

# CHAPTER

## 6

# Introducing Classes

---

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

Because the class is so fundamental to Java, this and the next few chapters will be devoted to it. Here, you will be introduced to the basic elements of a class and learn how a class can be used to create objects. You will also learn about methods, constructors, and the **this** keyword.

## Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been shown. The classes created in the preceding chapters primarily exist simply to encapsulate the **main( )** method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far.

Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

# The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class's code defines the interface to its data.

A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a **class** definition is shown here:

```
class classname {  
    type instance-variable1;  
    type instance-variable2;  
    // ...  
    type instance-variableN;  
  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(parameter-list) {  
        // body of method  
    }  
    // ...  
    type methodnameN(parameter-list) {  
        // body of method  
    }  
}
```

The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class's data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early.

All methods have the same general form as **main()**, which we have been using thus far. However, most methods will not be specified as **static** or **public**. Notice that the general form of a class does not specify a **main()** method. Java classes do not need to have a **main()** method. You only specify one if that class is the starting point for your program. Further, some kinds of Java applications don't require a **main()** method at all.

## A Simple Class

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods (but some will be added soon).

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

As stated, a class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, **mybox** will refer to an instance of **Box**. Thus, it will have “physical” reality. For the moment, don't worry about the details of this statement.

As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Box** object will contain its own copies of the instance variables **width**, **height**, and **depth**. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the **width** variable of **mybox** the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of **width** that is contained within the **mybox** object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object. One other point: Although commonly referred to as the *dot operator*, the formal specification for Java categorizes the . as a separator. However, since the use of the term “dot operator” is widespread, it is used in this book.

Here is a complete program that uses the **Box** class:

```
/* A program that uses the Box class.  
   Call this file BoxDemo.java  
*/  
class Box {  
    double width;  
    double height;  
    double depth;  
}  
  
// This class declares an object of type Box.  
class BoxDemo {  
    public static void main(String[] args) {  
        Box mybox = new Box();  
        double vol;  
  
        // assign values to mybox's instance variables  
        mybox.width = 10;  
        mybox.height = 20;  
        mybox.depth = 15;  
  
        // compute volume of box  
        vol = mybox.width * mybox.height * mybox.depth;  
  
        System.out.println("Volume is " + vol);  
    }  
}
```

You should call the file that contains this program **BoxDemo.java**, because the **main()** method is in the class called **BoxDemo**, not the class called **Box**. When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively.

To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

```
Volume is 3000.0
```

As stated earlier, each object has its own copies of the instance variables. This means that if you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two **Box** objects:

```
// This program declares two Box objects.

class Box {
    double width;
    double height;
    double depth;
}

class BoxDemo2 {
    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);

        // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
    }
}
```

The output produced by this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, **mybox1**'s data is completely separate from the data contained in **mybox2**.

## Declaring Objects

As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, essentially, the address in memory of the object allocated by **new**. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

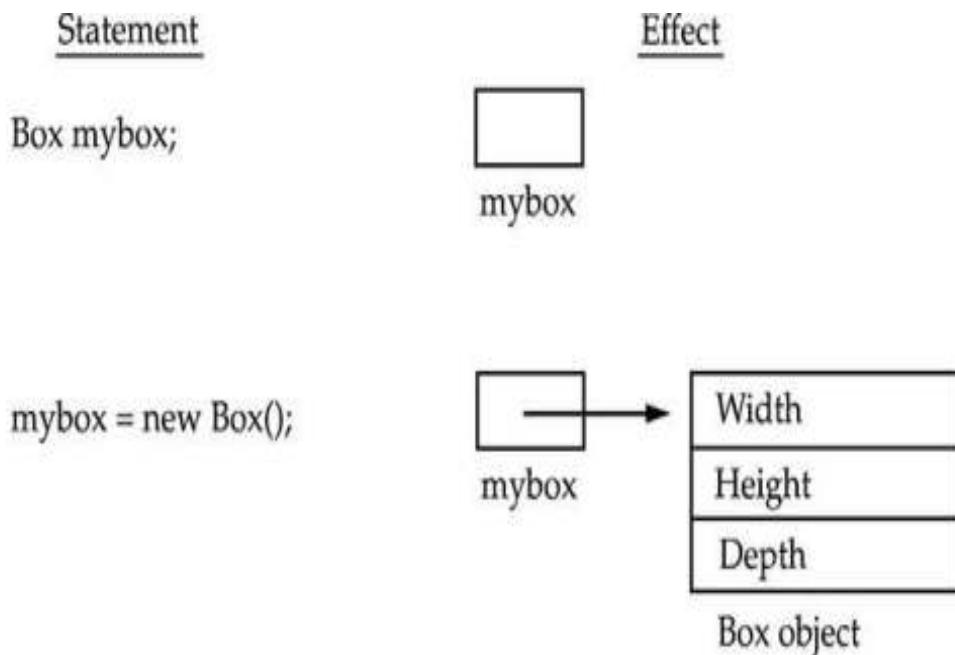
```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. At this point, **mybox** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality,

`mybox` simply holds, in essence, the memory address of the actual **Box** object. The effect of these two lines of code is depicted in [Figure 6-1](#).



**Figure 6-1** Declaring an object of type **Box**

## A Closer Look at `new`

As just explained, the `new` operator dynamically allocates memory for an object. In the context of an assignment, it has this general form:

*class-var* = new *classname* ( );

Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor. Soon, you will see how to define your own constructors.

At this point, you might be wondering why you do not need to use **new** for such things as integers or characters. The answer is that Java's primitive types are not implemented as objects. Rather, they are implemented as "normal" variables. This is done in the interest of efficiency. As you will see, objects have many features and attributes that require Java to treat them differently than it treats the primitive types. By not applying the same overhead to the primitive types that applies to objects, Java can implement the primitive types more efficiently. Later, you will see object versions of the primitive types that are available for your use in those situations in which complete objects of these types are needed.

It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. (You will learn how to handle exceptions in [Chapter 10](#).) For the sample programs in this book, you won't need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write.

Let's once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.

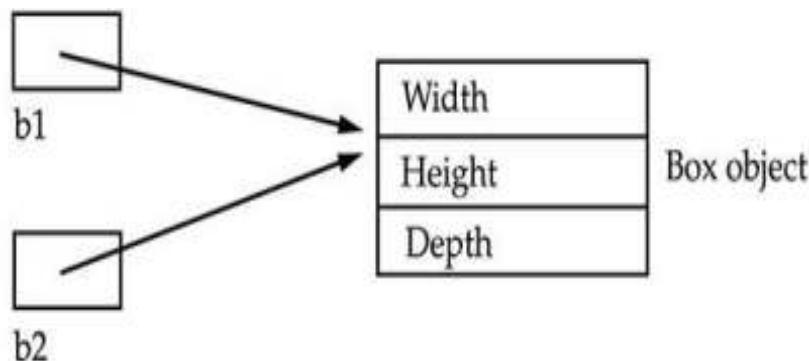
## Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

This situation is depicted here:



Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.

---

**REMEMBER** When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

## Introducing Methods

As mentioned at the beginning of this chapter, classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility. In fact, much

of the next chapter is devoted to methods. However, there are some fundamentals that you need to learn now so that you can begin to add methods to your classes.

This is the general form of a method:

