, this tells the Java environment to be on the "lookout" for errors that might occur as each method call is executed. The second change is the addition of the **catch**() method. This block of code performs the job of handling an I/O error that could occur with any of the calls contained in the **try** section. In our example, we are letting Java handle the work of processing an I/O error on its own by using the built-in *IOException,* and that's why no other code is provided with the **catch** statement.

These changes allow the code to compile and run. Unfortunately, they do not address any problems that could arise from actually writing a file to disk. In a perfect world, this code would be sufficient for our needs, but we don't live in a perfect world. For example, what if an error occurred while we were opening the file to be written because the disk is full or not even present? And even if the file could be opened, what would happen if an error occurred while we were writing the data to the file. All of these conditions are valid *exceptions* to writing a file to a disk. Unfortunately, you or others who use your classes might not detect them until it is too late. Remember, the advantage of using a language like Java or other flexible object-oriented languages is the ability to create robust code that can be reused by others.

Now, let's change the **WriteAFile** class once more to make it more robust. Don't worry about the syntax right now, we will discuss the details of implementing exceptions in the sections to follow.

```
// Writes to a file
public void write(String s) {
   FileOutputStream writeOut = null;
   DataOutputStream dataWrite = null;

   try {
      writeOut = new FileOutputStream(s);
      dataWrite = new DataOutputStream(writeOut);
   }
   catch (Throwable e) {
      System.out.println("Error in opening file");
      return;
   }
   try {
      dataWrite.writeChars("This is a Test");
      dataWrite.close();
   }
    catch(IOException e)  {
      System.out.println("Error in writing to file");
    }
}
```

This time around, we've included two **try** clauses. The first one checks the methods used to open the file, and the second one tests the methods used to write to the file and close the file. Notice how each of the **catch** statements specifies the type of object that it will *catch* or the exception that is thrown. We'll show you how to create custom error-handling routines later when we discuss the topic of c*atching*. For now it is important to realize that we have separated the possible errors we want to *catch* into two separate cases, opening and writing. By catching these errors, we have prevented the program from crashing as a result of not being able to open or write to a file. If errors like these are found, we could possibly ask the user to change disks or try again instead of having the user loose his or her data. In our case, we have simply written a message to the command-line telling the user where the operation has failed if an error occurs.

Defining a Try Clause

The **try** statement is responsible for indicating which section of code in a Java applet or application will most likely throw an exception. The syntax for using this statement is as follows:

```
  try {
   statement;
   statement;
  }
  catch (Throwable-subclass e) {
   statement;
   statement;
  }
```

For every **try** section, you must include at least one **catch** block that follows the **try** section. If an exception is thrown in the **try** section during the execution of the code, control flow is transferred to the matching section defined in the **catch** statement. If no match is found, the exception is passed up through the hierarchy of method calls. This allows each level to either handle the exception or pass it on. We'll cover this more when we present exception *throws*.

Using the catch Statement

If an exception is thrown during the execution of a **try** section, the flow of the program is immediately transferred to the corresponding **catch** block. The object, which is a reference to an instance of the exception class being thrown, is compared to the **catch**'s parameter type, also known as an *Exception Handler*. Here is the declaration for the **catch** block:

```
  catch (ExceptionType ExceptionObject) {
   statement;
   statement;
  }
```

The *ExceptionObject* reference parameter is a subclasses of the *Throwable class*. In most code, this reference is declared as **e** to distinguish it as a reference to an exception. In the event of an error, a subclass of the *Throwable* class is thrown, which is triggered by a violation of one of the procedures. This violation creates an object of the class type that the error originated from and is compared to the *Exception Handler* listed in each of the **catch** blocks that immediately follow the **try** section. The following code example illustrates how this process works:

```
import java.io.*;

// Reads from a file
public class  ReadAFile extends Object {
   ReadAFile(String s) {
      String line;
      FileInputStream fileName  = null;
      BufferedInputStream bufferedInput = null;
      DataInputStream dataIn = null;

      try {
         fileName = new FileInputStream(s);
         bufferedInput = new BufferedInputStream(fileName);
         dataIn = new DataInputStream(bufferedInput);
      }

      catch(FileNotFoundException e) {
         System.out.println("File Not Found");
         return;
      }
      catch(Throwable e) {
         System.out.println("Error in opening file");
         return;
      }

      try {
         while ((line = dataIn.readLine()) != null) {
            System.out.println(line + "\n");
         }
         fileName.close();
      }
      catch(IOException e) {
         System.out.println("Error in reading file");
      }
   }

   // Where execution begins in a stand-alone executable
   public static void main(String args[]) {
       new ReadAFile(args[0]);
   }
}
```

Here, the **try** block instructs the code to watch for an exception to be thrown from one of the methods contained in the block. The initializer that creates an instance of the class type **FileInputStream** named **fileName** is capable of throwing an exception in the event of an error. More specifically, the method contained in the class **FileInputStream** declares that an exception is to be thrown to the calling method. The topic of *throwing* exceptions will be covered later in the chapter, but for now you just need to know that you are required to address all exceptions thrown by handling them or passing them on. You *handle* the exception being thrown by placing *exception handlers*, declared in **catch** statements that the errors are then compared to. In the event of an error, the code will break from the normal flow of the code and immediately jump to the first *exception handler* that matches the class type defined in the **catch**. In the **ReadAFile**() method, the first **catch** identifies the **FileNotFoundException** class as a type that may be thrown upon instance of an error. This is followed by another **catch** identifying the **Throwable** class, which will act as a "catch all" for the exceptions being thrown. This match occurs because all exception classes are derived from the **Throwable** parent class.

When to Use the finally Statement

When an exception is "thrown," the compiler does not necessarily return to the exact spot it left off. The developers of Java realized that some procedures need to perform additional routines after an exception is handled, so they defined a **finally** statement. This statement instructs the Java Virtual Machine to return and finish any code after handling the exception before moving on. Here is the syntax required for using the **finally** statement:

```
try {
   statement;
   statement;
  }
  catch (Exception Handler) {
   statement;
   statement;
  }
  finally {
   statement;
   statement;
  }
```

The **finally** statement is not necessary to handle an exception, but it can be useful when you wish to handle an operation specific to a class. To see how it is used, let's expand our **WriteAFile** class to incorporate a **finally** statement that will create a backup file whether or not an exception occurs:

```
import java.io.*;

public class WriteAFile {
```

```
WriteAFile(String s) {
   write(s);
}

// Writes to a file
public void write(String s) {
   FileOutputStream writeOut = null;
   DataOutputStream dataWrite = null;

   try {
      writeOut = new FileOutputStream(s);
      dataWrite = new DataOutputStream(writeOut);
      dataWrite.writeChars("This is a Test");
      dataWrite.close();
   }
   catch(IOException e)  {
      System.out.println("Error in writing to file");
    }
   catch(Throwable e)  {
      System.out.println("Error in writing to file");
    }
   finally {
      System.out.println("\n\n.....creating a backup file.");
      try {
         writeOut = new FileOutputStream("MyBackup.sav");
         dataWrite = new DataOutputStream(writeOut);
         dataWrite.writeChars("This is a Test");
         dataWrite.close();
      }
      catch (IOException e) {
         System.out.println("Error in writing backup file");
      }
   }
}
   // Where execution begins in a stand-alone executable
   public static void main(String args[]) {
       new WriteAFile(args[0]);
   }
}
```

The Hierarchy of Exceptions

Like any other built-in Java classes you use in your applications, the standard exceptions are designed around a class hierarchy. Every exception is derived from the superclass **Throwable** as shown in Figure 7.2. The first subdivision is where the class splits into two categories: *Errors* and *Exceptions*. The *Exceptions* cat-

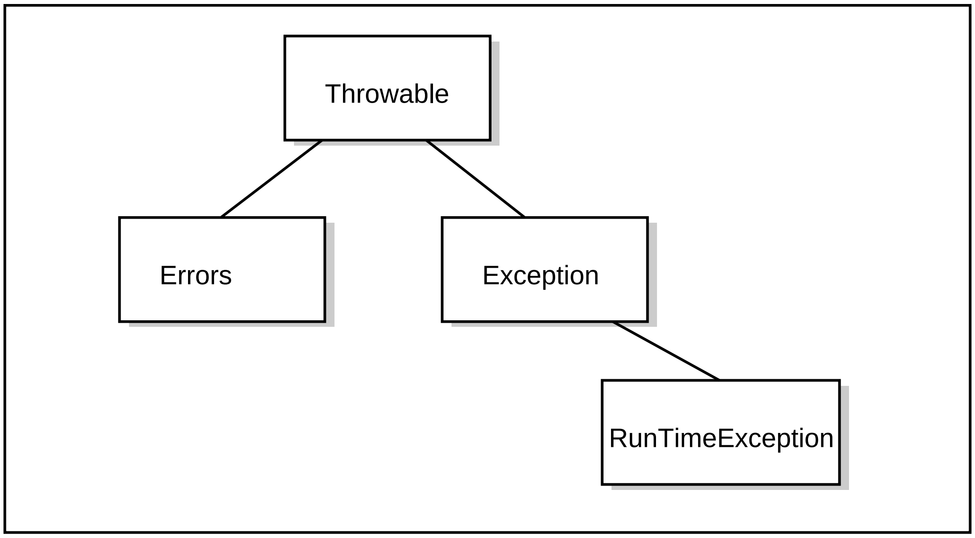


Figure 7.2
The Hierarchy of the *Throwable* class.

egory consists of the more common exceptions that you will want to "catch." The *Errors* category, on the other hand, consists of the low level exceptions that most programmers won't need to deal with.

The next major split occurs with the *Run-Time Exception* category, which is a subclass of *Exception*. Sun has arranged the hierarchy like this because they realized that by separating commonly used exceptions from specific exceptions, programmers would not be forced to include tons of handlers in their code.

Error Class

The exceptions included in the **Error** class are problems such as Linkage Error, ThreadDeaths, and other catastrophes that result in fatal errors. For the most part, these exceptions will not be handled by most applications. They are reserved for lower level programming tasks that require you to get into the internal workings of the language. For that reason, it is not a good idea to derive your own exception classes from **Error** unless you have a good working knowledge of Java. Table 7.1 provides a list of the key exceptions that are provided. All of the exceptions are defined in the Java Language Package java.lang, expect for **AWTError**, which is defined in the Advanced Windowing Toolkit Package, java.awt.

Table 7.1 Exceptions Included in the Error Class

Exception	Description
AbstractMethodError	This exception is thrown when your code attempts to call an abstract method.
AWTError	This exception is thrown when an error occurs in the Advanced Windowing Toolkit.
ClassCircularityError	This exception is thrown when a class hierarchy tries to establish a circle by linking itself to a parent class and the parent links itself to the child class or one of the children classes beneath it.
ClassFormatError	This exception is thrown as a result of an invalid file format being implemented.
IllegalAccessError	This exception occurs when an illegal access has been triggered.
IncompatibleClassChangeError	This exception is thrown when a class of incompatible types is changed.
InstantiationError	This exception occurs when a program attempts to instaniate an object from an abstract class or interface.
InternalError	This exception is thrown when an internal error occurs in the Java Virtual Machine.
LinkageError	This exception is thrown when the current class is dependant on another class, but the other class is not compatible with the current class.
NoClassDefFoundError	This exception is thrown when a class cannot be found by checking the path specified by the CLASSPATH environment variable or the current directory.
NoSuchFieldError	This exception is thrown when a specific field cannot be found.
NoSuchMethodError	This exception is thrown when a particular method cannot be found in the current class or one of its superclasses.
OutOfMemoryError	This exception is thrown in the event that no more memory can be allocated.
StackOverflowError	This exception signals that the stack has overflowed.
ThreadDeath	This exception is thrown in the thread that is being terminated. It should be handled when additional procedures are needed to be carried out before the **stop()** method has finished executing. If the exception is caught, it must be "rethrown" to actually finish killing off the thread. Because the exception is not required to be caught, it will not produce a command line message when it cycles up to the base class.
UnknownError	Bearing a close relation to mystery meat, this exception is triggered when a seriously unknown error occurs.
UnsatisfiedLinkError	This exception is thrown when a link to a library is unsuccessful.
VerifyError	This exception is thrown when the Java compiler is unable to verify if a linkage between classes is valid.
VirtualMachineError	This exception is thrown when the Virtual Machine has depleted its resources.

Exception Class

The exceptions included in the **Exception** class represent the most common errors that a programmer will want to deal with. For the most part, these excep-

Table 7.2 Exceptions Included in the Exception Class

	Defined in the Language Package (java.lang)
Exception	**Description**
ClassNotFoundException	This exception is thrown when the compiler is unable to locate a class in the current directory or the directory path specified by the environment variable CLASSPATH.
CloneNotSupportedException	This exception is thrown when an object attempts to clone an object that does not want to be cloned.
IllegalAccessException	This exception is thrown when a method is called from a class that does not have permission to do so. The access is determined by the modifiers used for the class and methods, resulting in the compiler being able to see or not see the calling method.
IllegalMonitorStateException	This exception is thrown in the event that a monitor is accessed that you do not own.
InstantiationException	This exception is thrown because of an attempt to create an instance of an abstract class or interface.
InterruptedException	This exception is thrown when a thread has interrupted the currently running thread.
NoSuchMethodException	This exception is thrown when a method can't be found in the calling class.
	Defined in the Utility Package (java.util)
Exception	**Description**
EmptyStackException	This exception is thrown in the event of an empty stack.
NoSuchElementException	This exception is thrown in the event that an enumeration is empty.
	Defined in the Input/Output Package (java.io)
Exception	**Description**
EOFException	This exception is thrown when an EOF is reached unexpectedly during input.
FileNotFoundException	This exception is thrown when a file is not found.
IOException	This exception is thrown in the event of an I/O error.
InterruptedIOException	This exception is thrown when an I/O operation has been interrupted.
UTFDataFormatException	This exception is thrown when a malformed UTF-8 string has been read in a DataInput stream.

continued

Table 7.2 Exceptions Included in the Exception Class (Continued)

	Defined in the Networking Package (java.net)
Exception	**Description**
MalformedURLException	This exception is thrown in the event of a bad URL.
ProtocolException	This exception is thrown when connect receives an EPROTO. This exception is specifically caught in the Socket class.
SocketException	This exception is thrown when an error occurs during the use of a socket.
UnknownHostException	This exception is thrown when there is an error in the connection to server from the client.
UnknownServiceException	This exception is thrown when a service is not identified.
	Defined in the Advanced Windowing Toolkit Package (java.awt)
Exception	**Description**
AWTException	This exception is thrown in the event of an error with the Advanced Windowing Toolkit.

tions can effectively be handled in the average program, to address problems between the user and the program. This class makes an obvious choice to derive your own personal classes from. Table 7.2 provides a list of the key exceptions that are provided in the **Exception** class.

Runtime Class

The exceptions included in the **Runtime** class are thrown during the execution of Java code. All of these exceptions are exempt from the restrictions of handling the exception at compile time. These exceptions are optional because of the need to keep Java code compact and easy to read. Table 7.3 provides a list of the key exceptions that are provided in the **Runtime** class. These exceptions are defined in the Language Package (java.lang).

Declaring a Method Capable of Throwing Exceptions

All methods capable of throwing an exception to a calling method must declare the exception in the method declaration. The type of exception being thrown to the calling method must be declared so that it understands how to handle the object it is receiving. The format for a method capable of throwing an exception is as follows:

Table 7.3 Exceptions Included in the Runtime Class

Exception	Description
ArithmeticException	This exception is thrown when an integer is divided by zero.
ArrayIndexOutOfBoundsException	This exception is thrown when an array is referenced outside the legal range.
ArrayStoreException `	This exception is thrown when you attempt to store an incompatible class or type in an array.
ClassCastException	This exception occurs when you attempt to cast an instance of a class to a subclass or a sibling class.
IllegalArgumentException	This exception is thrown when an invalid parameter is passed to a method that is outside the legal range or value.
IllegalThreadStateException	This exception occurs when a thread state is changed to an invalid state or one that the thread is currently in.
IndexOutOfBoundsException	This exception is thrown when an index to an array is outside the legal range.
NegativeArraySizeException	This exception occurs when an array of negative size is to allocated.
NullPointerException	This exception is thrown in the event that an object contains a null reference.
NumberFormatException	This exception is thrown when an invalid string to a number or a number to a string is encountered.
SecurityException	This exception is thrown when an applet attempts to breach the security defined by the browser.
StringIndexOutOfBoundsException	This exception occurs when a string is accessed outside the legal length of a string.

```
[Modifier] Return-type Identifier ([Parameter List]) [throws
  ExceptionName]
{
   Body;
}
```

The **throws** clause may list as many exceptions as will be thrown to it by separating each of them with a comma. For an example, let's take our **ReadAFileAFile** class to the next level and introduce a **throws** method:

```
import java.io.*;

public class wordProcessor extends Object {
   String fileName;

   void save(String fileName) {
      System.out.print ("Saving File Procedure\n");
      try {
              System.out.print ("Saving File " + fileName + "\n");
              ReadAFile aFile = new ReadAFile(fileName );

      }
      catch(FileNotFoundException e) {
         System.out.print ("Procedure to get another name and try again\n");
         // Procedure to get another name and try again
      }
      catch(IOException e) {
         System.out.print ("Procedure to try again\n");
         // Procedure to try again
      }
      finally {
         System.out.print ("Perform any cleanup\n" );
         // Perform any cleanup
      }
    }

   // Where execution begins in a stand-alone executable
   public static void main(String args[]) {
       wordProcessor myProgram = new wordProcessor();
       myProgram.save(args[0]);
   }
}

// Reads from a file
class  ReadAFile extends wordProcessor {

   ReadAFile(String s) throws  FileNotFoundException, IOException {
      String line;
      FileInputStream fileName  = null;
      BufferedInputStream bufferedInput = null;
      DataInputStream dataIn = null;

      try {
         fileName = new FileInputStream(s);
         bufferedInput = new BufferedInputStream(fileName);
         dataIn = new DataInputStream(bufferedInput);
      }
      catch(FileNotFoundException e) {
         System.out.println("File Not Found");
         throw e;
      }
      catch(Throwable e) {
         System.out.println("Error in opening file");
      }

      try {
         while ((line = dataIn.readLine()) != null) {
            System.out.println(line + "\n");
         }
     fileName .close();
      }
      catch(IOException e) {
         System.out.println("Error in reading file");
         throw e;
      }
   }
}
```

Notice that we didn't need to make many changes to the **ReadAFile** class used in this application. This class can quickly be made to pass exceptions as well as handle the ones that apply specifically to the class. The object**myProgram**, which is an instance of the class **wordProcessor**, calls the method **save**(). This method then calls the **ReadAFile**() method which declares that it will pass an exception to the calling method in the event of an error. Because the **ReadAFile**() method declares that it throws an exception, **save**() is required to address the exception that is being passed to it. If the method will not handle the exception, it must declare that it passes the particular exception on to the method that derived it:

```
ReadAFile(String s) throws  FileNotFoundException, IOException {
...
```

In our example, this line of code tells the method, **ReaAFile**(), that two exceptions, **FileNotFoundException** and **IOException**, can be thrown from the **try** block. This requires the **save**() method to handle them or declare the exceptions to be passed on to the method **main**() to deal with them.

Throwing Exceptions

The **throw** operator declares a particular exception may be thrown from the current method on to the calling method. This effectively passes the exception to the next method for it to deal with. In our previous example, the **ReadAFile** class declared that the method **save**() would pass two exceptions. In the code that follows, the example identifies which exceptions will be thrown.

```
try {
         fileName = new FileInputStream(s);
         bufferedInput = new BufferedInputStream(fileName);
         dataIn = new DataInputStream(bufferedInput);
      }
      catch(FileNotFoundException e) {
         System.out.println("File Not Found");
         throw e;
      }
      catch(Throwable e) {
         System.out.println("Error in opening file");
      }

      try {
         while ((line = dataIn.readLine()) != null) {
            System.out.println(line + "\n");
         }
     fileName .close();
      }
      catch(IOException e) {
         System.out.println("Error in reading file");
         throw e;
      }
```

The statement **throw e** specifies that the exception will be passed on for the calling method to deal with. Furthermore, much like error codes in other languages, messages can be passed along with the object to identify particular details to help process the exception. The following line of code shows how to throw an exception with a message attached:

```
throw new FileNotFoundException("MyFile.txt");
```

To reference the message in the calling method, you could simply call a **getMessage**() method to read the message attached to the file. The following code presents an example of this method:

```
catch(FileNotFoundException e) {
   System.out.println("The file " + e.getMessage +
     " was unable to be located.");
}
```

When to Catch and When to Throw

The issue of knowing when to catch an exception versus when to throw it to the calling method is typically a factor of what the exception does. If you refer back to our WriteAFile example, you'll see that we deal with a couple of exceptions. One of them caught an error that occurs in the event of an **IOException** by printing a message to the command line. This notifies the user of an error when writing to a file; but suppose **WriteAFile** class was a subclass in the hierarchy of the class **wordProcessor**. Here is a new version of our example that has been expanded to handle this:

```
import java.io.*;

public class WriteAFile extends wordProcessor{

WriteAFile(String s) throws IOException {
   write(s);
}

// Writes to a file
public void write(String s) throws IOException {
   FileOutputStream writeOut = null;
   DataOutputStream dataWrite = null;

   try {
      writeOut = new FileOutputStream(s);
      dataWrite = new DataOutputStream(writeOut);
   }
   catch (Throwable e) {
      System.out.println("Error in opening file");
      return;
   }
   try {
      dataWrite.writeChars("This is a Test");
      dataWrite.close();
   }
    catch(IOException e)  {
      System.out.println("Error in writing to file");
      throw e;
    }
}

}

import java.io.*;

public class wordProcessor extends Object {
   wordProcessor(String s) {
      new WriteAFile(s);
   }
   wordProcessor() {
      System.out.println("Create a backup file");
   }

   // Where execution begins in a stand-alone executable
   public static void main(String args[])  throws IOException {
      new wordProcessor(args[0]);
   }
}
```

Now, lets suppose we pass a filename to the **write**() method and it triggers the **IOException**. The **IOException** again writes a message to the command line, but notice it re-throws the exception to the calling method **wordProcessor**(). This method then allows for an additional message to be printed, in this case "Create a backup file." In place of the message, we could write an additional file to another location or do some other operation specific to the class**wordProcessor**. In addition, any other class could call the method and use it to fit its needs without being forced to perform an operation specific to **wordProcessor**.

Knowing When to Create Your Own Exceptions

The process of creating your own exceptions in Java is similar to creating other types of classes. Knowing when to create an exception is sometimes trickier than writing the exception itself. Here are some guidelines to help you create your own exceptions:

- Make sure you derive your new exception from the correct class. For example, if you create an exception that detects the corruption of a file, you'd want to subclass it beneath an **IOException**. Deriving the new exception from an exception like **ArrayOutOfBoundsException** would be pointless.
- If your code generates an error condition, you should handle it unless there is an obvious exception already created. For example, in the **ReadAFile** class we coded in this chapter, we used an exception to detect if a file cannot be found. On the other hand, if you created a class that determines whether a file has a virus or not, an **IOException** wouldn't necessarily be a wise choice. This would be a good place to subclass, however.
- Exceptions created in most applications should be derived from the **Exceptions** class. Only specific (lower-level) situations should require exceptions that need to be derived from the **Errors** or **RunTime** classes.

To create and use your exception classes, follow the same rules as standard classes. If you need a refresher, refer to Chapter 5. Here is a basic example of an exception:

```
public class AVirusDetectedException extends Exception {

   AVirusDetectedException(String fileName) {
  //perform some actions like read in libraries of virus types
    while(viruslibrary != null) {
      if (virus(fileName)) {
         throw new AVirusDetected(NameofVirus);
         //code after the throw operator is never executed.
       }
     }
  //this code is only executed if no virus is found
   }

   int virus(String fileName) {

  //perform some actions like read in libraries of virus types
  //test the byte code against patterns associated to viruses
    if (fileName = viruspattern) {
      return 1;
    }
  return 0;
  }
}
```

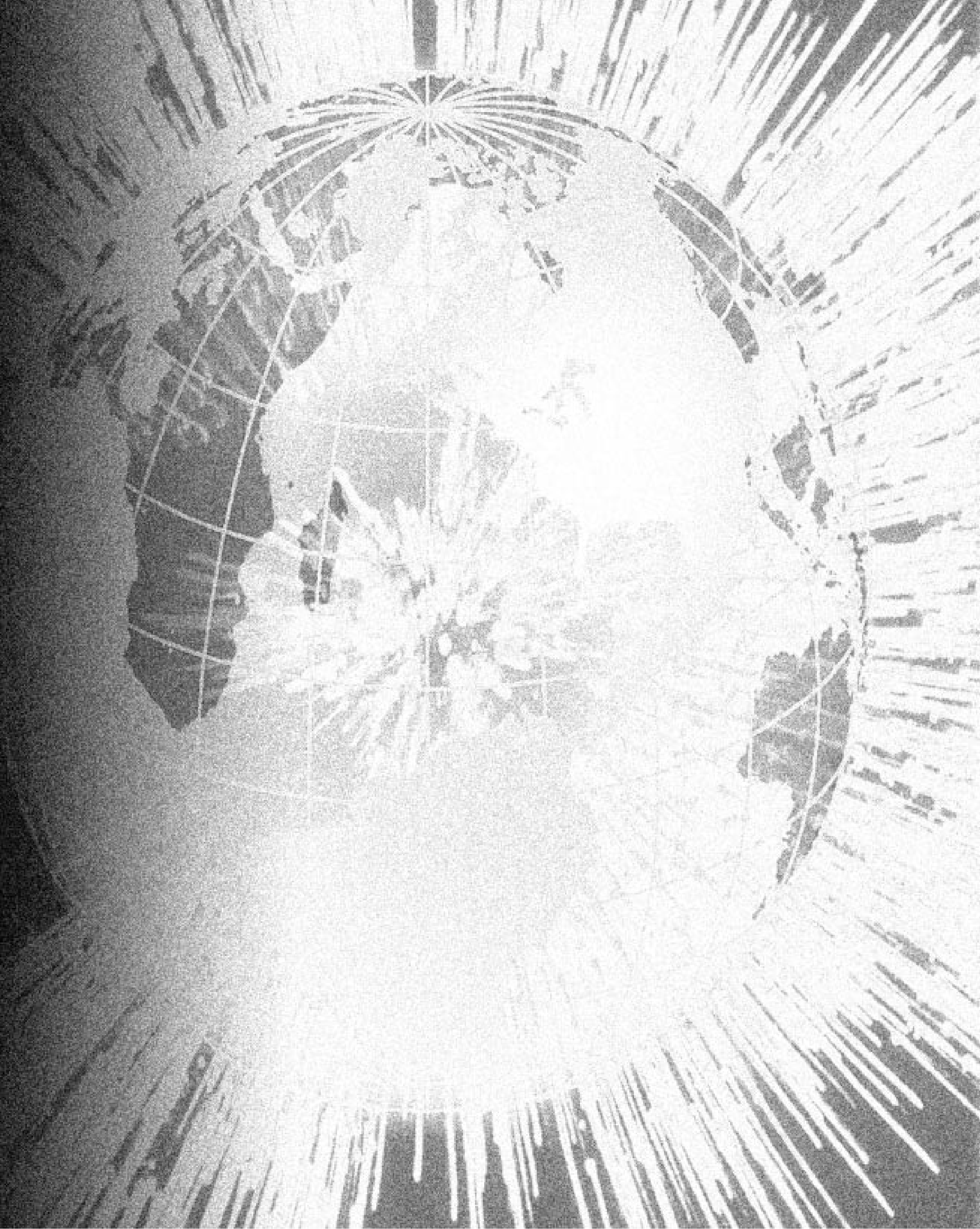
Trying to compile the source code will only result in an error. We subclassed the **AVirusDetectedException** from the **Exception** class because it will be triggered in the event of an I/O operation, but it does not fall under one of the predefined exceptions. This is used to demonstrate how an exception would look if it were created by you. To call this exception in your code, place the followingcode in your program:

```
try {
   if (file is questioned) {
      throw new AVirusDetectedException(fileName);
   }
} catch (AVirusDetectedException e) {
   System.out.println(e.getMessage + " has been found in " + fileName);
}
```

This tests whether a file was read from a disk drive, downloaded, and so on. An exception is then thrown in the event of a virus, as declared in the exception code above.

8

Threads



8 Threads

To create Java applets and applications that won't turn into system resource hogs, you'll need to arrange your programs into separate processes, which are called threads.

Imagine what our lives would be like if we could only do one thing at a time. You wouldn't be able to listen to music and program at the same time; and you definitely couldn't cook dinner, watch TV, and carry on a conversation with a friend. Although programming languages don't need to perform tasks like these, newer operating systems and environments like the Web are placing greater demands on programs, requiring them to handle multiple processes at the same time.

Java offers an advantage over most other languages because it was designed from the ground up to support multiple processes. When a Java applet runs in an environment like a Web browser, the browser can determine which parts of the program are separate processes and manage them to keep the program from draining the available system resources. As you gain more experience writing Java programs, you'll learn how to structure your programs to take advantage of the flexibility that multiple processes provide.

In this chapter we'll examine how threads are used to create multiple processes in Java programs. You'll learn how to create threads using either the pre-defined **Thread** class or the **Runnable** interface.

What Is a Thread?

One of the key jobs performed by the Java runtime system is to be able to handle programs that contain multiple processes called *threads*. If you've done any programming for an operating system such as Windows 95 or Windows NT, you've probably come across the concept called *multithreading.* The idea is to create applications that can handle different tasks at the same time, or at least be able to convince the user that multiple tasks are being performed. For example, a multithreaded version of an application that monitors the stock market would be able to download data from a central computer network, perform calculations in the background, and accept input from the user. Although only one thread can be executed at a time, the operating system that runs the program expertly divides up the different processes or threads, runs pieces of them, and jumps from thread to thread.

If you have ever loaded a Web page that contains multiple applets, you can see the process of multithreading at work. Assuming each applet is coded properly, your Web browser will make it look like each one is running at the same time. Of course, since most computers have only a single processor, the Web browser must be able to juggle each process so that each one gets its share of processing time.

To better understand how threads are coded, let's start with a simple example that contains a single process. Then we'll add to it so that you can see the effect that using threads has on the execution of the program. In the following **Hi3Person** class, the code executes in a linear fashion until the last line is reached, and the process ends:

```
public class Hi3Person {

   public static void main(String args[]) {
      Hi3Person people = new Hi3Person();

      people.hi("Person");
      people.hi("Person 2");
      people.hi("Person 3");

      System.out.println("Hello Everyone");
   }

   void hi(String who) {
      System.out.println("Hi " + who);
   }
}
```

Code execution begins by creating an instance of the class **Hi3Person.** Next, the three **hi**() methods are called. Each of these is executed one at a time, returning control back to the main body of the code. The final statement in **main**() writes the text “Hello Everyone” before the program ends.

As we introduce the concept of threads to **Hi3Person**, the linear path of execution will be disrupted. The program will be split into multiple processes, each responsible for writing to the screen. Let’s look at the new version of our code to see what is going on behind the scenes:

```
public class Hi3People implements Runnable {

   public static void main(String args[]) throws InterruptedException {

      int i = 0;

      Hi3People person = new Hi3People();
      // Create thread #1
      Thread aThread = new Thread(person, "Person 1");
      // Create thread #2
      Thread anotherThread = new Thread(person, "Person 2");

      aThread.start();                // Start the first thread
      anotherThread.start();          // Start the second thread

      // Body of main program
      while ((aThread.isAlive()) || (anotherThread.isAlive())) {
        i++;
      }

      // Executes after both threads have finished
      System.out.println(i + "\n");
      System.out.println("Hello Everyone");

      aThread.stop();               // Stop the first thread
      anotherThread.stop();         // Stop the second thread
   }

   public void run() {
      System.out.println("Hi " + Thread.currentThread().getName());
   }
}
```

(For now, don’t worry about the syntax used to create the threads that are used. We’ll explain the techniques for implementing threads a little later in this chapter.) Notice that the **Hi3People** class initiates two threads that run concurrently as our application continues on. After each thread has been created, the **start**()

method of the thread is called, which tells the Java interpreter to begin processing this thread. The **main()** method is responsible for setting up each thread and determining when the threads are finished. This is necessary because our program needs to know when it is safe to execute the code starting with the line:

```
System.out.println(i + "\n");
```

Otherwise, the program will end before the threads have finished and it will hang. In this case, we have placed a **while** loop to count continuously during the execution of the threads:

```
while ((aThread.isAlive()) || (anotherThread.isAlive())) {
   i++;
}
```

If you compile and run this example, you will notice that the value stored in the variable **i** will change after each execution of the code. This variable stores the number of times the **while** loop repeats during the life of *both* threads. The fact that this value changes illustrates the control that Java can have as it executes programs that are divided up into separate processes. Running a single- or multi-threaded program is not the only task that the Java runtime system must perform. Java also has its own internal threads that it must perform to manage tasks such as garbage collection.