```
type name(parameter-list) {  
    // body of method  
}
```

Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

```
return value;
```

Here, *value* is the value returned.

In the next few sections, you will see how to create various types of methods, including those that take parameters and those that return values.

## Adding a Method to the Box Class

Although it is perfectly fine to create a class that contains only data, it rarely happens. Most of the time, you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

Let's begin by adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// This program includes a method inside the box class.

class Box {
    double width;
    double height;
    double depth;

    // display volume of a box
    void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}

class BoxDemo3 {
    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
```

```
/* assign different values to mybox2's
   instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;

// display volume of first box
mybox1.volume();

// display volume of second box
mybox2.volume();
}
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Look closely at the following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

The first line here invokes the **volume( )** method on **mybox1**. That is, it calls **volume( )** relative to the **mybox1** object, using the object's name followed by the dot operator. Thus, the call to **mybox1.volume( )** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume( )** displays the volume of the box defined by **mybox2**. Each time **volume( )** is invoked, it displays the volume for the specified box.

If you are unfamiliar with the concept of calling a method, the following discussion will help clear things up. When **mybox1.volume( )** is executed, the Java run-time system transfers control to the code defined inside **volume( )**. After the statements inside **volume( )** have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. In the most general sense, a method is Java's way of implementing subroutines.

There is something very important to notice inside the **volume()** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that **width**, **height**, and **depth** inside **volume()** implicitly refer to the copies of those variables found in the object that invokes **volume()**.

Let's review: When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods.

## Returning a Value

While the implementation of **volume()** does move the computation of a box's volume inside the **Box** class where it belongs, this is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box but not display its value? A better way to implement **volume()** is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
// Now, volume() returns the volume of a box.

class Box {
    double width;
    double height;
    double depth;

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo4 {
    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

As you can see, when **volume( )** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume( )**. Thus, after

```
vol = mybox1.volume();
```

executes, the value of **mybox1.volume( )** is 3000 and this value then is stored in **vol**.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

One more point: The preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. The call to **volume( )** could have been used in the **println( )** statement directly, as shown here:

```
System.out.println("Volume is" + mybox1.volume());
```

In this case, when **println( )** is executed, **mybox1.volume( )** will be called automatically and its value will be passed to **println( )**.

## Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()
{
    return 10 * 10;
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make **square( )** much more useful.

```
int square(int i)
{
    return i * i;
}
```

Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

In the first call to **square( )**, the value 5 will be passed into parameter **i**. In the second call, **i** will receive the value 9. The third invocation passes the value of **y**, which is 2 in this example. As these examples show, **square( )** is able to return the square of whatever data it is passed.

It is important to keep the two terms *parameter* and *argument* straight. A *parameter* is a variable defined by a method that receives a value when the method is called. For example, in **square( )**, **i** is a parameter. An *argument* is a value that is passed to a method when it is invoked. For example, **square(100)** passes 100 as an argument. Inside **square( )**, the parameter **i** receives that value.

You can use a parameterized method to improve the **Box** class. In the preceding examples, the dimensions of each box had to be set separately by use of a sequence of statements, such as:

```
mybox1.width = 10;  
mybox1.height = 20;  
mybox1.depth = 15;
```

While this code works, it is troubling for two reasons. First, it is clumsy and error prone. For example, it would be easy to forget to set a dimension. Second, in well-designed Java programs, instance variables should be accessed only through methods defined by their class. In the future, you can change the behavior of a method, but you can't change the behavior of an exposed instance variable.

Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions of a box in its parameters and sets each instance variable appropriately. This concept is implemented by the following program:

```
// This program uses a parameterized method.  
  
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
  
    // sets dimensions of box  
    void setDim(double w, double h, double d) {  
        width = w;  
        height = h;
```

```

        depth = d;
    }
}

class BoxDemo5 {
    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // initialize each box
        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}

```

As you can see, the **setDim( )** method is used to set the dimensions of each box. For example, when

```
mybox1.setDim(10, 20, 15);
```

is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**. Inside **setDim( )** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

For many readers, the concepts presented in the preceding sections will be familiar. However, if such things as method calls, arguments, and parameters are new to you, then you might want to take some time to experiment before moving on. The concepts of the method invocation, parameters, and return values are fundamental to Java programming.

## Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience methods like `setDim()`, it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the `new` operator completes. Constructors look a little strange because they have no return type, not even `void`. This is because the implicit return type of a class's constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace `setDim()` with a constructor. Let's begin by defining a simple constructor that sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo6 {
    public static void main(String[] args) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println( )** statement inside **Box( )** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object.

Before moving on, let's reexamine the **new** operator. As you know, when you allocate an object, you use the following general form:

```
class-var = new classname ( );
```

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

```
Box mybox1 = new Box();
```

**new Box( )** is calling the **Box( )** constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor. When using the default constructor, all non-initialized instance variables will have their default values, which are zero, **null**, and **false**, for numeric types, reference types, and **boolean**, respectively. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

## Parameterized Constructors

While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess,

this makes it much more useful. For example, the following version of **Box** defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
    double width;
    double height;
    double depth;

    // This is the constructor for Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // compute and return volume
    double volume() {
```

```

        return width * height * depth;
    }
}

class BoxDemo7 {
    public static void main(String[] args) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}

```

The output from this program is shown here:

```

Volume is 3000.0
Volume is 162.0

```

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the **Box()** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

## The **this** Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any

method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class's type is permitted.

To better understand what **this** refers to, consider the following version of **Box( )**:

```
// A redundant use of this.  
Box(double w, double h, double d) {  
    this.width = w;  
    this.height = h;  
    this.depth = d;  
}
```

This version of **Box( )** operates exactly like the earlier version. The use of **this** is redundant but perfectly correct. Inside **Box( )**, **this** will always refer to the invoking object. While it is redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

## Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, that overlap with the names of the class's instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box( )** constructor inside the **Box** class. If they had been, then **width**, for example, would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables. For example, here is another version of **Box( )**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.  
Box(double width, double height, double depth) {  
    this.width = width;  
    this.height = height;  
    this.depth = depth;  
}
```

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity—and use **this** to overcome the instance variable hiding. It is a matter of taste which approach you adopt.

## Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as traditional C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no need to explicitly destroy objects. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

## A Stack Class

While the **Box** class is useful to illustrate the essential elements of a class, it is of little practical value. To show the real power of classes, this chapter will conclude with a more sophisticated example. As you recall from the discussion of object-oriented programming (OOP) presented in [Chapter 2](#),

one of OOP’s most important benefits is the encapsulation of data and the code that manipulates that data. As you have seen, the class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it. Further, the methods define a consistent and controlled interface to the class’s data. Thus, you can use the class through its methods without having to worry about the details of its implementation or how the data is actually managed within the class. In a sense, a class is like a “data engine.” No knowledge of what goes on inside the engine is required to use the engine through its controls. In fact, since the details are hidden, its inner workings can be changed as needed. As long as your code uses the class through its methods, internal details can change without causing side effects outside the class.

To see a practical application of the preceding discussion, let’s develop one of the archetypal examples of encapsulation: the stack. A *stack* stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called *push* and *pop*. To put an item on top of the stack, you will use push. To take an item off the stack, you will use pop. As you will see, it is easy to encapsulate the entire stack mechanism.

Here is a class called **Stack** that implements a stack for up to ten integers:

```
// This class defines an integer stack that can hold 10 values
class Stack {
    int[] stck = new int[10];
    int tos;

    // Initialize top-of-stack
    Stack() {
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==9)
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}
```

As you can see, the **Stack** class defines two data items, two methods, and a constructor. The stack of integers is held by the array **stck**. This array is indexed by the variable **tos**, which always contains the index of the top of the stack. The **Stack()** constructor initializes **tos** to  $-1$ , which indicates an empty stack. The method **push()** puts an item on the stack. To retrieve an item, call **pop()**. Since access to the stack is through **push()** and **pop()**, the fact that the stack is held in an array is actually not relevant to using the stack. For example, the stack could be held in a more complicated data structure, such as a linked list, yet the interface defined by **push()** and **pop()** would remain the same.

The class **TestStack**, shown here, demonstrates the **Stack** class. It creates two integer stacks, pushes some values onto each, and then pops them off.

```
class TestStack {  
    public static void main(String[] args) {  
        Stack mystack1 = new Stack();  
        Stack mystack2 = new Stack();  
  
        // push some numbers onto the stack  
        for(int i=0; i<10; i++) mystack1.push(i);  
        for(int i=10; i<20; i++) mystack2.push(i);  
  
        // pop those numbers off the stack  
        System.out.println("Stack in mystack1:");  
        for(int i=0; i<10; i++)  
            System.out.println(mystack1.pop());  
  
        System.out.println("Stack in mystack2:");  
        for(int i=0; i<10; i++)  
            System.out.println(mystack2.pop());  
    }  
}
```

This program generates the following output:

```
Stack in mystack1:  
9  
8  
7  
6  
5  
4  
3  
2  
1  
0
```

```
Stack in mystack2:  
19  
18  
17  
16  
15  
14  
13  
12  
11  
10
```

As you can see, the contents of each stack are separate.

One last point about the **Stack** class. As it is currently implemented, it is possible for the array that holds the stack, **stck**, to be altered by code outside of the **Stack** class. This leaves **Stack** open to misuse or mischief. In the next chapter, you will see how to remedy this situation.

# CHAPTER

## 7

# A Closer Look at Methods and Classes

---

This chapter continues the discussion of methods and classes begun in the preceding chapter. It examines several topics relating to methods, including overloading, parameter passing, and recursion. The chapter then returns to the class, discussing access control, the use of the keyword **static**, and one of Java's most important built-in classes: **String**.

## Overloading Methods

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as *method overloading*. Method overloading is one of the ways that Java supports polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an

overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    // Overload test for one integer parameter.
    void test(int a) {
        System.out.println("a: " + a);
    }

    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }

    // Overload test for a double parameter
    double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}

class Overload {
    public static void main(String[] args) {
        OverloadDemo ob = new OverloadDemo();
        double result;

        // call all versions of test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

As you can see, **test()** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double** parameter. The fact that the fourth version of **test()** also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution. For example, consider the following program:

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
}
```

```
// Overload test for a double parameter
void test(double a) {
    System.out.println("Inside test(double) a: " + a);
}
}

class Overload {
    public static void main(String[] args) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;

        ob.test();
        ob.test(10, 20);

        ob.test(i); // this will invoke test(double)
        ob.test(123.2); // this will invoke test(double)
    }
}
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88.0
Inside test(double) a: 123.2
```

As you can see, this version of **OverloadDemo** does not define **test(int)**. Therefore, when **test()** is called with an integer argument inside **Overload**, no matching method is found. However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call. Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**. Of course, if **test(int)** had been defined, it would have been called instead. Java will employ its automatic type conversions only if no exact match is found.

Method overloading supports polymorphism because it is one way that Java implements the “one interface, multiple methods” paradigm. To understand how, consider the following. In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In languages

that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function **abs( )** returns the absolute value of an integer, **labs( )** returns the absolute value of a long integer, and **fabs( )** returns the absolute value of a floating-point value. Since C does not support overloading, each function has its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in Java, because each absolute value method can use the same name. Indeed, Java's standard class library includes an absolute value method, called **abs( )**. This method is overloaded by Java's **Math** class to handle all numeric types. Java determines which version of **abs( )** to call based upon the type of argument.

The value of overloading is that it allows related methods to be accessed by use of a common name. Thus, the name **abs** represents the *general action* that is being performed. It is left to the compiler to choose the right *specific* version for a particular circumstance. You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help you manage greater complexity.

When you overload a method, each version of that method can perform any activity you desire. There is no rule stating that overloaded methods must relate to one another. However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not. For example, you could use the name **sqr** to create methods that return the *square* of an integer and the *square root* of a floating-point value. But these two operations are fundamentally different. Applying method overloading in this manner defeats its original purpose. In practice, you should only overload closely related operations.