Java Note

To Thread or Not to Thread

The trick to programming with threads is knowing when you need to use them. Threads have the ability to hinder as well as help the execution of programs. You should only use threads when two or more processes need to exist at the same time. For example, in a windows environment, multiple windows can be opened at once *to give the impression* that two operations are occurring at the same time. These are examples of threads being implemented into real world application. Most Java programmers implement threads for building interface components by using the AWT package, which we'll explore in Chapter 9.

Keep in mind that when an application runs, multiple processes don't actually run at the same time. It's the operating system's job to give the user the impression that everything happens at once. Even in a multithreading environment like Unix, processes do not occur at the same time. As Figure 8.1 shows, the illusion of processes running concurrently is created by carefully and quickly cycling instructions through a channel. The Java Virtual Machine handles its own type of processor management by determining what executions will occur and in what order.

When you add multiple threads to your program, you effectively set up an events manager. You must manage how the instructions from your application are handled. You determine what process receives more or less time and when to change the focus of the program. As you will see later in this chapter, you can make your application appear transparent or painfully slow just by the way you schedule your threads.

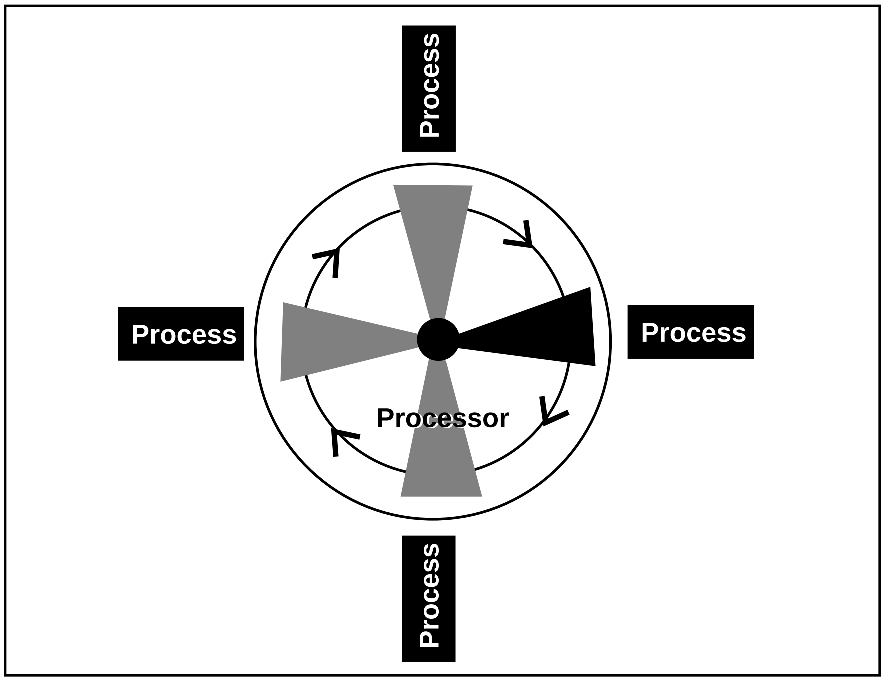


Figure 8.1

The technique of managing multiple processes.

Creating a Thread

Before you can create a thread, you must set up a class to handle the thread. This is done in either of two ways: extending a class (subclassing the **Thread** class) or implementing an interface. As you'll see in the next two sections, the approach you use will depend on your own needs.

Subclassing the Thread Class

The most obvious way to create a thread is to subclass the **Thread** class that Java provides. This approach allows you to override the methods in the **Thread** class to perform the functions you define for your program. Here is the syntax for creating a class using the **Thread** class.

```
[Modifier] class ClassName extends Thread {
   ClassBody;
}
```

Here's an example Java application that utilizes the **Thread** class as a superclass:

```
public class Hiagain extends Thread {

   public static void main(String args[]) {

      int i =0;

      Hiagain tony = new Hiagain();  // Create a new object
      Thread t1 = new Thread(tony);  // Create each thread of the object type
      Thread t2 = new Thread(tony);

      t1.start();    // Start each thread
      t2.start();

     while ((t1.isAlive()) || (t2.isAlive())) {
        i++;
      }

      System.out.println("Hello Everyone");
      t1.stop();     // End the threads
      t2.stop();
   }

   public void run() {
      System.out.println("Hi");
   }
}
```

The class **Hiagain** subclasses the **Thread** class and overrides the **run**() method defined by the **Thread** class. Because **Hiagain** is derived from **Thread**, it inherits all of the methods defined in **Thread** including **start**(), **run**(), and **stop**(). The original versions of these methods defined in **Thread** are used except in the case of the **run**() method, which has been overridden. Because this method tells the thread which operations to perform after the thread has been started, it typically needs to be overridden. The key methods that are defined in the**Thread** class will be presented later in this chapter.

Implementing the Runnable Interface

When designing class hierarchies for your Java applications or applets, situations arise where subclassing from the **Threads** class is just not possible. In such cases, you can implement the **Runnable** interface to avoid a conflict. To declare a class **Runnable**, follow the template shown here:

```
[Modifier] class ClassName extends SuperClass implements Runnable {
   ClassBody;
}
```

The advantage of this approach is that you can create a new class that is both derived from another class and uses the methods defined by the **Runnable** interface. Of course, you will then be required to go along with the implementation created by the designers of this interface. Let's revisit our ticker tape applet introduced in Chapter 2 because it provides a good example of an implementation of the **Runnable** interface:

```
// TickerTape Class
public class TickerTape extends Applet implements Runnable{

   ....

   // Initialize Applet
   public void init(){
   ....
   }

   ....
// Start Applet as thread
   public void start(){
      if(ttapeThread == null){
         ttapeThread = new Thread(this);
         ttapeThread.start();
      }
   }
   ...
   // Change coordinates and repaint
   public void run(){
      while(ttapeThread != null){
         try {Thread.sleep(50);} catch (InterruptedException e){}
         setcoord();
         repaint();
      }
   }
....
// Stop thread then clean up before close
   public void stop(){
      if(ttapeThread != null)
         ttapeThread.stop();
      ttapeThread = null;
   }

} // End TickerTape
```

As with all applets, you must use the **Runnable** interface to implement threads. (You are forced to subclass the **Applet** class to perform the basic operations of initializing and starting your applet.) The reason you would want to implement threads in an applet is to reduce the processor load for performing operations that occur over and over. One example would be the graphics routine in our ticker tape applet that updates the screen to make the text appear as if it floats across the screen. The **run**() method is coded to redraw the screen and then set the thread to sleep for a specified amount of time:

```
// Change coordinates and repaint
public void run() {
   while(ttapeThread != null) {
      try { Thread.sleep(50); } catch (InterruptedException e) {}
      setcoord();
      repaint();
   }
}
```

The reason for putting the thread to sleep is covered in more detail later in the chapter. The two methods responsible for moving the text are **setcoord**() and **repaint**(). They are executed as long as the thread exists.

Initializing a Thread

Before you can use a thread, you must initialize it by creating an instance of the **Thread** class. The best way to do this is to use the constructor for the **Thread** class. The simple form of this constructor is:

```
Thread Identifier = new Thread();
```

A few other variations of this constructor are also supported:

```
Thread(ObjectReference);
Thread(StringName);
Thread(ObjectReference, StringName);
```

In the first example, a parameter that references the object to be used to create the thread for is provided. We actually used this type of constructor in the**Hiagain** class example presented earlier:

```
Hiagain tony = new Hiagain();    // Create a new object
Thread t1 = new Thread(tony);    // Create a thread of the object type
```

The next two constructors allow you to pass a string to create references to individual threads, which can then be used as symbolic references. We'll show you how to use this feature later in this chapter to keep track of multiple threads created from the same class instance.

If you return to the ticker tape applet outlined above, you'll see that the thread is created in the **start**() method for the applet:

```
// Start Applet as thread
public void start() {
   if(ttapeThread == null) {
      ttapeThread = new Thread(this);
      ttapeThread.start();
   }
}
```

In this case, notice the **Thread**() constructor is passed the **this** parameter. Using **this** allows us to tell the constructor the name of the class that implements the **Runnable** interface. The new thread object that is created is assigned to the

variable ttapeThread. Once the thread has been initialized, the **start**() method for the thread can be called by using the statement **ttapeThread.start**().

Who Goes First; Who Finishes Last?

Although only one thread can be started at a time, don't expect the first one called to always be the one that finishes first. In fact, the order in which threads are called won't necessarily determine the order in which they finish. In most cases, it is impossible to determine which thread will finish first. Let's return to our simple **Hi3People** class introduced in the beginning of this chapter, to see how the execution of threads can be hard to predict:

```
public class Hi3People implements Runnable {
  public static void main(String args[]) throws InterruptedException {

   int i1 = 0;
   int i2 = 0;

   Hi3People person = new Hi3People();
   Thread aThread = new Thread(person, "Person 1");
   Thread anotherThread = new Thread(person, "Person 2");
   aThread.start();
   anotherThread.start();
   while ((aThread.isAlive()) || (anotherThread.isAlive())) {
      if (aThread.isAlive()) { ++i1;}  // Counter for the first thread
      if (anotherThread.isAlive()) { ++i2;} // Counter for the second thread
   }

   System.out.println("The time for Person1 is " + i1 + "\n");
   System.out.println("The time for Person2 is " + i2 + "\n");

   aThread.stop();
   anotherThread.stop();
   }

   public void run() {
      System.out.println("Hi " + Thread.currentThread().getName());
   }
}
```

First, notice the types of constructors that are used:

```
Hi3People person = new Hi3People();
Thread aThread = new Thread(person, "Person 1");
Thread anotherThread = new Thread(person, "Person 2");
```

Both the object name (**person**) and a unique string is passed to each call to **Thread**(). Since both threads are created using the same object, the string is passed to assign each thread its own unique name. In the **run**() method of the program, the **getName**() method is used to display the name of the current thread. A companion method named **setName**() is provided in the **Thread** class for setting the name of a thread.

Next, by changing a few lines of code, we converted our while loop to count the time it takes to process *each* thread instead of counting the time it takes to process the two together. You would need to run this code about 10 to 15 times before running across an instance where the first person beats the second one. This is due to the fact that Java's scheduler is still in a beta version. (Hopefully, Sun will consider implementing a method for determining the order in which threads are processed.) The scheduler is responsible for determining what threads may run and which ones must wait in the queue. This process is determined in either one of two methods: priority or first in first out (FIFO).

Priority versus FIFO Scheduling

When a thread is processed, it is automatically thrown into a queue. Then, the thread must wait until its turn is up. This process of waiting is called *blocking*. The implementation of the thread that is ahead of the one waiting will determine if a thread will wait until the current thread has completed. This method of waiting is referred to as *First in First Out* (*FIFO*). Like everything in this world, there are exceptions to the rules. If a thread has priority over another thread, it switches places with the thread. This process continues up the queue until a thread reaches an equal or greater priority or it is executed. The threads that were pushed aside may not continue their operation until all the higher priority threads either step aside or finish. The most common case of this is the Garbage Collector, the thread that runs in the background and has the lowest priority.

To control how the priority is set for a thread, the **Thread** class provides the following three variables:

- MAX_PRIORITY
- NORM_PRIORITY
- MIN_PRIORITY

Each of these variables holds integer values that specify a thread's priority level. For example, **MAX_PRIORITY** stores a number that indicates the maximum allowable value for a thread's priority. To set or retrieve the current priority setting of a thread, the **Thread** class provides the **setPriority**() and **getPriority**() methods. In setting priorities, you can use one of the three priority instance variables. For example, the following method call would set a thread's priority to the value contained in the NORM_PRIORITY variable:

```
Thread aThread = new Thread(person, "Person 1");
aThread.setPriority(aThread.NORM_PRIORITY);
```

Controlling the Life of a Thread

A thread is like a human or plant life form; it has a beginning, middle, and an ending. During these stages, a thread will take many different paths depending on the objective to be reached. Figure 8.2, shows the cycle of life that a thread can *possibly* take.

The stages of life of a thread are determined by a set of pre-defined methods that can be overridden to perform certain tasks.

The start() Method

This method starts the "birthing" process for a thread. By default, it sets up a few initializations and calls the **start**() method. You can override it to initialize your own variables and then call the **start**() method:

```
// Start Applet as a thread
   public void start() {
      if(ttapeThread == null) {
         ttapeThread = new Thread(this);
         ttapeThread.start();
      }
   }
```

In this example the **start**() method checks to see if the thread **ttapeThread** exists. If it doesn't, it creates an instance of the **Thread** class and assigns the variable **ttapeThread** to it.

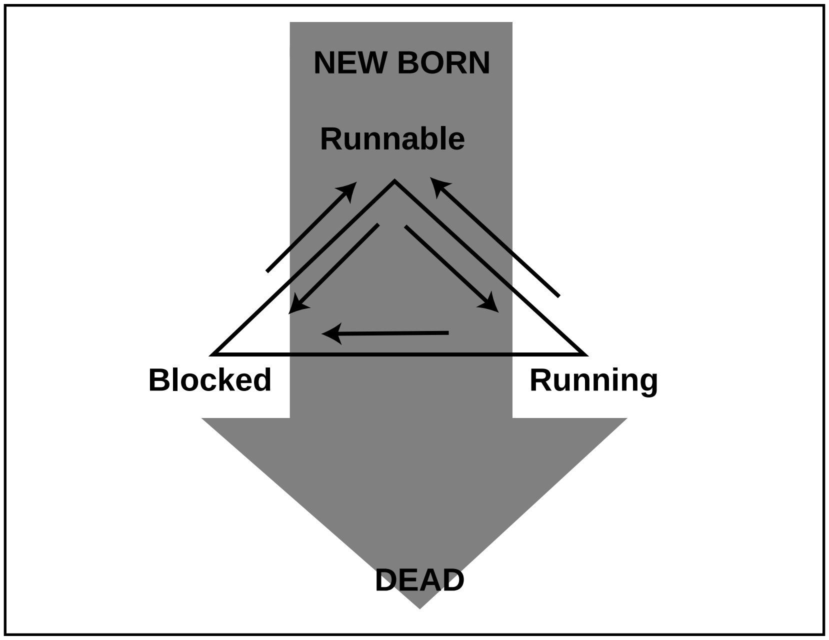


Figure 8.2
The cycle of life pertaining to a thread.

The run() Method

This method defines the actions to take place during the life of a thread. Here's an example:

```
// Change coordinates and repaint
   public void run() {
      while(ttapeThread != null) {
         try {Thread.sleep(50);} catch (InterruptedException e) {}
         setcoord();
         repaint();
      }
   }
```

In this example, the **run**() method makes the process sleep for 50 milliseconds while the instance of the class named **ttapeThread** is not equal to null. Then, the **setcoord**() method is called followed by the **repaint**() method.

The sleep() Method

This method releases control of a thread from the processor for a specified amount of time. The syntax for calling this method is:

```
Thread.sleep(Miliseconds);
```

In the **run**() method we just presented, the **sleep**() method is called to allow other threads to be processed while the **ttapeThread** is put on hold. This allows the browser to accept other input and not redraw the screen every instance of the thread.

The suspend() Method

This method suspends execution of a thread. As the following example shows, it requires no parameters:

```
// Handle mouse clicks
   public boolean handleEvent(Event evt) {
      if (evt.id == Event.MOUSE_DOWN) {
         if (suspended) {
            ttapeThread.resume();
         } else {
            ttapeThread.suspend();
         }
      suspended = !suspended;
      }
      return true;
   }
```

This line of code states that in the event of a mouse click and the thread is running, the thread will be suspended from operation. This allows other threads in the queue to be processed; but as soon as the thread is reactivated, it will resume its place in the queue as long as a higher priority thread is not executing.

The resume() Method

This method resumes a suspended thread. Here's an example of how it is used:

```
// Handle mouse clicks
   public boolean handleEvent(Event evt) {
      if (evt.id == Event.MOUSE_DOWN) {
         if (suspended) {
            ttapeThread.resume();
         } else {
            ttapeThread.suspend();
         }
      suspended = !suspended;
      }
      return true;
   }
```

The **resume**() method is responsible for reactivating a method that is asleep. This allows for the thread to re-enter the queue.

The yield() Method

This method causes the current thread to move to the end of the queue and lets the next thread take control of the processor. Here's an example of how it can be used:

```
// Change coordinates and repaint
   public void run() {
      while(ttapeThread != null) {
         setcoord();
         repaint();
         yield();
      }
   }
```

If the thread exists, the **setcoord**() method is executed followed by the **repaint**() method. Then the **yield**() method is called to permit the next thread in line to execute. Unfortunately, this is not wise if we are to depend on a scheduled **repaint**(). We could fall victim to the mercy of the threads that will be placed before the current thread that is moved to the end of the queue.

The stop() Method

This method ceases the life of a thread and performs the required cleanup operations:

```
// Stop thread then clean up before close
   public void stop(){
      if(ttapeThread != null)
         ttapeThread.stop();
      ttapeThread = null;
   }
```

In the event that the end of the process is reached, this method is called to clean up after the thread and perform any final procedures before closing out.

The destroy() Method

This method causes a thread to die without any cleanup operations being performed:

```
// Stop thread then clean up before close
   public void stop(){
      if(ttapeThread != null)
         ttapeThread.destroy();
      ttapeThread = null;
   }
```

In the event that the **stop()** method of the applet is called, the thread **ttapeThread** will be destroyed and no further lines of code for that object will be executed.

Multiple Objects Interacting with One Source

When you have multiple threads in your application running all at once, the need for limiting access to devices that write data and perform other critical tasks becomes absolutely necessary. After all, there is no way of telling when objects may try to update a storage device like a file at the same time. As a result, data may become corrupt or false information may be extracted from a device.

Synchronizing Revisited

If you recall from Chapter 5, we showed you how to declare a **synchronized** method. If you don't remember, here is the syntax.

```
synchronized ReturnType Identifier([ParameterList]) [Throws]
{
   MethodBody;
}
```

The **synchronized** modifier is used to declare a method of which only one object can execute at any one time. This is accomplished by the Java Virtual Machine setting up an object monitor for every portion of code that declares itself as synchronized. In order for the code to run, the object must attain this monitor. In the event that a thread wants to run a synchronized section of code, it is blocked until the thread ahead of it finishes executing the particular section of code. Let's look at an example of how the synchronized techniques works:

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

public class MyApp2 extends Frame implements Runnable {

    static TextArea t1;
    static TextArea t2;

    MyApp2() {
         // Calls the parent constructor Frame(string title)
         // Same as setTitle("Duck Duck Goose");
         super("Counting example");

        // A new panel to the south that 4 buttons and 1 choice
        t1 = new TextArea();
        t2 = new TextArea();
        add("East", t1);
        add("West", t2);

         pack();
         show();
        }

     public static void main(String args[]) {

        int i = 0;

        MyApp2 game = new MyApp2();
        Thread person1 = new Thread(game, "duck");
        Thread person2= new Thread(game, "goose");

        person1.start();
        person2.start();

        while ((person1.isAlive()) || (person2.isAlive())) {
            ++i;
            t2.setText(Integer.toString(i));
        }

        t2.appendText("\n Time through the loop \n\nYour It");
        person1.stop();
        person2.stop();

     }
     public synchronized void run() {
         int d = 0;
         int change = 0;

        while(d < 100) {
           t1.appendText("\n  " + Thread.currentThread().getName() +
             "  " + d );
           ++d;
         }
     }

     public boolean handleEvent(Event evt) {

       switch(evt.id) {
          case Event.WINDOW_DESTROY: {
              System.exit(0);
              return true;
          }
        default:
            return false;
        }
   }
}
```

The above code initiates two threads that cycle through the synchronized **run**() method. When you compile this program, you will see the first thread, **person1**, count up to 99, followed by the next thread, **person2**, count up to 99 and end. The thread **person2** must wait until the monitor is released by the previous thread before executing. While this process is occurring, notice that the counter timing the execution of the synchronized threading event is running alongside.

Wait() a Second... Notify() Me When...

Running a thread through a synchronized method is perfectly fine if you don't need any additional information from the outside. But let's suppose you wish to execute a thread within a synchronized method, and half way through you need to collect information from an additional thread. The first thread establishes a foundation, perhaps opening a file. Then, the following thread will enter the method and write the information and leave. Finally, the original thread will perform the cleanup necessary for closing the file. Well, this is all great, but remember we synchronized the method to permit only one thread to execute at any one time. This is easily remedied by using the **wait**() method, which causes the currently executing method to release the monitor to the next thread. The thread that released the monitor can then reacquire the monitor when the **notify**() method is called from within the same method. The thread waiting then picks up right from the point where it began waiting. Let's modify our previous example **MyApp2** to utilize the **wait**() and **notify**() methods:

```
public synchronized void run() {
    int d = 0;
    int change = 0;


   while(d < 100) {
       t1.appendText("\n   " + Thread.currentThread().getName() + "   " +
         Integer.toString(d) );
       ++d;
       if( d == 50) {
           try {
                if (Thread.currentThread().getName().equals("duck")) {
                   this.wait();
                }
           }
        catch(InterruptedException e) {
        }
     }
   }
   if (Thread.currentThread().getName().equals("goose")) {
       this.notify();
    }
}
```

After compiling the class again and running it, you will notice that the first thread counts to 50. The thread **person1** then releases the monitor to the next thread, **person2**. The thread then counts up to 99 and notifies the previous thread to begin executing from where it left off.

Grouping Your Threads

Once you have created your threads, they are like children who run loose in the mall. No matter how often you call them, they will go off into their own little world. Just like children, threads themselves can have parents. These parents are established by assigning them to a **ThreadGroup**. The **ThreadGroup** is actually a class just like the one the threads are derived from. You can create a **ThreadGroup** the same way you initialize any class in Java:

```
ThreadGroup parentAuthor = new ThreadGroup( " The Potts ");
```

This statement sets up a **ThreadGroup** named " The Potts " with a reference to the object **parentAuthor**. From here we can assign threads to this group by passing the name of the **ThreadGoup** with the initialization of the thread. For example, if we wish to add to threads to the **ThreadGroup parentAuthor**, we would enter the following:

```
Thread child1 = new Thread( parentAuthor, "Angela");
Thread child2 = new Thread( parentAuthor, "Anthony");
```

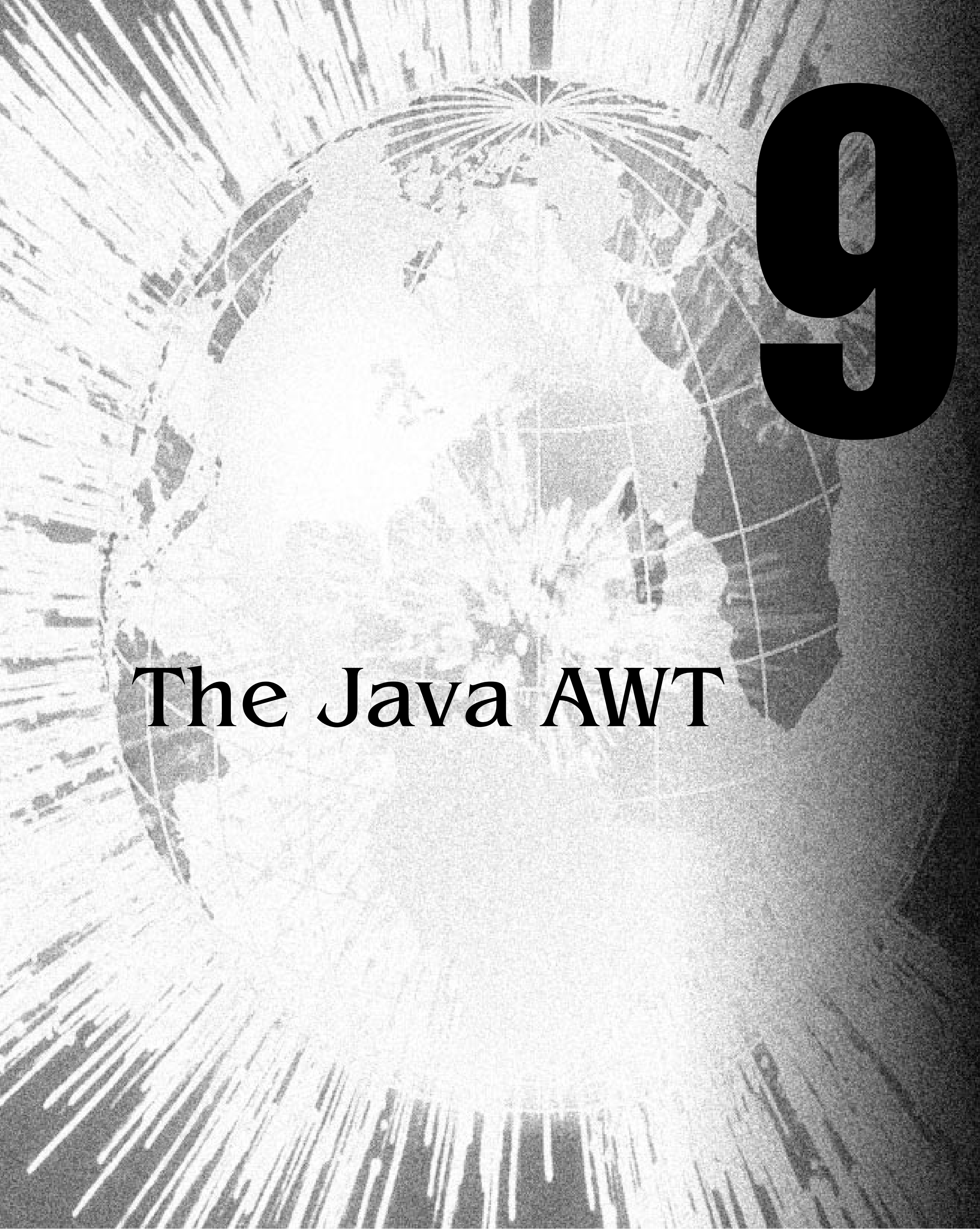
Creating hierarchies of these groups is just as easy as assigning threads to the group. For example, let's suppose that the **parentAuthor ThreadGroup** also wants to have a subgroup underneath it named **petsAuthor**. To accomplish this we would simply use the following code:

```
ThreadGroup petsAuthor = new ThreadGroup( parentAuthor, "Our Pets");
```

This allows for quick subgrouping of like threads. There are three main advantages to subgrouping threads:

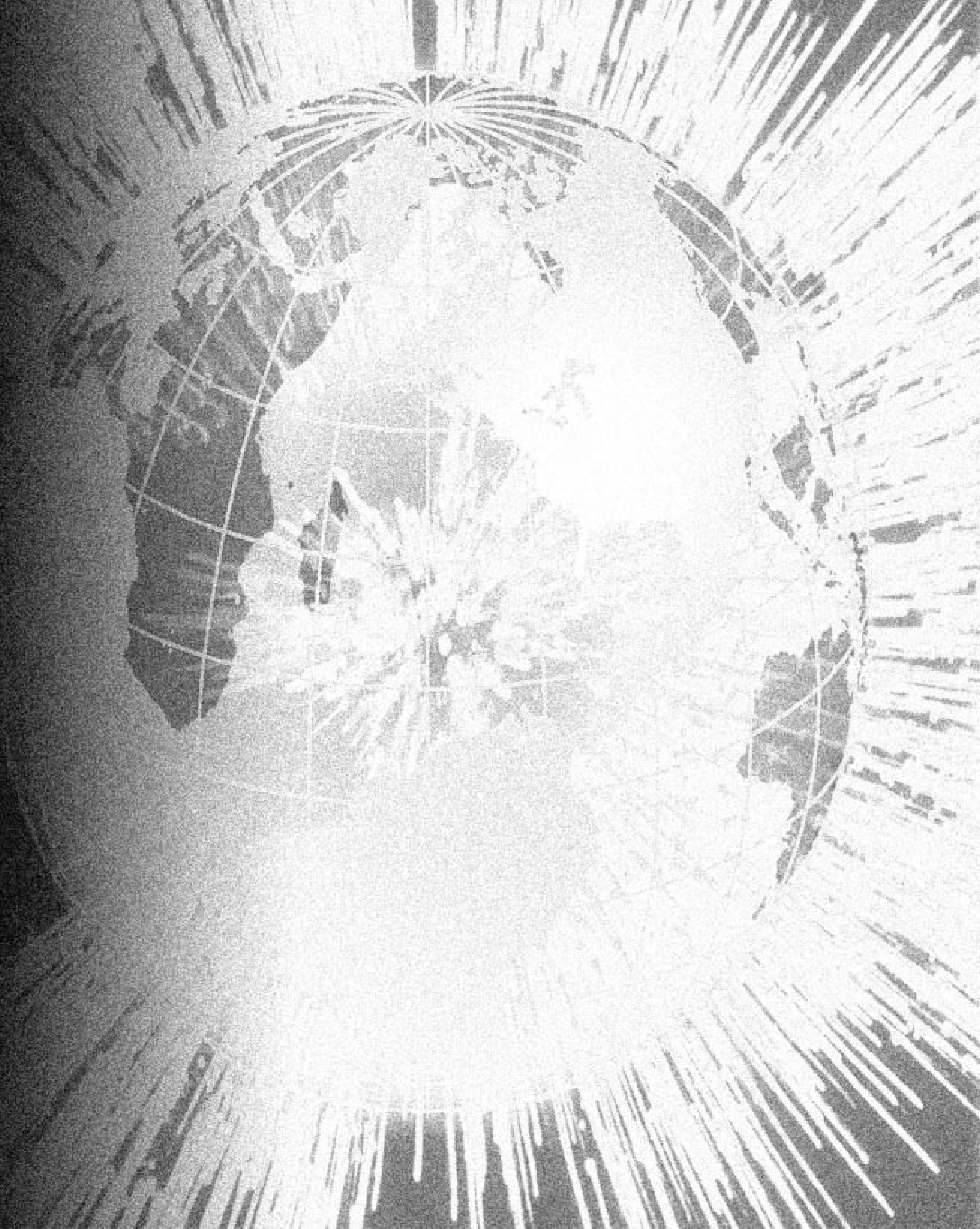
- Controlling the states of all the threads contained within the group without having to individually set each one.
- Retrieving all the threads in the group easily so that you can identify a thread quickly.
- Setting the priority of all the threads within the group with one command to the group.

Note: Setting the priority of the ThreadGroup only effects the threads of less priority than the calling method. If a thread is currently set at a high priority, it will continue at this level until it dies.



9

The Java AWT



9 The Java AWT

If you're wondering where to look for information on creating interface components for Java programs, you've come to the right place. The AWT provides a treasure chest of powerful interface classes.

No one would use a programming language these days if it did not have built-in support for common user interface objects like windows, menus, dialogs, and buttons. Fortunately, the designers of Java did not overlook this. They created a package called the *Abstract Window Toolkit* or *AWT,* which allows Java programmers to build GUIs very easily. Although AWT is very flexible and powerful, its shear size will overwhelm you until you understand how it is organized and the basics for using it.

To help you get more out of the AWT, we'll look at how the AWT is arranged. Then we'll present each of the key AWT classes. We'll show you how to use the layout manager, and we'll present the basics for creating interface components such as menus. If we tried to cover the AWT in a lot of detail, we could easily turn this chapter into an entire book. However, since Java is still a very young language and much of the AWT is still being solidified, we will only cover enough of this library to get you started with the AWT so that you can use it effectively. You'll want to keep your browser tuned to Sun's Java site for the latest information on this powerful package.

Introducing the AWT

When you first start to work with the AWT, you may notice that it lacks many of the features that you would find in other graphical user interface (GUI) libraries. This is because Java is a cross-platform language, and the tools provided with any Java library must be designed so that they can work with all systems.

As you build more complex applets and applications, you will find it difficult to *not* use the AWT because of its extreme flexibility. If a component such as a window or menu doesn't do what you need, you can simply subclass it and add your own custom features.

To use the AWT in a Java applet or program, you must first import the AWT package by using the **import** statement as shown here:

```
import java.awt.*;   // Include the AWT
```

The asterisk (*) is used with the **import** statement to tell the Java compiler to include *all* classes in the *immediate* subdirectory. Once you include this package, you can use any of the AWT controls or packages to derive your own. With just a little bit of programming effort, you'll be amazed at the types of interface components you can create for your Java applications—everything from scrollable windows to fully functional pop-up menus.

Here's an example of a simple Java program that displays a window that contains a text message:

```
import java.awt.*;   // Include the AWT

public class testWin extends Frame {   // Use the Frame class

   public testWin(){}   // Constructor

   public static void main(String args[]) {
      testWin Test = new testWin();
      // Display a line of text
      Test.setText("This text will be displayed in the window");
      // Add a second line
      Test.appendText(" Add this text to the next line in the window");
      Test.show();    // Display the frame
   }
}
```

Introducing the Layout Manager

Creating applications with a visual programming language like Visual Basic and Delphi can simply involve choosing from a selection of custom components written by other programmers and then dragging a component onto a form. Visual programmers like to refer to this practice as "drop-and-drag" programming. Unfortunately, Java programming is not quite this easy (although Java development is headed in this direction). When you place controls on forms to build applications with visual languages, you usually specify absolute positions. In other words, you tell your development environment exactly where you want to place a control—right down to the pixel. The problem with this approach is that different operating systems use different methods to display graphical components. So, a form that looks good on a PC may not look right on a Mac screen.

If you have only programmed for an environment like Windows, many of these problems are not as apparent because Windows takes care of specific system and interface related details for you. To provide this type of flexibility, Java provides a development tool called the layout manager, which works in conjunction with the AWT.

Java's layout manager helps you control where items will appear on the screen. The layout manager is actually an abstract class itself that you can use to create custom layouts. There are several custom layout methods that come with the Java Development Kit. As we present the AWT in this chapter, you'll learn more about the layout manager.

What About Menus?

You can't create a good user interface without implementing some form of menus. The Macintosh and Windows environments are filled with menus. Computer users expect your applications to present them with some sort of menuing system so that they can access the available features. When creating applets, menus are not as important and can be confusing because your Web browser already has its own menu system. But when you are creating Java applications, you'll more than likely want to add menus to your programs—and that's where the AWT comes in. The AWT can help you create menus very easily. However, it's up to you to make them functional and intuitive.

The AWT Hierarchy

The AWT package consists of many classes and subclasses. Most of the controls, like buttons and text fields, are derived from the **Component** class. Since all of the

AWT controls are descendants from the **Component** class, they all share some of the same key methods such as **createImage**(), **disable**(), and **hide**(). Figure 9.1 presents a tree that illustrates the class hierarchy for the controls of the AWT and Table 9.1 lists the complete set of classes.

If you have done any graphics or interface programming before, some of these class names should be somewhat familiar to you. Instead of reinventing the wheel, the developers of Java used traditional user interface components—windows, dialogs, scrollbars, and so on—to build up the AWT. Of course, you'll find that the AWT heavily embraces object-oriented programming techniques.

- Component
    - Button
    - Canvas
    - Checkbox
    - Choice
    - Container
        - Panel
        - Window
            - Dialog
            - Frame
    - Label
    - List
    - Scrollbar
    - TextComponent
        - TextArea
        - TextField

Figure 9.1
The class hierarchy of the AWT.

The Component Class

Since all of the controls and GUI elements that we'll be using are subclassed from the **Component** class, let's start with it. The **Component** class is rarely (if ever) used as is. It is almost always subclassed to create new objects that inherit all its functionality. **Component** has a tremendous number of methods built into it that all of the classes that subclass it share. You will be using these methods quite often to perform tasks such as making the control visible, enabling or disabling it, or resizing it.

Key Component Class Methods

The declarations and descriptions for the **Component** class would fill up a hundred pages. Fortunately, they are available online at Sun's Java site. You can download the API reference and load up the **Component** class to examine the methods

Table 9.1 The Complete List of Classes in AWT

BorderLayout	FlowLayout	MenuComponent
Button	Font	MenuItem
Canvas	FontMetrics	Panel
CardLayout	Frame	Point
Checkbox	Graphics	Polygon
CheckboxGroup	GridBagConstraints	Rectangle
CheckboxMenuItem	GridBagLayout	Scrollbar
Choice	GridLayout	TextArea
Color	Image	TextComponent
Component	Insets	TextField
Container	Label	Toolkit
Dialog	List	Window
Dimension	MediaTracker	
Event	Menu	
FileDialog	MenuBar	

available in this class. However, there are a few methods that are important, and you will probably use them for all the controls that subclass the **Component** class. We'll introduce these methods next, and we'll examine some of the key event handling methods in Chapter 10.

bounds()

This method returns the current rectangular bounds of the component.

disable()

This method disables a component.

enable([Boolean])

This method enables a component. You can pass zero arguments, or a Boolean argument to enable or disable a control. Here's a few examples of how this method can be called:

```
myComponent.enable();
myComponent.enable(x==1);
```

getFontMetrics()

This method gets the font metrics for the component. It returns null if the component is currently not on the screen.

getGraphics()

This method gets a graphics context for the component. This method returns null if the component is currently not on the screen. This method is an absolute necessity for working with graphics.

getParent()

This method gets the parent of the component.

handleEvent(Event evt)

This method handles all window events. It returns true if the event is handled and should not be passed to the parent of the component. The default event handler calls some helper methods to make life easier on the programmer. This method is used to handle messages from the operating system.

HIDE()

This method hides the component. It performs the opposite operation of**show**().

INSIDE(INT X, INT Y)

This method checks to see if a specified x,y location is “inside” the component. By default, x and y are inside a component if they fall within the bounding box of that component.

ISENABLED()

This method checks to see if the component is enabled. Components are initially enabled.

ISSHOWING()

This method checks to see if the component is showing on screen. This means that the component must be visible, and it must be in a container that is visible and showing.

ISVISIBLE()

This method checks to see if the component is visible. Components are initially visible (with the exception of top level components such as Frame).

LOCATE(INT X, INT Y)

This method returns the component or subcomponent that contains the x,y location. It is very useful for checking for mouse movement or mouse clicks.

LOCATION()

This method returns the current location of the component. The location will be specified in the parent’s coordinate space.

MOVE(INT X, INT Y)

This method moves the component to a new location. The x and y coordinates are in the parent’s coordinate space.

REPAINT([TIME])

This method repaints the component. This will result in a call to the **update**() method as soon as possible. You can specify a time argument so that Java knows that you want the component repainted within a specified number of milliseconds. You can also update just a certain portion of a control by sending the x and y coordinates that specify where you want to start the update and a width and height that specify how much to update. Here are some examples:

```
// Regular update
myComponent.update();

// Update within 250 milliseconds
myComponent.update(250);

// Update rectangle
myComponent.update(50, 50, 200, 200);

// Update same rectangle within 250 milliseconds
myComponent.update(250, 50, 50, 200, 200);
```

resize(int Width, int Height)

This method resizes the component to the specified width and height. You could also pass it a dimension object instead of the two integers. Here are a few examples:

```
myComponent.resize(300, 200);

myComponent.show(new dim(300, 200));
```

setFont(Font)

This method sets the font of the component. The argument passed is a **Font** object. For example, this method call

```
myComponent.setFont(new Font("Helvetica", Font.PLAIN, 12);
```

would change the font of **myComponent** to a Helvetica type face with no bolding or italics or underline and a point size of 12.

show([Boolean])

This method "shows" the component. By calling this method you make a control visible or not. It can also be passed a conditional statement. Here are a few examples:

```
myComponent.show();
myComponent.show(x==1);
```

SIZE()

This method returns the current size of the component. The size is returned in dimensions of width and height.

The Frame Class

The **Frame** class is used to create standard application windows. It is a direct descendant of the **Window** class and is used to create windows that automatically include the menu bar, title bar, control elements, and so on. **Frame** looks like a standard window that you would expect to see running under most operating systems like Windows or Macintosh OS.

You will probably use the **Frame** class to create window interface components for the majority of your applications. However, when you need to create custom windows for special situations, you may need to use the **Window** class instead.

Hierarchy for Frame

```
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Window
        java.awt.Frame
```

Declaration for Frame

To create a frame, you need to use a constructor to initialize your class. The constructor method does not even need to have any code in it; the method just needs to be provided so that you can instantiate your object. The following listing shows a complete Java application that uses a frame:

```
import java.awt.*;   // Include the AWT

public class testWin extends Frame {   // Use the Frame class

   public testWin(){}   // Constructor

   public static void main(String args[]) {
      testWin Test = new testWin();
      Test.show();    // Display a window
   }
}
```

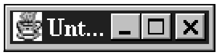


Figure 9.2
A simple windowed Java application.

In the **main**() method we use our class's constructor to create our object named **Test**. Then we call the object's **show**() method to make the window frame visible (frames are invisible by default). If you were running this Java program in Windows 95, you'd see a window that looks like the one shown in Figure 9.2.

As you can see, the window is quite simple. You need to use several of the **Frame** class's methods to make the frame useful. The other thing you may notice is that when you try and close the window and terminate the program, nothing happens! That's because you have not told Java to do it. You need to add an event handling method to catch windows messages. In this case, we are looking for the **WINDOW_DESTROY** call from the operating system. Here is the extended code that sets the sizes of the frame, gives it a title, and catches messages:

```
import java.awt.*;

public class winTest1 extends Frame {

   public winTest1() {}

   public synchronized boolean handleEvent(Event e) {
      if (e.id == Event.WINDOW_DESTROY) {    // Has window been destroyed?
         System.exit(0);
         return true;
      }
      return super.handleEvent(e);
   }

   public static void main(String args[]) {
      winTest Test = new winTest();
      Test.setTitle("Test Window");
      Test.resize(300 ,200);
      Test.show();
   }
}
```

Figure 9.3 shows what the new application looks like.

Figure 9.3
A fully functioning application using a frame.

We are gong to discuss event handling in more detail in the next chapter, so don't get worried if you do not understand that part of the above code.

What you should notice is the two new calls to two of the frames methods: **setTitle**() and **resize**(). These methods perform the function they are named after; they set the title of our application and resize it respectively.

Let's look at the methods that are specific to the **Frame** class.

Methods for the Frame Class

dispose()

This method removes the frame. It must be called to release the resources that are used to create the frame. This method should be called when you exit a window in an applet. In an application, you would usually use the **System.exit**() method to terminate the program and release all memory used by the application. This method overrides the **dispose**() method from the **Window** class.

getIconImage()

This method returns the icon image for the frame.

getMenuBar()

This method gets the menu bar for the frame.

getTitle()

This method gets the title of the frame.

isResizable()

This method returns true if the user can resize the frame.

remove(MenuComponent m)

This method removes the specified menu bar from the frame.

setCursor(Image img)

This method sets the current cursor image that will be used while the cursor is within the frame.

setIconImage(Image img)

This method sets the image to display when the frame is iconized. Note that not all platforms support the concept of iconizing a window. The icon will also be displayed in the title window in Windows 95.

setMenuBar(MenuBar mb)

This method sets the menu bar for the frame to the specified **MenuBar**.

setResizable(Boolean bool)

This method sets the resizable flag.

setTitle(String title)

This method sets the title for the frame to the specified **title**.

The Panel Class

Panels are probably one of the most useful components that no one will ever see. Most of the time, panels are just used as storage containers for other components. You could also use a panel for drawing or to hold images, but panels are not usually used for these tasks.

Panels are important because they offer you a way to gain very strict control over the layout of the other interface controls in your program. Unless you have a very simple application, you will need multiple controls on the screen at once. Usually these controls will not all fit where you want them using one layout class or another. Panels provide a mechanism so that you can mix and match.

Let's look at an example. Assume you have a frame that you want to fill with a text field in the upper part of the frame, and three buttons lined up along the bottom. If you only used a single layout class for the entire form, you would not have enough control to do this. Figure 9.4 illustrates what we want the frame to look like. Figure 9.5 and 9.6 shows the best you can achieve using a single layout class. This is not bad, but if you resize the frame the layout gets pretty ugly.

What we need to be able to do is use one type of layout class for the upper part of the frame, and another for the bottom. We can do this by using a pair of panels, one for the top using a border style layout and another panel for the bottom using a flow style layout. Now, when we add our controls, we add them to their respective panels instead of the frame, and everything is taken care of for

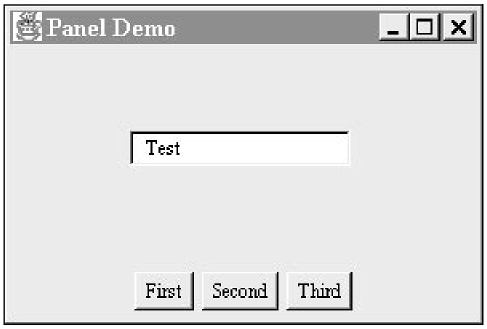


Figure 9.4
Creating a window using the Panel class.

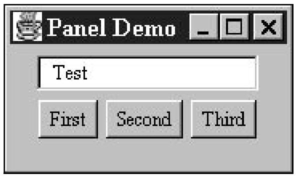


Figure 9.5
A close approximation with a single layout class.

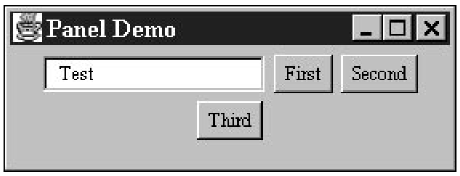


Figure 9.6
Resizing the Panel.

us. The user can resize the control all they want and our controls will stay where we placed them originally. Here is the code that performs this new configuration. The output is shown in Figure 9.7.

```
import java.awt.*;

public class mixLayout extends Frame {

   public mixLayout() {
      super("Mixed Layout Demo");
      setLayout(new BorderLayout());
      Panel top = new Panel();
      Panel bottom = new Panel();
      top.setLayout(new BorderLayout());
      top.add("Center", new TextArea("HelloWorld", 15, 5));
      bottom.setLayout(new FlowLayout());
      bottom.add(new Button("Load"));
      bottom.add(new Button("Save"));
      bottom.add(new Button("Quit"));
      add("Center", top);
      add("South", bottom);
      resize(300, 200);
      show();
   }
   public static void main(String args[]) {
      mixLayout test = new mixLayout();
   }

}
```

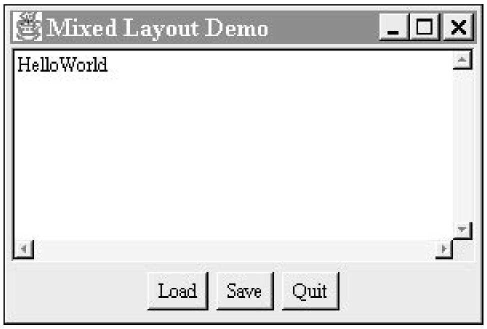
Hierarchy for Panel

```
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Panel
```

Declaration for Panel

Panels have a very straight-forward declaration because they cannot accept any arguments whatsoever.

```
Panel myPanel = new Panel();
```



HelloWorld

Figure 9.7
The new two-panel program with different layout methods.

Methods for Panel

Panels have very few new methods ouside of the ones inherited from the **Container** class. The key new method that is introduced is **setlayout**().

SETLAYOUT(LAYOUTMANAGER)

As you have already seen, the **setlayout**() method is used to define which layout manager will be used to place controls on the panel. If you do not set a layout manager, the panel control defaults to **flowLayout**().

The Label Class

The **Label** class defines a very simple control that is simply used to display a text string. This text string is usually used to indicate what tasks are controlled by another user interface control. For example, if you had a group of radio buttons, a label control might be used to tell the user what the group is about.

Hierarchy for Label

```
java.lang.Object
  java.awt.Component
    java.awt.Label
```

Declaration for Label

The most common declaration for a label is to assign the text string as the control is being declared. Here's an example:

```
new Label("Fruits of the world:");
```

We can also assign the text string to an object variable like this:

```
Label fruitLabel = new Label("Fruits of the world:");
```

Table 9.2 shows the three ways you can declare a **Label** class.

Methods for Label

Labels have very little functionality of their own—they can't even process mouse clicks. They are not intended to do much more than sit there and display their string. There are, however, a few methods you should know about.

getAlignment()

This method returns the current alignment of the label.

getText()

This method does what it sounds like—it returns the text the label is displaying.

setAlignment(int)

This method changes the alignment of the label. The argument is the same as the one used with **Label.LEFT**, **Label. CENTER**, and **Label.RIGHT** (see Table 9.2 for more information).

Figure 9.8 shows a few labels with different alignments. Here is the code used to produce them:

```
add(new Label("Left")); // no need to specify alignment because it
                        // defaults to left
add(new Label("Center", Label.CENTER));
add(new Label("Right", Label.RIGHT));
```

Table 9.2 Options for Declaring a Label Class

Declaration	Description
Label()	This constructor will create an empty label.
Label(String)	This constructor will create a label with the given string.
Label(String, int)	This constructor will create a label with the given string as well as define the alignment of the string. The **int** part of the declaration is represented by setting it to **Label.LEFT**, **Label.CENTER**, or **Label.RIGHT**. (These labels should be pretty self-explanatory.) Remember though, that if you do not assign an alignment, the label control will default to left-justified.

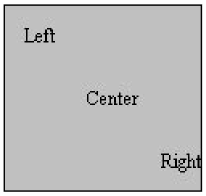


Figure 9.8
The Label component.

setText(String)

This method sets or changes the displayed text.

Button Class

The **Button** class is one of the most often used classes, for obvious reasons. How often do you see Windows or Mac programs without some sort of button somewhere?

Buttons are UI components that trigger actions when they are pressed. They have multiple states to provide feedback to the user. The neat thing about buttons (and all the UI elements for that matter) is that a button you create for a Java program will change its appearance depending on the operating system. That's one of the benefits of cross-platform. A button on your application or applet will look like a standard Windows button on a PC. Or, it will look like a standard Mac button on a Mac. The disadvantage here is that if you create other elements that are dependent on the size and/or shape of the graphics for your button, then you will run into trouble.

Figure 9.9 illustrates a few different buttons. Notice that the size of the buttons depends on the length of the caption.

Hierarchy for Button

```
java.lang.Object
  java.awt.Component
    java.awt.Button
```

Declaration for Button

Buttons have two different constructor options. The first one simply creates a blank button. The second option creates a button that you assign a string to display.

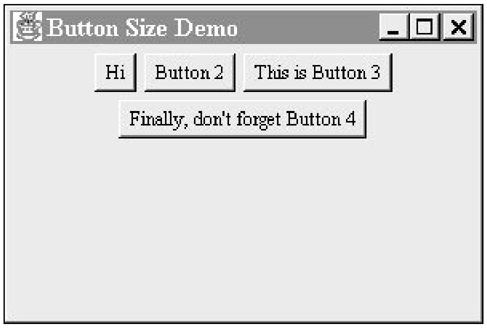


Figure 9.9
A few Button components.