## Overloading Constructors

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create,

overloaded constructors will be the norm, not the exception. To understand why, let's return to the **Box** class developed in the preceding chapter. Following is the latest version of **Box**:

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // This is the constructor for Box.  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
}
```

As you can see, the **Box( )** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box( )** constructor. For example, the following statement is currently invalid:

```
Box ob = new Box();
```

Since **Box( )** requires three arguments, it's an error to call it without them. This raises some important questions. What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? As the **Box** class is currently written, these other options are not available to you.

Fortunately, the solution to these problems is quite easy: simply overload the **Box** constructor so that it handles the situations just described. Here is a program that contains an improved version of **Box** that does just that:

```
/* Here, Box defines three constructors to initialize
   the dimensions of a box various ways.
*/
class Box {
    double width;
    double height;
    double depth;

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons {
    public static void main(String[] args) {
        // create boxes using the various constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        double vol;
```

```
// get volume of cube  
vol = mycube.volume();  
System.out.println("Volume of mycube is " + vol);  
}  
}  
// get volume of second box  
vol = mybox2.volume();  
The output produced by this program is shown here:  
System.out.println("Volume of mybox2 is " + vol);  
  
Volume of mybox1 is 3000.0  
Volume of mybox2 is -1.0  
Volume of mycube is 343.0
```

As you can see, the proper overloaded constructor is called based upon the arguments specified when **new** is executed.

## Using Objects as Parameters

So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```
// Objects may be passed to methods.
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // return true if o is equal to the invoking object
    boolean equalTo(Test o) {
        if(o.a == a && o.b == b) return true;
        else return false;
    }
}

class PassOb {
    public static void main(String[] args) {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);

        System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

As you can see, the **equalTo()** method inside **Test** compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns **true**. Otherwise, it returns **false**. Notice that the parameter **o** in **equalTo()** specifies **Test** as its type. Although **Test** is a class type created by the program, it is used in just the same way as Java's built-in types.

One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter. For example, the following version of **Box** allows one object to initialize another:

```
// Here, Box allows one object to initialize another.

class Box {
    double width;
    double height;
    double depth;

    // Notice this constructor. It takes an object of type Box.
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons2 {
    public static void main(String[] args) {
        // create boxes using the various constructors
    }
}
```

```
Box mybox2 = new Box();
Box mycube = new Box(7);

Box myclone = new Box(mybox1); // create copy of mybox1

double vol;

// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);

// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);

// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);

// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

As you will see when you begin to create your own classes, providing many forms of constructors is usually required to allow objects to be constructed in a convenient and efficient manner.

## A Closer Look at Argument Passing

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is *call-by-value*. This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is *call-by-reference*. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This

means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, although Java uses call-by-value to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.

When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Primitive types are passed by value.
class Test {
    void meth(int i, int j) {
        i *= 2;
        j /= 2;
    }
}

class CallByValue {
    public static void main(String[] args) {
        Test ob = new Test();

        int a = 15, b = 20;

        System.out.println("a and b before call: " +
                           a + " " + b);

        ob.meth(a, b);

        System.out.println("a and b after call: " +
                           a + " " + b);
    }
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

As you can see, the operations that occur inside **meth()** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30

and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument. For example, consider the following program:

```
// Objects are passed through their references.

class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // pass an object
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}

class PassObjRef {
    public static void main(String[] args) {
        Test ob = new Test(15, 20);
```

```
System.out.println("ob.a and ob.b before call: " +
    ob.a + " " + ob.b);

ob.meth(ob);

System.out.println("ob.a and ob.b after call: " +
    ob.a + " " + ob.b);
}
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

As you can see, in this case, the actions inside **meth()** have affected the object used as an argument.

---

**REMEMBER** When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

## Returning Objects

A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen()** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

```
// Returning an object.
class Test {
    int a;

    Test(int i) {
        a = i;
    }

    Test incrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}

class RetOb {
    public static void main(String[] args) {
        Test ob1 = new Test(2);
        Test ob2;

        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);

        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after second increase: "
                           + ob2.a);
    }
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

As you can see, each time **incrByTen()** is invoked, a new object is created, and a reference to it is returned to the calling routine.

The preceding program makes another important point: Since all objects are dynamically allocated using **new**, you don't need to worry about an object going out of scope because the method in which it was created

terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

## Recursion

Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number  $N$  is the product of all the whole numbers between 1 and  $N$ . For example, 3 factorial is  $1 \times 2 \times 3$ , or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial {
    // this is a recursive method
    int fact(int n) {
        int result;

        if(n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}

class Recursion {
    public static void main(String[] args) {
        Factorial f = new Factorial();

        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 4 is " + f.fact(4));
        System.out.println("Factorial of 5 is " + f.fact(5));
    }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

If you are unfamiliar with recursive methods, then the operation of **fact()** may seem a bit confusing. Here is how it works. When **fact()** is called with an argument of 1, the function returns 1; otherwise, it returns the product of **fact(n-1)\*n**. To evaluate this expression, **fact()** is called with **n-1**. This process repeats until **n** equals 1 and the calls to the method begin returning.

To better understand how the **fact()** method works, let's go through a short example. When you compute the factorial of 3, the first call to **fact()** will cause a second call to be made with an argument of 2. This invocation will cause **fact()** to be called a third time with an argument of 1. This call will return 1, which is then multiplied by 2 (the value of **n** in the second invocation). This result (which is 2) is then returned to the original invocation of **fact()** and multiplied by 3 (the original value of **n**). This yields the answer, 6. You might find it interesting to insert **println()** statements into **fact()**, which will show at what level each call is and what the intermediate answers are.

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to “telescope” out and back.

Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional method calls. A large number of recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception. However, this is typically not an issue unless a recursive routine runs wild.

The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives. For example, the QuickSort sorting algorithm is quite

difficult to implement in an iterative way. Also, some types of AI-related algorithms are most easily implemented using recursive solutions.

When writing recursive methods, you must have an **if** statement somewhere to force the method to return without the recursive call being executed. If you don't do this, once you call the method, it will never return. This is a very common error in working with recursion. Use **println( )** statements liberally during development so that you can watch what is going on and abort execution if you see that you have made a mistake.

Here is one more example of recursion. The recursive method **printArray( )** prints the first **i** elements in the array **values**.

```
// Another example that uses recursion.

class RecTest {
    int[] values;

    RecTest(int i) {
        values = new int[i];
    }

    // display array -- recursively
    void printArray(int i) {
        if(i==0) return;
        else printArray(i-1);
        System.out.println("[" + (i-1) + "] " + values[i-1]);
    }
}

class Recursion2 {
    public static void main(String[] args) {
        RecTest ob = new RecTest(10);
        int i;

        for(i=0; i<10; i++) ob.values[i] = i;
        ob.printArray(10);
    }
}
```

This program generates the following output:

```
[0] 0  
[1] 1  
[2] 2  
[3] 3  
[4] 4  
[5] 5  
[6] 6  
[7] 7  
[8] 8  
[9] 9
```

## Introducing Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control*. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Thus, when correctly implemented, a class creates a “black box” that may be used, but the inner workings of which are not open to tampering. However, the classes that were presented earlier do not completely meet this goal. For example, consider the **Stack** class shown at the end of [Chapter 6](#). While it is true that the methods **push()** and **pop()** do provide a controlled interface to the stack, this interface is not enforced. That is, it is possible for another part of the program to bypass these methods and access the stack directly. Of course, in the wrong hands, this could lead to trouble. In this section, you will be introduced to the mechanism by which you can precisely control access to the various members of a class.

How a member can be accessed is determined by the *access modifier* attached to its declaration. Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance or packages. (A *package* is, essentially, a grouping of classes.) These parts of Java’s access control mechanism will be discussed in subsequent chapters. Here, let’s begin by examining access control as it applies to a single class.

Once you understand the fundamentals of access control, the rest will be easy.

---

**NOTE** The modules feature added by JDK 9 can also impact accessibility. Modules are described in [Chapter 16](#).

Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved. The other access modifiers are described next.

Let's begin by defining **public** and **private**. When a member of a class is modified by **public**, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class. Now you can understand why **main( )** has always been preceded by the **public** modifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. (Packages are discussed in [Chapter 9](#).)

In the classes developed so far, all members of a class have used the default access mode. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.

An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;  
private double j;  
  
private int myMethod(int a, char b) { //...
```

To understand the effects of public and private access, consider the following program:

```
/* This program demonstrates the difference between
   public and private.
*/
class Test {
    int a; // default access
    public int b; // public access
    private int c; // private access

    // methods to access c
    void setc(int i) { // set c's value
        c = i;
    }
    int getc() { // get c's value
        return c;
    }
}

class AccessTest {
    public static void main(String[] args) {
        Test ob = new Test();

        // These are OK, a and b may be accessed directly
        ob.a = 10;
        ob.b = 20;

        // This is not OK and will cause an error
        // ob.c = 100; // Error!

        // You must access c through its methods
        ob.setc(100); // OK
        System.out.println("a, b, and c: " + ob.a + " " +
                           ob.b + " " + ob.getc());
    }
}
```

As you can see, inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**. Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods:

**setc( )** and **getc( )**). If you were to remove the comment symbol from the beginning of the following line,

```
// ob.c = 100; // Error!
```

then you would not be able to compile this program because of the access violation.

To see how access control can be applied to a more practical example, consider the following improved version of the **Stack** class shown at the end of [Chapter 6](#):

```
// This class defines an integer stack that can hold 10 values.
class Stack {
    /* Now, both stck and tos are private. This means
       that they cannot be accidentally or maliciously
       altered in a way that would be harmful to the stack.
    */
    private int[] stck = new int[10];
    private int tos;

    // Initialize top-of-stack
    Stack() {
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==9)
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }
}
```

```
// Pop an item from the stack
int pop() {
    if(tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
    }
    else
        return stck[tos--];
}
}
```

As you can see, now both **stck**, which holds the stack, and **tos**, which is the index of the top of the stack, are specified as **private**. This means that they cannot be accessed or altered except through **push( )** and **pop( )**. Making **tos** private, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the **stck** array.

The following program demonstrates the improved **Stack** class. Try removing the commented-out lines to prove to yourself that the **stck** and **tos** members are, indeed, inaccessible.

```

class TestStack {
    public static void main(String[] args) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // push some numbers onto the stack
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");

        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());

        // these statements are not legal
        // mystack1.tos = -2;
        // mystack2.stck[3] = 100;
    }
}

```

Although methods will usually provide access to the data defined by a class, this does not always have to be the case. It is perfectly proper to allow an instance variable to be public when there is good reason to do so. For example, most of the simple classes in this book were created with little concern about controlling access to instance variables for the sake of simplicity. However, in most real-world classes, you will need to allow operations on data only through methods. The next chapter will return to the topic of access control. As you will see, it is particularly important when inheritance is involved.

## **Understanding static**

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member

must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main()**. **main()** is declared as **static** because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

- They can only directly call other **static** methods of their class.
- They can only directly access **static** variables of their class.
- They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance and is described in the next chapter.)

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
    static int a = 3;
    static int b;

    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String[] args) {
        meth(42);
    }
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes, which prints a message and then initializes **b** to **a\*4** or **12**. Then **main()** is called, which calls **meth()**, passing **42** to **x**. The three **println()** statements refer to the two **static** variables **a** and **b**, as well as to the parameter **x**.

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

*classname.method()*

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
class StaticDemo {  
    static int a = 42;  
    static int b = 99;  
  
    static void callme() {  
        System.out.println("a = " + a);  
    }  
}  
  
class StaticByName {  
    public static void main(String[] args) {  
        StaticDemo.callme();  
        System.out.println("b = " + StaticDemo.b);  
    }  
}
```

Here is the output of this program:

```
a = 42  
b = 99
```

## Introducing **final**

A field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a **final** field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is probably the most common. Here is an example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use **FILE\_OPEN**, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for **final** fields, as this example shows.

In addition to fields, both method parameters and local variables can be declared **final**. Declaring a parameter **final** prevents it from being changed within the method. Declaring a local variable **final** prevents it from being assigned a value more than once.

The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This additional usage of **final** is explained in the next chapter, when inheritance is described.

## Arrays Revisited

Arrays were introduced earlier in this book, before classes had been discussed. Now that you know about classes, an important point can be made about arrays: they are implemented as objects. Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable. All arrays have this variable, and it will always hold the size of the array. Here is a program that demonstrates this property:

```
// This program demonstrates the length array member.  
class Length {  
    public static void main(String[] args) {  
        int[] a1 = new int[10];  
        int[] a2 = {3, 5, 7, 1, 8, 99, 44, -10};  
        int[] a3 = {4, 3, 2, 1};  
  
        System.out.println("length of a1 is " + a1.length);  
        System.out.println("length of a2 is " + a2.length);  
        System.out.println("length of a3 is " + a3.length);  
    }  
}
```

This program displays the following output:

```
length of a1 is 10  
length of a2 is 8  
length of a3 is 4
```

As you can see, the size of each array is displayed. Keep in mind that the value of **length** has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

You can put the **length** member to good use in many situations. For example, here is an improved version of the **Stack** class. As you might recall, the earlier versions of this class always created a ten-element stack. The following version lets you create stacks of any size. The value of **stck.length** is used to prevent the stack from overflowing.

```
// Improved Stack class that uses the length array member.
class Stack {
    private int[] stck;
    private int tos;

    // allocate and initialize stack
    Stack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==stck.length-1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class TestStack2 {
    public static void main(String[] args) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);

        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
    }
}
```

Notice that the program creates two stacks: one five elements deep and the other eight elements deep. As you can see, the fact that arrays maintain their own length information makes it easy to create stacks of any size.

```
for(int i=0; i<8; i++)
    System.out.println(mystack2.pop());
```

## Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: *static* and *inner*. A static nested class is one that has the **static** modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly.