```
new Button(); // Empty button
new Button(String); // Button with "String" as the caption
```

Methods for Button

There are only two methods specific to the button component.

GETLABEL()

This method returns the current caption of the button.

SETLABEL()

This method changes the caption of the button.

The real power behind buttons is realized when you handle the events a button triggers. This is usually handled in the **handleEvent**() method that we will be showing you how to use in the next chapter.

The Canvas Class

The **Canvas** class is a very simple component. It is designed to be subclassed and used as a container for graphics methods, much like an artist's canvas.

Hierarchy for Canvas

```
java.lang.Object
  java.awt.Component
    java.awt.Canvas
```

Declaration for Canvas

The **Canvas** component has only a single constructor:

```
new Canvas();
```

Methods for Canvas

The only method that is used with the **Canvas** component is the **paint**() method which is called to update the contents. If you do not override this method, the **Canvas** component is pretty much worthless because it has very little functionality of its own.

When you subclass a **Canvas** component, you can really have some fun. Override the **paint**() method and add in calls to painting and geometric methods. Or, you can create a Canvas object and make up your own calls to it that react to button clicks, mouse movements, and so on.

The Checkbox Class

The **Checkbox** class is actually two controls in one. Along with checkboxes you also can implement the functionality of radio buttons by grouping multiple checkbox controls. Checkboxes are great for giving yes/no options. Radio buttons are good for multiple choice questions.

Figure 9.10 shows a couple of individual checkboxes on the left and a single group of checkboxes on the right. Here is the code that produced them:

```
import java.awt.*;

public class testMe extends Frame {

   public testMe() {
      super("Checkbox Demo");
      Panel P1 = new Panel();
      Panel P2 = new Panel();
      CheckboxGroup G1 = new CheckboxGroup();
      setLayout(new GridLayout(1,2));
      add(P1);
      add(P2);
      P1.setLayout(new FlowLayout());
      P1.add(new Checkbox("E-Mail Tom"));
      P1.add(new Checkbox("E-Mail Jack"));
      P2.setLayout(new FlowLayout());
      P2.add(new Checkbox("Me", G1, true));
      P2.add(new Checkbox("You", G1, false));
      P2.add(new Checkbox("Them", G1, false));
      resize(300, 200);
      show();
   }

   public static void main(String args[]) {
      testMe test = new testMe();
   }
}
```

Hierarchy for Checkbox

```
java.lang.Object
  java.awt.Component
    java.awt.Checkbox
```

Declaration for Checkbox

A number of options are available for calling a **Checkbox** constructor. Here is each option with a summary description:

```
new Checkbox();
```

Constructs a checkbox with no label, no checkbox group, and initializes it to a false state.

```
new Checkbox(String);
```

Constructs a checkbox with the specified label, no checkbox group, and initializes it to a false state.

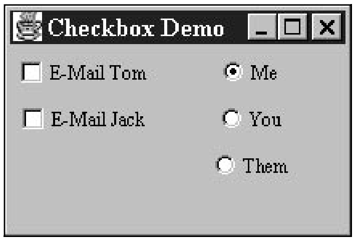


Figure 9.10

A few iterations of the Checkbox component.

```
new Checkbox(String, boolean);
```

Constructs a checkbox with the specified label, no checkbox group, and initializes it to a specified boolean state.

```
new Checkbox(String, CheckboxGroup, boolean);
```

Constructs a checkbox with the specified label, specified checkbox group, and initializes it to a specified boolean state.

Methods for Checkbox

Although checkboxes are simple UI components, they offer us a lot of options and therefore provide a few custom methods we can use to control them.

getCheckboxGroup()

This method returns the checkbox group that this checkbox belongs to.

getLabel()

This method gets the label of the button.

getState()

This method returns the boolean state of the checkbox.

setCheckboxGroup(CheckboxGroup)

This method sets the **CheckboxGroup** of the check box.

setLabel(String)

This method changes the label of the checkbox.

setState(boolean)

This method sets the checkbox to the specified boolean state.

The Choice Class

The **Choice** class implements a pop-up menu that shows the currently chosen option normally; but when it has the focus, it opens up and displays all the options a user can select from. Figure 9.11 shows a **Choice** component at rest and Figure 9.12 shows the same component in action.

Here is the code that creates the **Choice** component shown in Figures 9.11 and 9.12:

```
import java.awt.*;

public class testMe extends Frame {

   public testMe() {
      super("Choice Demo");
      Choice C1 = new Choice();
      setLayout(new FlowLayout());
      add(C1);
      C1.addItem("You");
      C1.addItem("Me");
      C1.addItem("Them");
      C1.addItem("Us");
      C1.addItem("Everyone");
      resize(300, 200);
      show();
   }

   public static void main(String args[]) {
      testMe test = new testMe();
   }
}
```

Hierarchy for Choice

```
java.lang.Object
  java.awt.Component
    java.awt.Choice
```

Declaration for Choice

When you are creating a **Choice** component, you cannot set any options. All the options must be set after the object is created. The call for initializing **Choice** is simply:

```
new Choice();
```

Methods for Choice

The set of special methods available for the **Choice** class include:

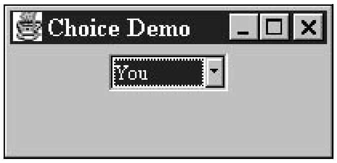


Figure 9.11
The Choice component without the focus.

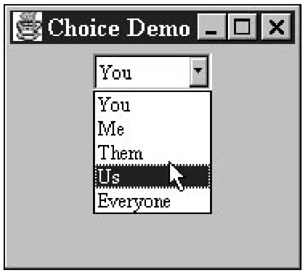


Figure 9.12
The Choice component with the focus.

addItem(String)
This method adds an item to the list of choices.

countItems()
This method returns the number of items.

getItem(int)
This method returns the item at the specified index.

getSelectedIndex()
This method returns the index of the currently selected item.

getSelectedItem()
This method returns a string representation of the current choice.

select(int)
This method selects the item with the specified index position.

SELECT(STRING)

This method selects the item with the specified **String** if present.

The List Class

The **List** class is actually very similar to **Choice** except that it allows you to display multiple items and scroll through the ones not displayed. It also gives you the ability to select multiple items.

Figure 9.13 shows a form that contains a simple list with multiple elements. Here is the code that created these lists:

```
import java.awt.*;

public class testMe extends Frame {

   public testMe() {
      super("List Demo");
      List L1 = new List();
      setLayout(new FlowLayout());
      add(L1);
      L1.addItem("You");
      L1.addItem("Me");
      L1.addItem("Them");
      L1.addItem("Us");
      L1.addItem("Everyone");
      resize(300, 200);
      show();
   }

   public static void main(String args[]) {
      testMe test = new testMe();
   }
}
```

Hierarchy for List

```
java.lang.Object
  java.awt.Component
    java.awt.List
```

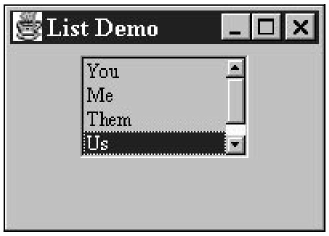


Figure 9.13
Create lists with the List class.

Declaration for List

Here are the two constructors you can call to initialize a **List** class:

```
new List();
```

Creates a scrolling list initialized with no visible lines and multiple selections disabled.

```
new List(int, boolean);
```

Creates a new scrolling list initialized with the specified number of visible lines and a boolean stating if multiple selections are allowed or not.

Methods for List

You'll find a number of methods for this class because of all the different options it offers.

addItem(String)

This method adds the specified item to the end of list.

allowsMultipleSelections()

This method returns true if the list allows multiple selections.

clear()

This method clears the list.

countItems()

This method returns the number of items in the list.

delItem(int)

This method deletes an item from the list at the specified index.

delItems(int, int)

This method deletes multiple items from the list. Items are deleted from the index position specified by the first parameter to the index position specified by the second parameter.

deselect(int)

This method deselects the item at the specified index.

getItem(int)

This method gets the item associated with the specified index.

getRows()

This method returns the number of visible lines in the list.

getSelectedIndex()

This method returns the selected item in the list or -1 if no item is selected.

getSelectedIndexes()

This method returns the selected indexes in the list in the form of an array.

getSelectedItem()

This method returns the selected item in the list or null if no item is selected, or it returns the first selected item if multiple items are selected.

getSelectedItems()

This method returns the selected items in the list into an array of strings.

getVisibleIndex()

This method gets the index of the item that was last made visible by the method **makeVisible**().

isSelected(int)

This method returns true if the item at the specified index has been selected; false otherwise.

makeVisible(int)

This method forces the item at the specified index to be visible. The method automatically scrolls the list to display the specified index.

minimumSize()

This method returns the minimum dimensions needed for the list.

minimumSize(int)

This method returns the minimum dimensions needed for the number of rows in the list.

preferredSize()

This method returns the preferred dimensions needed for the list.

preferredSize(int)

This method returns the preferred dimensions needed for the list with the specified amount of rows.

select(int)

This method selects the item at the specified index.

setMultipleSelections(boolean)

This method sets up a list to allow for multiple selections or not.

TextField and TextArea Classes

The **TextField** and **TextArea** classes implement other controls that you will use often. They are often accompanied with **Label** controls that tell the user what a text entry box is used for. Text fields can only have a single line of text, while text areas can have multiple lines like a word processor. One of the nice things about the **TextField** and **TextArea** components is that they interact with both the keyboard and the mouse. You can enter text by typing it on the keyboard. Then, you can use the mouse to place the cursor within that text. If you are a Windows

95 user, the components even support the right mouse button. If you right-click within a text field or text area, a pop-up menu will be displayed like the one shown in Figure 9.14.

Hierarchy for TextField and TextArea

```
java.lang.Object
  java.awt.Component
    java.awt.TextComponent
      java.awt.TextField
      java.awt.TextArea
```

Declaration for TextField and TextArea

A number of constructors are available for initializing **TextField** and **TextArea** classes:

```
new TextField();
```

Constructs a new **TextField**.

```
new TextField(int);
```

Creates a new **TextField** initialized with the specified number of columns (1 column = 1 character).

```
TextField(String)
```

Creates a new **TextField** initialized with the specified text.

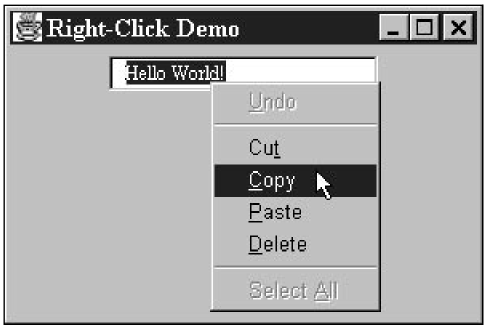


Figure 9.14
Right-clicking on a TextField component to display a pop-up menu.

```
new TextField(String, int);
```

Creates a new **TextField** initialized with the specified text and number of columns.

```
new TextArea();
```

Creates a new **TextArea**.

```
new TextArea(int, int);
```

Creates a new **TextArea** with the specified number of rows and columns.

```
new TextArea(String);
```

Constructs a new **TextArea** with the specified text displayed.

```
new TextArea(String, int, int);
```

Creates a new **TextArea** with the specified text and the specified number of rows and columns.

Methods for TextField and TextArea

Many of the methods used with these two classes are actually methods of the **TextComponent** class which both of these classes subclass. Let's look at these methods first.

getSelectionEnd()

This method returns the selected text's end position.

getSelectedText()

This method returns the selected text contained in the text component.

getSelectionStart()

This method returns the start position of the selected text. If no text is selected, the method returns -1.

getText()

This method returns the text contained in the text component.

isEditable()

This method returns a boolean value that tells us if the text component is editable or not. Text components are editable by default.

select(int, int)

This method selects the text found between the specified start and end locations.

selectAll()

This method causes all of the text in the text component to be selected.

setEditable(boolean)

This method sets whether or not this text component can be edited.

setText(String)

This method changes the text of the text component to the specified text.

Now, let's look at a few methods that are specific to the **TextField** component:

echoCharIsSet()

This method returns true if the **TextField** has a character set for echoing. Echoing is used for situations where you do not want the text people are typing to be displayed.

getColumns()

This method returns the number of columns in the **TextField**.

getEchoChar()

This method returns the character to be used for echoing. The character is not returned in a **String** format, just a simple **char**.

minimumSize()

This method returns the minimum size dimensions needed for the **TextField** in columns.

minimumSize(int)

This method is used to request a minimum size for the text box. The parameter specifies the number of columns for the text box. The method returns the minimum size dimensions needed for the **TextField** with the specified amount of columns.

preferredSize()

This method returns the preferred size dimensions needed for the **TextField** class.

preferredSize(int)

This method returns the preferred size dimensions needed for the **TextField** with the specified amount of columns being passed to it.

setEchoCharacter(char)

This method sets the echo character for the **TextField**. Most often you'll want to set this character to the asterisk symbol, especially if you are working with password boxes.

Now, we need to look at the methods specific to the **TextArea** class:

getColumns()

This method returns the number of columns in the **TextArea**.

getRows()

This method returns the number of rows in the **TextArea**.

insertText(String, int)

This method inserts the specified text at the specified position. The position tells Java the number of characters it needs to move over before it inserts the string.

preferredSize()

This method returns the preferred size dimensions of the **TextArea**.

preferredSize(int, int)

This method returns the row and column dimensions of the **TextArea**.

minimumSize()

This method returns the minimum size dimensions of the **TextArea**.

minimumSize(int, int)

This method returns the specified minimum size dimensions of the **TextArea**.

REPLACETEXT(STRING, INT, INT)

This method replaces text from the indicated start to end positions with the specified new text.

The Scrollbar Class

Several of the Java components we've been discussing in this chapter automatically use scrollbars when needed. For example, the **TextArea** control we just described will automatically create scrollbars when the text runs off the edge or the bottom. Scrollbars can also be created as standalone components. Scrollbars are useful when you need a user to pick a value. They offer a graphical interface rather than a simple text box where a user would usually have to type in a value.

Scrollbars are implemented using the concepts of a minimum, maximum, and current value. A scrollbar can be moved in three ways: small increments, large increments, or to an absolute position. The small increment occurs when the user clicks on the arrows at the top or bottom of the scrollbar. The large changes occur when the user clicks on the space between the current position and either arrow. Finally, the absolute change occurs when the user drags the current location indicator (sometimes called an *elevator*) and drags it to a specific spot within the scrollbar. Figure 9.15 shows a standard scrollbar and labels each part.

Let's consider a very typical use of a scrollbar. In this case, we have two components, a **Textfield** control and a **Scrollbar** control. We want to display the current value of the **Scrollbar** in the text box. Also, when the user changes the value in the text box, the scrollbar should match the change. Figure 9.16 shows what this simple application looks like.

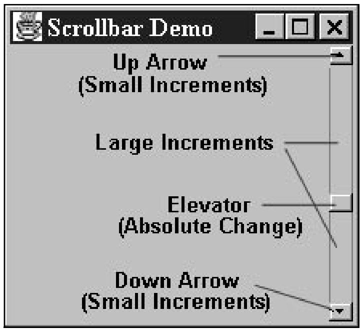

Scrollbar Demo
Up Arrow (Small Increments)
Large Increments
Elevator (Absolute Change)
Down Arrow (Small Increments)

Figure 9.15

A typical Java scrollbar.

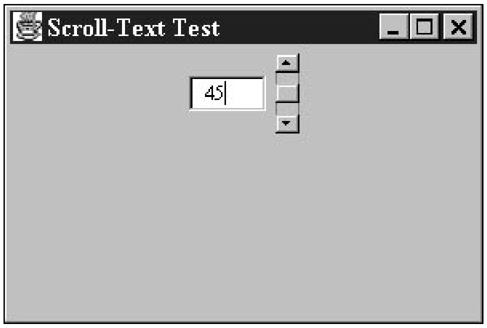


Figure 9.16
A Scrollbar control and a Textfield control with linked values.

And, here is the code for the entire application. Type it in and give it a try:

```
import java.awt.*;

public class ScrollTest extends Frame {
   TextField text;
   Scrollbar scroll;
   public ScrollTest() {
      super("Scroll-Text Test");
      text = new TextField(5);
      scroll = new Scrollbar(Scrollbar.VERTICAL, 0, 10, 0, 100);
      setLayout(new FlowLayout());
      add(text);
      add(scroll);
      resize(300, 200);
      show();
   }
   public boolean handleEvent(Event evt) {
      if (evt.target.equals(scroll)) {
         text.setText("" + scroll.getValue());
         } else if (evt.target.equals(text))
            {scroll.setValue(Integer.parseInt(text.getText()));
         } else if (evt.id == Event.WINDOW_DESTROY) {
         System.exit(0);
         return true;
      }
      return super.handleEvent(evt);
   }
   public static void main(String args[]) {
      ScrollTest test = new ScrollTest();
   }
}
```

Keep in mind that the scrollbar, like all the other controls, will change its appearance to match the operating system. Obviously, you would not want Windows 95 style scrollbars being displayed on a Macintosh. That would sure cause a commotion!

Hierarchy for Scrollbar

```
java.lang.Object
  java.awt.Component
    java.awt.Scrollbar
```

Declaration for Scrollbar

The three constructors provided for initializing scrollbars include:

```
new Scrollbar();
```

Constructs a new vertical **Scrollbar**.

```
new Scrollbar(int);
```

Constructs a new **Scrollbar** with the specified orientation. You can specify **Scrollbar.**HORIZONTAL or **Scrollbar.**VERTICAL. Scrollbars are vertical by default.

```
new Scrollbar(int, int, int, int, int);
```

Creates a new **Scrollbar** with the specified orientation, current value, large change size, minimum value, and maximum value.

Methods for Scrollbar

Here are the set of specialized methods for the **Scrollbar** class:

getMaximum()

This method returns an integer representing the maximum value of the scrollbar.

getMinimum()

This method returns an integer representing the minimum value of the scrollbar.

getOrientation()

This method returns an integer that gives us the orientation for the scrollbar. You can check the returned integer against **Scrollbar.**HORIZONTAL and **Scrollbar.**VERTICAL to determine the scrollbar's orientation.

getValue()

This method returns an integer representing the current value of the scrollbar.

getVisible()

This method returns the visible amount of the scrollbar.

setValue(int)

This method sets the value of the scrollbar to the specified value. If you try to set the current value to a number that is greater than the maximum or less than the minimum, the number becomes the new maximum or minimum, respectively.

setValues(int, int, int, int)

This method changes the values for the scrollbar. The arguments in order of appearance include:

- value - the position in the current window
- large change - the value that will be moved each time the user clicks in the area between the elevator and the arrows. This value is also called the *amount visible per page.*
- minimum - the minimum value of the scrollbar
- maximum - the maximum value of the scrollbar

Building Menus

Menus in Java are as easy to build and manage as the other visual components. Every part of a menu, from the menu bar to individual items is represented as a separate class, each having specialized properties and methods. Let's start our investigation of menus with the main component—the menu bar. Then, we'll work our way down into the other more specialized components, such as a menu itself and menu items.

The MenuBar Class

Each window or frame you create in a Java application or applet can have its own menu bar. The standard menu bar is always displayed along the top of the window. You are not allowed to have multiple menu bars within a single window unless you create the entire system yourself (but why would you want to!).

The **MenuBar** class itself provides little functionality. By itself it looks like a blank bar across the top of your window. To give your menuing system functionality, you need to add **Menu** and **MenuItem** components, which we will discuss later.

Hierarchy for MenuBar

```
java.lang.Object
  java.awt.MenuComponent
    java.awt.MenuBar
```

Declaration for MenuBar

Only a single constructor is provided for initializing a **MenuBar** class. This constructor creates a menu object but does not display it. You must first assign the **MenuBar** to a frame for it to be displayed:

```
new MenuBar();
```

Methods for MenuBar

ADD(MENU)

This method adds the specified menu to the menu bar.

COUNTMENUS()

This menu returns an integer representing the number of menus on the menu bar.

GETHELPMENU()

This method returns the name of the menu component on the current menu bar that has been designated as the “Help” menu. Help menus are discussed in the next section in more detail.

GETMENU(INT)

This menu gets the specified menu. Input is an integer representing the index position of a menu and it returns a **Menu** object.

REMOVE(INT)

This method removes the menu located at the specified index from the menu bar.

REMOVE(MENU)

This method removes the specified menu from the menu bar.

SETHELPMENU(MENU)

This method sets the current help menu to the specified menu on the menu bar.

The Menu Class

The **Menu** class is used to implement the selections for the section headings for each type of menu. Typical **Menu** components will be labeled "File," "Options," and "Help." Figure 9.17 shows a few menus.

Menu classes are always children of a single **MenuBar** class. They usually have **MenuItem** classes as children. However, a **Menu** class does not *need* to have menu items under it; it can react to events on its own. In Figure 9.17, you could click on the **Menu** component labeled "File" to expose its **MenuItem** children. However, if you were to click on the "Help" **Menu** item, it would not display any child **MenuItem** components because it does not have any. It acts on its own.

Hierarchy for Menu

```
java.lang.Object
  java.awt.MenuComponent
    java.awt.MenuItem
      java.awt.Menu
```

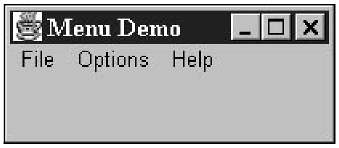


Figure 9.17

Examples of menus created with the Menu class.

Declaration for Menu

Here are the two constructors provided for the **Menu** class:

```
new Menu(String);
```

Constructs a new **Menu** with the specified string as the label. This menu will not be able to be "torn off." Tear-off menus are menus that will still appear on the screen after the mouse button has been released.

```
new Menu(String, boolean);
```

Constructs a new **Menu** with the specified label. The menu will be able to be torn off if the boolean value is set to true.

Methods for Menu

The specialized methods for the **Menu** class include:

add(MenuItem)

This method adds the specified item to the menu.

add(String)

This method adds an item with the specified label to the menu.

addSeparator()

This method adds a separator line to the menu at the current position.

countItems()

This method returns the number of elements in the menu as an integer.

getItem(int)

This method returns the menu item located at the specified index of the menu.

isTearOff()

This method returns true if the menu is a tear-off menu.

remove(int)

This method deletes the item at the specified index from the menu.

remove(MenuItem)

This method deletes the specified item from the menu.

The MenuItem Class

The **MenuItem** class is the last in the line of children of the three menu classes. We should mention, however, that **Menu** classes can have other **Menu** classes as children. These child **Menu** components look like menu items, but another menu would be under each menu to display more menu items. This is technique is used to create cascading menus as shown in Figure 9.18.

Here is the Java code that implements the menu system shown in Figure 9.18:

```
import java.awt.*;

public class testMe extends Frame {
   public testMe() {
      super("Menu Demo");
      MenuBar MB = new MenuBar();
      Menu M1 = new Menu("File");
      Menu M2 = new Menu("Options");
      Menu M3 = new Menu("More");
      MB.add(M1);
      MB.add(M2);
      MB.add(new Menu("Help"));
      M1.add(new MenuItem("Open"));
      M1.add(new MenuItem("Close"));
      M1.add(new MenuItem("Save"));
      M2.add(new MenuItem("General"));
      M2.add(M3);
      M3.add(new MenuItem("Screen"));
      M3.add(new MenuItem("Font"));
      setMenuBar(MB);
      resize(300, 200);
      show();
   }
   public static void main(String args[]) {
      testMe test = new testMe();
   }
}
```

The AWT also provides a subclass of the **MenuItem** class called **CheckboxMenuItem**. This class is identical to the standard **MenuItem** class

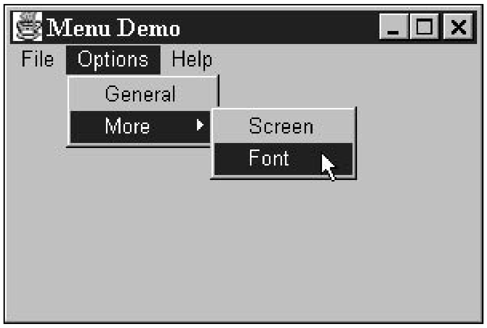


Figure 9.18
Creating cascading menus using Menus as subclasses.

except that it provides the ability to be "checked" or "unchecked" when the user clicks on a menu item. Figure 9.19 shows an example of this component.

Hierarchy for MenuItem

```
java.lang.Object
  java.awt.MenuComponent
    java.awt.MenuItem
```

Declaration for MenuItem

The **MenuItem** class provides a single constructor that takes one argument:

```
new MenuItem(String);
```

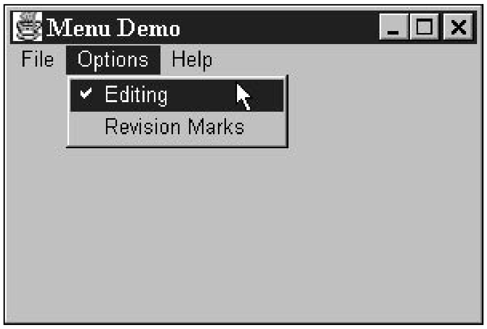


Figure 9.19
Using the CheckboxMenuItem to check and uncheck menu items.

This class constructs a new **MenuItem** with the specified **String** displayed as the menu component's label. Note that the hyphen symbol (-) is reserved to mean a separator between menu items. Separators should do nothing except delineate different sections of a menu.

Methods for MenuItem

The specialized methods for the **MenuItem** class include:

DISABLE()

This method makes the menu item "unselectable" by the user and grays it out.

ENABLE()

This method makes the menu item "selectable" by the user. The user is given a visual cue when the menu is disabled because it is grayed out.

ENABLE(BOOLEAN)

This method conditionally enables a component.

GETLABEL()

This method gets the label for the menu item. The value is returned as a string.

ISENABLED()

This method checks to see if the menu item is enabled. The return value is a boolean value.

SETLABEL()

This method sets the label to be the specified string.

The following two methods are used only with the **CheckboxMenuItem** component.

GETSTATE()

This method returns a boolean value that represents the state of the menu item.

SETSTATE(BOOLEAN)

This method sets the state of the menu item.

Creating a Sample Menu Application

Now that we've introduced you to each of the three key classes for creating menus (**MenuBar**, **Menu**, and **MenuItem**), you're probably anxious to write a Java program that puts them to work. Let's build a sample application that incorporates all of them to create a practical menuing system. The application will not respond to any of the menu items being chosen, but it will show you the basic techniques for constructing a menuing system.

We need to start by planning our menu. Let's use three **Menu** components with the labels "File," "Options," and "Help." Under the File menu we will have seven **MenuItems** including two separators. The "Options" **Menu** component will have two **CheckboxMenuItem** components, and the "Help" menu will not have any menu items associated with it.

Figures 9.20 through 9.22 show several different views of our test program with different menu options being chosen. The code that creates this menuing system is as follows:

```
import java.awt.*;

class TestFrame extends Frame {
   TestFrame() {
      super("Menu Test");
      MenuBar mb = new MenuBar();
      Menu fileMenu = new Menu("File");
      Menu optionMenu = new Menu("Option");
      Menu helpMenu = new Menu("Help");
      fileMenu.add(new MenuItem("New"));
      fileMenu.add(new MenuItem("-"));
      fileMenu.add(new MenuItem("Open"));
      fileMenu.add(new MenuItem("Close"));
      fileMenu.add(new MenuItem("Save"));
      fileMenu.addSeparator();
      fileMenu.add(new MenuItem("Exit"));
      optionMenu.add(new CheckboxMenuItem("Large Fonts"));
      optionMenu.add(new CheckboxMenuItem("Save Settings on Exit"));
      mb.setHelpMenu(helpMenu);
      mb.add(fileMenu);
      mb.add(optionMenu);
      mb.add(helpMenu);
      setMenuBar(mb);
      resize(300,200);
      show();
   }
   public boolean action(Event evt, Object obj) {
      String label = (String)obj;
      if (evt.target instanceof MenuItem) {
         if (label.equals("Exit")) {
            System.exit(0);
            return true;
            }
      }
      return true;
   }

   public boolean handleEvent(Event evt) {
      if (evt.id == Event.WINDOW_DESTROY) {
         System.exit(0);
         return true;
      }
      return super.handleEvent(evt);
   }
   public static void main(String args[]) {
      TestFrame tf = new TestFrame();
  }
}
```

You my notice that we used two different methods for creating the two separators in the "File" menu. The **addSeparator**() method is probably easier. However, if using the standard **add**() method with the hyphen character, you can create a full-fledged menu item that you can then change options for and set up so that it can respond to mouse clicks. Now that you've seen the basics for creating menus and GUI objects, you'll need a way to position them within the frames you build. And that's where the layout manager comes in.

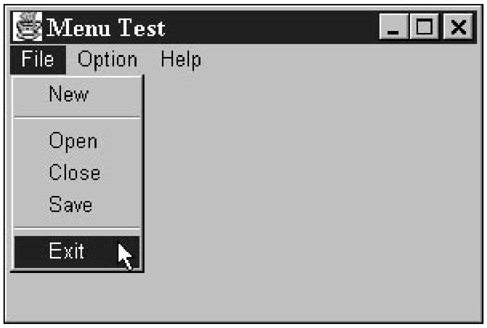


Figure 9.20

Menu test app view #1.

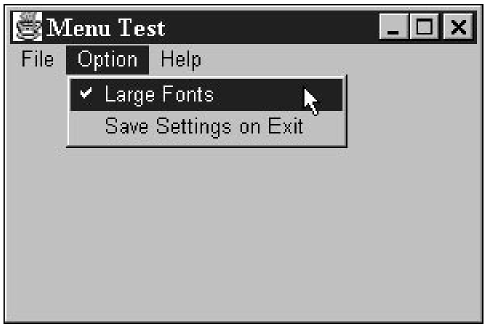


Figure 9.21
Menu test app view #2.

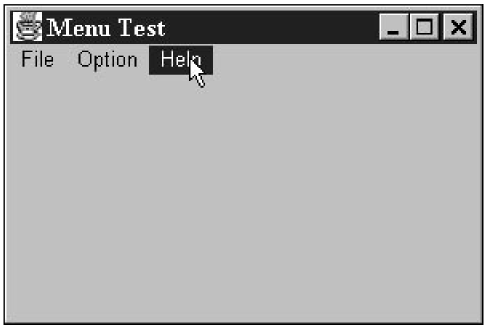


Figure 9.22
Menu test app view #3.

Working with the Layout Manager

To most programmers, the layout manager is a pretty strange concept. Instead of telling Java exactly where you want components to be located within a window, you use the layout manager to actively place interface components depending on the container size, shape, and on a few variables you set.

All controls you create for a Java application must *always* use the layout manager. Even if you do not implicitly tell Java what to do, it defaults to using a particular version of the layout manager. The layout manager comes in different "flavors" for different situations. At this point in the evolution of Java, five different layout managers are provided:

- FlowLayout
- BorderLayout
- GridLayout
- GridBagLayout
- CardLayout

The layout manager is actually just an Interface. You then create classes that implement the **LayoutManager** interface. These classes are used by the AWT and your program to get the look and feel you want for the user. Let's look at each of the layout manager classes in detail.

Any component that is a container in Java can use a different layout. So, you could have a **Frame** that uses one class to lay out two panels which each have their own layouts, and so on.

The FlowLayout Class

The **FlowLayout** class is the default layout for all containers in Java. This class tries to lay out components in a very orderly fashion that is much like a word processor that wraps your words and centers them. The order in which you add components to a container using **FlowLayout** is vital. The first component you add is the first in line and the last will be placed at the end. It is possible to go back and insert and remove components, but it is much easier to do it all correctly at the beginning.

Figure 9.23 shows several buttons on a panel. As you can see, they are centered as a group within the panel. If we resize the panel, the buttons automatically align themselves again to be centered. If we continue to shrink the panel so that all the buttons no longer fit across the panel, the last button will be "wrapped" to the next row as shown in Figure 9.24.

Declaration for FlowLayout

To set the layout for a container, you use the **setLayout**() method. This method has a single argument which is an instance of one of the layout classes. Let's look at an example. The code shown here is used to create the arrangements shown in Figures 9.23 and 9.24:

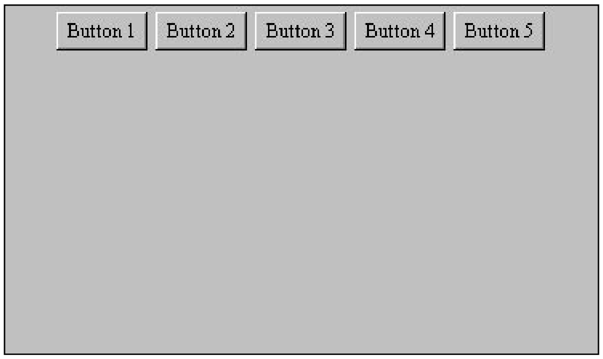


Figure 9.23
Using the FlowLayout class to lay out some buttons.

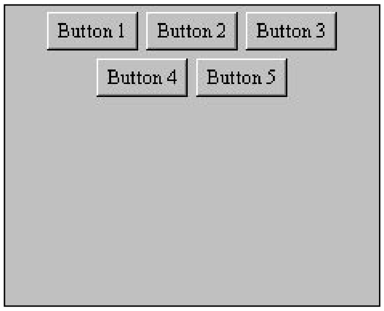


Figure 9.24
The same panel resized so that the buttons wrap to the next row.

```
import java.awt.*;

class LayoutFrame extends Frame {

   LayoutFrame() {
      super("Layout Test");
      setLayout(new FlowLayout());
      add(new Button("Button 1"));
      add(new Button("Button 2"));
      add(new Button("Button 3"));
      add(new Button("Button 4"));
      add(new Button("Button 5"));
      resize(300,200);
      show();
   }
public static void main(String args[]) {
      LayoutFrame lf = new LayoutFrame();
  }
}
```

Here we call the **setLayout**() method of the **Frame** we have extended, sending it a new instance of the **FlowLayout** class. The frame will then query the **FlowLayout** object where to position each component. This query happens whenever the window needs to be refreshed, such as when it is resized or uncovered.

Methods for FlowLayout

The specialized methods for **FlowLayout** include:

layoutContainer(Container)

This method lays out the container. It will actually reshape the components in the target container to satisfy the constraints of the **FlowLayout** object.

minimumLayoutSize (Container)

This method returns the minimum dimensions needed to lay out the components contained in the specified target container. These dimensions are extremely useful because they can help you ensure that the user will not shrink the container to such a small size that it forces some of the UI components to slip off the visible screen.

preferredLayoutSize(Container)

This method returns the preferred dimensions for this layout given the components in the specified target container. The return value is a **Dimension** variable consisting of two values representing the width and height of the layout.

toString()

This method returns a string representation of this **FlowLayout**'s values, including: (in this order) X position, Y position, container dimensions, layout class being used, whether or not the container can be resized, and the title of the container.

The BorderLayout Class

The **BorderLayout** class creates a unique layout scheme that is useful for working with components that need to maintain their position relative to an edge of your container. Border layouts use a compass analogy to allow you to specify which side of the container to attach your control to.

The controls you use for border layouts are automatically sized to take up as much space as possible. Figure 9.25 shows a sample container with five panels, one in each area of the container that the border layout allows.

To create this sample, we used the same code from the previous figure and made a few very minor changes. In fact, all the changes take place in the constructor for the class, so we will only show you that:

```
LayoutFrame() {
      super("Layout Test");
      setLayout(new BorderLayout());
      add("North", new Button("North"));
      add("East", new Button("East"));
      add("South", new Button("South"));
      add("West", new Button("West"));
      add("Center", new Button("Center"));
      resize(300,200);
      show();
   }
```

The first change is obviously the switch to specifying the **BordeLayout**() class as our layout scheme. The other changes occur in the **add**() method. What are

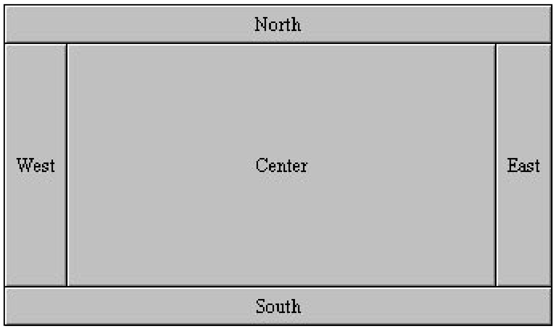


Figure 9.25
The BorderLayout() class in action.

those extra strings doing there? They specify which side of the container to place the new control. In this case, we are using buttons with labels that match the position names (North, East, South, West, and Center). We used buttons here for clarity sake, but you would probably not use them for any real projects. Panel components are probably the best for this type of layout. You would specify a few panels using a border style layout and then use a different layout scheme for the individual panels. Since panels are for design purposes mostly—they do not show up since they default to the same color as the background of the frame—they blend in perfectly and bring the whole thing together.

Declaration for BorderLayout

There are two ways to declare a border layout. You can use the simple constructor we used in the sample application or you can also specify a vertical and horizontal space.

```
new BorderLayout();
```

The simple way. Constructs a new **BorderLayout**.

```
new BorderLayout(int, int);
```

Constructs a **BorderLayout** with the specified gaps. The first integer represents the horizontal gap to be placed between components and the second integer represents the vertical gap to be used.

Methods for BorderLayout

The specialized methods for **BorderLayout** include:

addLayoutComponent(String, component)

This method adds the specified named component to the layout. The **String** argument gives us a name to refer to the component within the layout. The component can be any type of interface component we want to add.

layoutContainer(Container)

This method lays out the specified container. It will actually reshape the components in the specified target container to satisfy the constraints of the **BorderLayout** object.

minimumLayoutSize(Container)

This method returns the minimum dimensions needed to lay out the components contained in the specified target container. The return value is a dimension variable.

preferredLayoutSize(Container)

This method returns the preferred dimensions for the layout given the components in the specified target container. The method also returns a dimension style variable.

removeLayoutComponent(component)

This method removes the specified component from the layout.

toString()

This method returns the string representation of the **BorderLayout**'s values. At this point in the development of Java, this method only returns the size of the horizontal and vertical gaps.

The GridLayout Class

The grid style layout works just like it sounds; it creates a grid pattern for laying out components. You can control the number of rows and columns as well as the space between the components and the space between the components and the edge of the container. Figure 9.26 shows a sample frame using a grid layout.

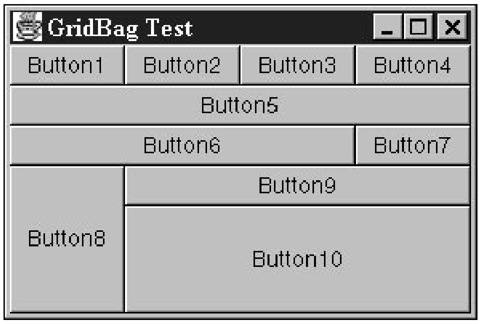


Figure 9.26
Using the GridLayout() class to create a sample frame.

Declaration for GridLayout

Like the **BorderLayout** class, **GridLayout** gives you a few constructor options. If the number of rows or columns is invalid, you will get an error (i.e., no negative numbers). Also, if you specify more columns than you need, the columns will not be used and your grid will appear as if you've specified fewer columns. However, if you specify more rows than you need, they will show up as blank space with enough space that matches the amount of space taken up by a component in this layout. We think this is a bug in Java, so you should experiment with this phenomenon if you plan on using this layout style.

```
new GridLayout(int, int);
```

Creates a grid layout with the specified rows and columns.

```
new GridLayout (int, int, int ,int);
```

Creates a grid layout with the specified rows, columns, horizontal gap, and vertical gap.

Methods for GridLayout

The specialized methods for **GridLayout** include:

addLayoutComponent(String, component)

This method adds the specified named component to the layout. The **String** argument gives us a name to refer to the component within terms of the layout. The component can be any interface component you want to add.

layoutContainer(Container)

This method lays out the specified container. It will actually reshape the components in the specified target container to satisfy the constraints of the**GridLayout** object.

minimumLayoutSize(Container)

This method returns the minimum dimensions needed to lay out the components contained in the specified target container. The return value is a dimension variable.

preferredLayoutSize(Container)

This method returns the preferred dimensions for this layout given the components in the specified target container. It also returns a dimension style variable.

removeLayoutComponent(component)

This method removes the specified component from the layout.

toString()

This method returns the string representation of the **GridLayout**'s values. At this point in the development of Java, this method only returns the size of the horizontal and vertical gaps.

The GridBagLayout Class

The **GridBagLayout** class is one of the most versatile of all the layout classes. It allows you to create a grid of an arbitrary size and use it to create components within the grid that are of variable size.

You use this layout method to align components vertically and horizontally, without requiring that the components be the same size or without them being sized for you in ways you may not want. Each **GridBagLayout** uses a rectangular grid of cells, with each component occupying one or more cells.

Each component that resides in a container that is using a **GridBagLayout** has an associated **GridBagConstraints** instance that specifies how the component is laid out within the grid. How a **GridBagLayout** places a set of components depends on each component's **GridBagConstraints**, minimum size, and preferred size.

To use a **GridBagLayout** effectively, you must customize one or more of its component's **GridBagConstraints**. Here are some of the variables you need to customize to create a layout:

- **gridx**, **gridy** Specifies the cell in the grid at the upper left of the component. The upper-left-most cell of a container has address gridx=0, gridy=0.
- **gridwidth**, **gridheight** Specifies the width and height of our component in grid space terms. You can set either of these to **GridBagConstraints.REMAINDER** to make the component be the last one in its row or column.

- **fill** Used when the component's display area is larger than the component's requested size to determine whether (and how) to resize the component.
- **ipadx**, **ipady** Specifies the internal padding. Padding represents the amount of space to add to the minimum size of the component. The width of the component will be at least its minimum width plus **ipadx***2 pixels (since the padding applies to both sides of the component). Similarly, the height of the component will be at least the minimum height plus **ipady***2 pixels.
- **insets** Sets the external padding of the component—the minimum amount of space between the component and the edges of its display area.
- **anchor** Used when the component is smaller than its display area to determine where to place the component. Valid values are:

```
GridBagConstraints.CENTER (the default)
GridBagConstraints.NORTH
GridBagConstraints.NORTHEAST
GridBagConstraints.EAST
GridBagConstraints.SOUTHEAST
GridBagConstraints.SOUTH
GridBagConstraints.SOUTHWEST
GridBagConstraints.WEST
GridBagConstraints.NORTHWEST
```

- **weightx, weighty** Used to determine how to distribute space; this is important for specifying resizing behavior. Unless you specify a weight for at least one component in a row and column, all the components clump together in the center of their container. This is because when the weight is zero (the default), the GridBagLayout puts any extra space between its grid of cells and the edges of the container.

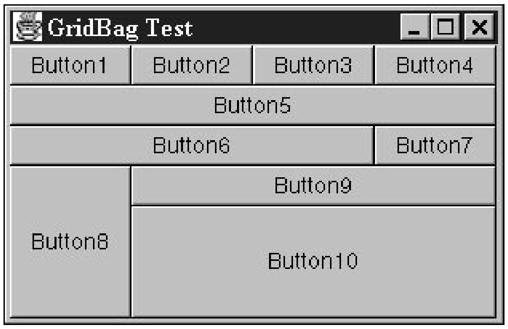


Figure 9.27

Sample program using the GridBagLayout class.

It is probably easiest to give you an example. Figure 9.27 shows the layout we wish to end up with. Following it is the code that we used to create it:

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

public class GridBagTest extends Frame {

   GridBagTest() {
      super("GridBag Test");
      GridBagLayout gridbag = new GridBagLayout();
      GridBagConstraints c = new GridBagConstraints();
      setFont(new Font("Helvetica", Font.PLAIN, 14));
      setLayout(gridbag);
      c.fill = GridBagConstraints.BOTH;
      c.weightx = 1.0;
      makebutton("Button1", gridbag, c);
      makebutton("Button2", gridbag, c);
      makebutton("Button3", gridbag, c);
      c.gridwidth = GridBagConstraints.REMAINDER; //end row
      makebutton("Button4", gridbag, c);
      c.weightx = 0.0;                   //reset to the default
      makebutton("Button5", gridbag, c); //another row
      c.gridwidth = GridBagConstraints.RELATIVE; //next-to-last in row
      makebutton("Button6", gridbag, c);
      c.gridwidth = GridBagConstraints.REMAINDER; //end row
      makebutton("Button7", gridbag, c);
      c.gridwidth = 1;                   //reset to the default
      c.gridheight = 2;
      c.weighty = 1.0;
      makebutton("Button8", gridbag, c);
      c.weighty = 0.0;                   //reset to the default
      c.gridwidth = GridBagConstraints.REMAINDER; //end row
      c.gridheight = 1;                  //reset to the default
      makebutton("Button9", gridbag, c);
      makebutton("Button10", gridbag, c);
      resize(300, 100);
      show();
   }

   protected void makebutton(String name, GridBagLayout gridbag,
                             GridBagConstraints c) {
      Button button = new Button(name);
      gridbag.setConstraints(button, c);
      add(button);
   }
   public static void main(String args[]) {
      GridBagTest test = new GridBagTest();
   }
}
```

Declaration for GridBagLayout

Since most of the work of setting up a **GridBagLayout** is achieved using a **GridBagConstraints** class, the constructor for **GridBagLayout** is very simple, in fact, it requires no arguments at all:

```
new GridBagLayout();
```

Methods for GridBagLayout

The specialized methods for **GridBagLayout** include:

DUMPCONSTRAINTS(GRIDBAGCONSTRAINTS)

This method prints the layout constraints to the **System** object. It is useful for debugging.

GETCONSTRAINTS(COMPONENT)

This method returns a copy of the **GridBagConstraints** object for the specified component.

LAYOUTCONTAINER(CONTAINER)

This method lays out the specified container. This method will actually reshape the components in the specified target container to satisfy the constraints of the **GridBagLayout** object.

LOOKUPCONSTRAINTS(COMPONENT)

This method retrieves the constraints for the specified component. The return value is not a copy, but is the actual constraints class used by the layout mechanism. The object returned by this method can then be altered to affect the looks of the component.

MINIMUMLAYOUTSIZE(CONTAINER)

This method returns the minimum dimensions needed to lay out the components contained in the specified target container. The return value is a dimension variable.

preferredLayoutSize(Container)

This method returns the preferred dimensions for this layout given the components in the specified target container. It also returns a dimension style variable.

setConstraints(Component, GridBagConstraints)

This method sets the constraints for the specified component.

toString()

This method returns the string representation of the **GridBagLayout**'s values. At this point in the development of Java, this method only returns the size of the horizontal and vertical gaps.

The CardLayout Class

The card layout style is much different than the previous four. Instead of using this class to lay out your controls, you use it like a layering system to specify which layer certain controls appear on. The most common use for this class is to simulate tabbed dialogs. You can create a card layout containing several panels that each use their own layout method and controls. Then, you can make a call to the class to specify which card to display and therefore which panel and respective components to show.

Declaration for CardLayout

Two versions of the **CardLayout**() constructor are available—one requires no parameters. To add extra cards, you simply use the **add**() method and the layout manager automatically handles the extra cards. Optionally, you can specify the horizontal and vertical gaps that surround your cards.

```
new CardLayout();
// Creates a new card layout.
new CardLayout(int, int);
```

Creates a card layout with the specified horizontal and vertical gaps.

Methods for CardLayout

The specialized methods for **CardLayout** include:

addLayoutComponent(String, component)

This method adds the specified named component to the layout. The **String** argument gives us a name to refer to the component within terms of the layout. The component can be any interface component you want to add.

first(Container)

This method flips to the first card. The argument is the parent container that you assigned the layout style to.

last (Container)

This method flips to the last card of the specified container.

layoutContainer(Container)

This method lays out the specified container. This method will actually reshape the components in the specified target container to satisfy the constraints of the **CardLayout** object.

minimumLayoutSize(Container)

This method returns the minimum dimensions needed to layout the components contained in the specified target container. The return value is a dimension variable.

next(Container)

This method flips to the next card in the stack.

preferredLayoutSize(Container)

This method returns the preferred dimensions for the layout given the components in the specified target container. This also returns a dimension style variable.

previous(Container)

This method flips to the previous card.

removeLayoutComponent(component)

This method removes the specified component from the layout.

show(Container, String)

This method flips to the specified component name in the specified container. This method is best used when you cannot use any of the previous four methods and/or you want to switch directly to a specified card. The **Container** argument specifies the owner of the card layout and the string is the name of the component you wish to switch to.

toString()

This method returns the string representation of the **CardLayout**'s values. At this point in the development of Java, this method only returns the size of the horizontal and vertical gaps.

Index