The second type of nested class is the *inner* class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer\_x**, one instance method named **test( )**, and defines one inner class called **Inner**.

```

// Demonstrate an inner class.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class
    class Inner {
        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
}

class InnerClassDemo {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}

```

Output from this application is shown here:

```
display: outer_x = 100
```

In the program, an inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer\_x**. An instance method named **display()** is defined inside **Inner**. This method displays **outer\_x** on the standard output stream. The **main()** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test()** method. That method creates an instance of class **Inner**, and the **display()** method is called.

It is important to realize that an instance of **Inner** can be created only in the context of class **Outer**. The Java compiler generates an error message otherwise. In general, an inner class instance is often created by code within its enclosing scope, as the example does.

As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example:

```
// This program will not compile.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class
    class Inner {
        int y = 10; // y is local to Inner

        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }

    void showy() {
        System.out.println(y); // error, y not known here!
    }
}

class InnerClassDemo {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Here, **y** is declared as an instance variable of **Inner**. Thus, it is not known outside of that class and it cannot be used by **showy()**.

Although we have been focusing on inner classes declared as members within an outer class scope, it is possible to define inner classes within any block scope. For example, you can define a nested class within the block

```
// Define an inner class within a for loop.
class Outer {
    int outer_x = 100;

    void test() {
        for(int i=0; i<10; i++) {
            class Inner {
                void display() {
                    System.out.println("display: outer_x = " + outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}

class InnerClassDemo {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

While nested classes are not applicable to all situations, they are particularly helpful when handling events. We will return to the topic of nested classes in [Chapter 25](#). There you will see how inner classes can be used to simplify the code needed to handle certain types of events. You will also learn about *anonymous inner classes*, which are inner classes that don't have a name.

One point of interest: Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1.

## Exploring the String Class

Although the **String** class will be examined in depth in Part II of this book, a short exploration of it is warranted now, because we will be using strings in some of the sample programs shown toward the end of Part I. **String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.

The first thing to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects. For example, in the statement

```
System.out.println("This is a String, too");
```

the string "This is a String, too" is a **String** object.

The second thing to understand about strings is that objects of type **String** are immutable; once a **String** object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:

- If you need to change a string, you can always create a new one that contains the modifications.
- Java defines peer classes of **String**, called **StringBuffer** and **StringBuilder**, that allow strings to be altered, so all of the normal string manipulations are still available in Java. (**StringBuffer** and **StringBuilder** are described in Part II of this book.)

Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

```
String myString = "this is a test";
```

Once you have created a **String** object, you can use it anywhere that a string is allowed. For example, this statement displays **myString**:

```
System.out.println(myString);
```

Java defines one operator for **String** objects: +. It is used to concatenate two strings. For example, the statement

```
String myString = "I" + " like " + "Java.;"
```

results in **myString** containing "I like Java."

The following program demonstrates the preceding concepts:

```
// Demonstrating Strings.  
class StringDemo {  
    public static void main(String[] args) {  
        String strOb1 = "First String";  
        String strOb2 = "Second String";  
        String strOb3 = strOb1 + " and " + strOb2;  
  
        System.out.println(strOb1);  
  
        System.out.println(strOb2);  
        System.out.println(strOb3);  
    }  
}
```

The output produced by this program is shown here:

```
First String  
Second String  
First String and Second String
```

The **String** class contains several methods that you can use. Here are a few. You can test two strings for equality by using **equals()**. You can obtain the length of a string by calling the **length()** method. You can obtain the character at a specified index within a string by calling **charAt()**. The general forms of these three methods are shown here:

```
boolean equals(secondStr)
int length()
char charAt(index)
```

Here is a program that demonstrates these methods:

```
// Demonstrating some String methods.
class StringDemo2 {
    public static void main(String[] args) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1;

        System.out.println("Length of strOb1: " +
                           strOb1.length());

        System.out.println("Char at index 3 in strOb1: " +
                           strOb1.charAt(3));

        if(strOb1.equals(strOb2))
            System.out.println("strOb1 == strOb2");
        else
            System.out.println("strOb1 != strOb2");

        if(strOb1.equals(strOb3))
            System.out.println("strOb1 == strOb3");
        else
            System.out.println("strOb1 != strOb3");
    }
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example:

Here is the output from this program:

```
str[0]: one  
str[1]: two  
str[2]: three
```

As you will see in the following section, string arrays play an important part in many Java programs.

# Using Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments* to `main()`. A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a `String` array passed to the `args` parameter of `main()`. The first command-line argument is stored at `args[0]`, the second at `args[1]`, and so on. For example, the following program displays all of the command-line arguments that it is called with:

Try executing this program, as shown here:

```
java CommandLine this is a test 100 -1
```

When you do, you will see the following output:

```
args[0] : this
args[1] : is
args[2] : a
args[3] : test
args[4] : 100
args[5] : -1
```

---

**REMEMBER** All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually, as explained in [Chapter 19](#).

## Varargs: Variable-Length Arguments

Modern versions of Java include a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called *varargs*, and it is short for *variable-length arguments*. A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.

Situations that require that a variable number of arguments be passed to a method are not unusual. For example, a method that opens an Internet connection might take a user name, password, filename, protocol, and so on, but supply defaults if some of this information is not provided. In this situation, it would be convenient to pass only the arguments to which the defaults did not apply. Another example is the `printf()` method that is part of Java's I/O library. As you will see in [Chapter 22](#), it takes a variable number of arguments, which it formats and then outputs.

In the early days of Java, variable-length arguments could be handled two ways, neither of which was particularly pleasing. First, if the maximum number of arguments was small and known, then you could create overloaded versions of the method, one for each way the method could be called. Although this works and is suitable for some cases, it applies to only a narrow class of situations.

In cases where the maximum number of potential arguments was larger, or unknowable, a second approach was used in which the arguments were put into an array, and then the array was passed to the method. This approach, which you may still find in older legacy code, is illustrated by the following program:

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
    static void vaTest(int[] v) {
        System.out.print("Number of args: " + v.length +
                         " Contents: ");
        for(int x : v)
            System.out.print(x + " ");
        System.out.println();
    }

    public static void main(String[] args)
    {
        // Notice how an array must be created to
        // hold the arguments.
        int[] n1 = { 10 };
        int[] n2 = { 1, 2, 3 };
        int[] n3 = { };

        vaTest(n1); // 1 arg
        vaTest(n2); // 3 args
        vaTest(n3); // no args
    }
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

In the program, the method **vaTest()** is passed its arguments through the array **v**. This old-style approach to variable-length arguments does enable **vaTest()** to take an arbitrary number of arguments. However, it requires

that these arguments be manually packaged into an array prior to calling **vaTest()**. Not only is it tedious to construct an array each time **vaTest()** is called, it is potentially error-prone. The varargs feature offers a simpler, better option.

A variable-length argument is specified by three periods (...). For example, here is how **vaTest()** is written using a vararg:

```
static void vaTest(int ... v) {
```

This syntax tells the compiler that **vaTest()** can be called with zero or more arguments. As a result, **v** is implicitly declared as an array of type **int[ ]**. Thus, inside **vaTest()**, **v** is accessed using the normal array syntax. Here is the preceding program rewritten using a vararg:

```
// Demonstrate variable-length arguments.
class VarArgs {

    // vaTest() now uses a vararg.
    static void vaTest(int ... v) {
        System.out.print("Number of args: " + v.length +
                         " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String[] args)
    {
        // Notice how vaTest() can be called with a
        // variable number of arguments.
        vaTest(10);           // 1 arg
        vaTest(1, 2, 3);     // 3 args
        vaTest();             // no args
    }
}
```

The output from the program is the same as the original version.

There are two important things to notice about this program. First, as explained, inside **vaTest( )**, **v** is operated on as an array. This is because **v** is an array. The ... syntax simply tells the compiler that a variable number of arguments will be used and that these arguments will be stored in the array referred to by **v**. Second, in **main( )**, **vaTest( )** is called with different numbers of arguments, including no arguments at all. The arguments are automatically put in an array and passed to **v**. In the case of no arguments, the length of the array is zero.

A method can have “normal” parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method. For example, this method declaration is perfectly acceptable:

```
int doIt(int a, int b, double c, int ... vals) {
```

In this case, the first three arguments used in a call to **doIt( )** are matched to the first three parameters. Then, any remaining arguments are assumed to belong to **vals**.

Remember, the varargs parameter must be last. For example, the following declaration is incorrect:

```
int doIt(int a, int b, double c, int ... vals, boolean
stopFlag) { // Error!
```

Here, there is an attempt to declare a regular parameter after the varargs parameter, which is illegal.

There is one more restriction to be aware of: there must be only one varargs parameter. For example, this declaration is also invalid:

```
int doIt(int a, int b, double c, int ... vals, double ...
morevals) { // Error!
```

The attempt to declare the second varargs parameter is illegal.

Here is a reworked version of the **vaTest( )** method that takes a regular argument and a variable-length argument:

```
// Use varargs with standard arguments.
class VarArgs2 {

    // Here, msg is a normal parameter and v is a
    // varargs parameter.
    static void vaTest(String msg, int ... v) {
        System.out.print(msg + v.length +
                          " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String[] args)
    {
        vaTest("One vararg: ", 10);
        vaTest("Three varargs: ", 1, 2, 3);
        vaTest("No varargs: ");
    }
}
```

The output from this program is shown here:

```
One vararg: 1 Contents: 10
Three varargs: 3 Contents: 1 2 3
No varargs: 0 Contents:
```

## Overloading Vararg Methods

You can overload a method that takes a variable-length argument. For example, the following program overloads `vaTest()` three times:

```
// Varargs and overloading.
class VarArgs3 {

    static void vaTest(int ... v) {
        System.out.print("vaTest(int ...): " +
                        "Number of args: " + v.length +
                        " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    static void vaTest(boolean ... v) {
        System.out.print("vaTest(boolean ...): " +
                        "Number of args: " + v.length +
                        " Contents: ");

        for(boolean x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    static void vaTest(String msg, int ... v) {
        System.out.print("vaTest(String, int ...): " +
                        msg + v.length +
                        " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String[] args)
    {
        vaTest(1, 2, 3);
        vaTest("Testing: ", 10, 20);
        vaTest(true, false, false);
    }
}
```

The output produced by this program is shown here:

```
vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...): Number of args: 3 Contents: true false false
```

This program illustrates both ways that a varargs method can be overloaded. First, the types of its vararg parameter can differ. This is the case for **vaTest(int ...)** and **vaTest(boolean ...)**. Remember, the ... causes the parameter to be treated as an array of the specified type. Therefore, just as you can overload methods by using different types of array parameters, you can overload vararg methods by using different types of varargs. In this case, Java uses the type difference to determine which overloaded method to call.

The second way to overload a varargs method is to add one or more normal parameters. This is what was done with **vaTest(String, int ...)**. In this case, Java uses both the number of arguments and the type of the arguments to determine which method to call.

---

**NOTE** A varargs method can also be overloaded by a non-varargs method. For example, **vaTest(int x)** is a valid overload of **vaTest()** in the foregoing program. This version is invoked only when one **int** argument is present. When two or more **int** arguments are passed, the varargs version **vaTest (int...v)** is used.

## Varargs and Ambiguity

Somewhat unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method. For example, consider the following program:

```
// Varargs, overloading, and ambiguity.
//
// This program contains an error and will
// not compile!
class VarArgs4 {

    static void vaTest(int ... v) {
        System.out.print("vaTest(int ...): " +
                         "Number of args: " + v.length +
                         " Contents: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    static void vaTest(boolean ... v) {
        System.out.print("vaTest(boolean ...): " +
                         "Number of args: " + v.length +
                         " Contents: ");

        for(boolean x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String[] args)
    {
        vaTest(1, 2, 3); // OK
        vaTest(true, false, false); // OK

        vaTest(); // Error: Ambiguous!
    }
}
```

In this program, the overloading of `vaTest()` is perfectly correct. However, this program will not compile because of the following call:

```
vaTest(); // Error: Ambiguous!
```

Because the vararg parameter can be empty, this call could be translated into a call to **vaTest(int ...)** or **vaTest(boolean ...)**. Both are equally valid. Thus, the call is inherently ambiguous.

Here is another example of ambiguity. The following overloaded versions of **vaTest()** are inherently ambiguous even though one takes a normal parameter:

```
static void vaTest(int ... v) { // ...
static void vaTest(int n, int ... v) { // ...
```

Although the parameter lists of **vaTest()** differ, there is no way for the compiler to resolve the following call:

```
vaTest(1)
```

Does this translate into a call to **vaTest(int ...)**, with one varargs argument, or into a call to **vaTest(int, int ...)** with no varargs arguments? There is no way for the compiler to answer this question. Thus, the situation is ambiguous.

Because of ambiguity errors like those just shown, sometimes you will need to forego overloading and simply use two different method names. Also, in some cases, ambiguity errors expose a conceptual flaw in your code, which you can remedy by more carefully crafting a solution.

## Local Variable Type Inference with Reference Types

As you saw in [Chapter 3](#), beginning with JDK 10, Java supports local variable type inference. Recall that when using local variable type inference, the type of the variable is specified as **var** and the variable must be initialized. Earlier examples have shown type inference with primitive types, but it can also be used with reference types. In fact, type inference with reference types constitutes a primary use. Here is a simple example that declares a **String** variable called **myStr**:

```
var myStr = "This is a string";
```

Because a quoted string is used as an initializer, the type **String** is inferred.

As explained in [Chapter 3](#), one of the benefits of local variable type inference is its ability to streamline code, and it is with reference types where such streamlining is most apparent. The reason for this is that many class types in Java have rather long names. For example, in [Chapter 13](#), you will learn about the **FileInputStream** class, which is used to open a file for input operations. In the past, you would declare and initialize a **FileInputStream** using a traditional declaration like the one shown here:

```
FileInputStream fin = new FileInputStream("test.txt");
```

With the use of **var**, it can now be written like this:

```
var fin = new FileInputStream("test.txt");
```

Here, **fin** is inferred to be of type **FileInputStream** because that is the type of its initializer. There is no need to explicitly repeat the type name. As a result, this declaration of **fin** is substantially shorter than writing it the traditional way. Thus, the use of **var** streamlines the declaration. This benefit becomes even more apparent in more complex declarations, such as those involving generics. In general, the streamlining attribute of local variable type inference helps lessen the tedium of entering long type names into your program.

Of course, the streamlining aspect of local variable type inference must be used carefully to avoid reducing the readability of your program and, thus, obscuring its meaning. For example, consider a declaration such as the one shown here:

```
var x = o.getNext();
```

In this case, it may not be immediately clear to someone reading your code what the type of **x** is. In essence, local variable type inference is a feature that you should use wisely.

As you would expect, you can also use local variable type inference with user-defined classes, as the following program illustrates. It creates a class called **MyClass** and then uses local variable type inference to declare and initialize an object of that class.

```
// Local variable type inference with a user-defined class type.
class MyClass {
    private int i;

    MyClass(int k) { i = k; }

    int geti() { return i; }
    void seti(int k) { if(k >= 0) i = k; }
}

class RefVarDemo {
    public static void main(String[] args) {
        var mc = new MyClass(10); // Notice the use of var here.

        System.out.println("Value of i in mc is " + mc.geti());
        mc.seti(19);
        System.out.println("Value of i in mc is now " + mc.geti());
    }
}
```

The output of the program is shown here:

```
Value of i in mc is 10
Value of i in mc is now 19
```

In the program, pay special attention to this line:

```
var mc = new MyClass(10); // Notice the use of var here.
```

Here, the type of **mc** will be inferred as **MyClass** because that is the type of the initializer, which is a new **MyClass** object.

As explained earlier in this book, for the benefit of readers working in Java environments that do not support local variable type inference, it will not be used by most examples in the remainder of this edition of this book. This way, the majority of examples will compile and run for the largest number of readers.

# CHAPTER

## 8

# Inheritance

---

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

## Inheritance Basics

In order to have your class inherit from a superclass, you simply incorporate the definition of the superclass into your class using the **extends** keyword. To see how, let's begin with a short example. The following program creates a superclass called **A** and a subclass called **B**. Notice how the keyword **extends** is used to create a subclass of **A**.

```
// A simple example of inheritance.

// Create a superclass.
class A {
    int i, j;

    void showij() {
        System.out.println("i and j: " + i + " " + j);
    }
}

// Create a subclass by extending class A.
class B extends A {
    int k;

    void showk() {
        System.out.println("k: " + k);
    }
}
```

```
void sum() {
    System.out.println("i+j+k: " + (i+j+k));
}
}

class SimpleInheritance {
    public static void main(String[] args) {
        A superOb = new A();
        B subOb = new B();

        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();

        /* The subclass has access to all public members of
           its superclass. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();

        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
    }
}
```

The output from this program is shown here:

```
Contents of superOb:  
i and j: 10 20
```

```
Contents of subOb:  
i and j: 7 8  
k: 9
```

```
Sum of i, j and k in subOb:  
i+j+k: 24
```

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij()**. Also, inside **sum()**, **i** and **j** can be referred to directly, as if they were part of **B**.

Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

The general form of a **class** declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {  
    // body of class  
}
```

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

## Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain
private to their class.

This program contains an error and will not
compile.
*/
// Create a superclass.
class A {
    int i; // default access
    private int j; // private to A

    void setij(int x, int y) {
        i = x;
        j = y;
    }
}

// A's j is not accessible here.
class B extends A {
    int total;

    void sum() {
        total = i + j; // ERROR, j is not accessible here
    }
}

class Access {
    public static void main(String[] args) {
        B subOb = new B();

        subOb.setij(10, 12);

        subOb.sum();
        System.out.println("Total is " + subOb.total);
    }
}
```

This program will not compile because the use of **j** inside the **sum()** method of **B** causes an access violation. Since **j** is declared as **private**, it is

only accessible by other members of its own class. Subclasses have no access to it.

---

**REMEMBER** A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

## A More Practical Example

Let's look at a more practical example that will help illustrate the power of inheritance. Here, the final version of the **Box** class developed in the preceding chapter will be extended to include a fourth component called **weight**. Thus, the new class will contain a box's width, height, depth, and weight.

```
// This program uses inheritance to extend Box.
class Box {
    double width;
    double height;
    double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

// Here, Box is extended to include weight.
class BoxWeight extends Box {
```

```

        double weight; // weight of box

        // constructor for BoxWeight
        BoxWeight(double w, double h, double d, double m) {
            width = w;
            height = h;
            depth = d;
            weight = m;
        }
    }

    class DemoBoxWeight {
        public static void main(String[] args) {
            BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
            BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
            double vol;

            vol = mybox1.volume();
            System.out.println("Volume of mybox1 is " + vol);
            System.out.println("Weight of mybox1 is " + mybox1.weight);
            System.out.println();

            vol = mybox2.volume();
            System.out.println("Volume of mybox2 is " + vol);
            System.out.println("Weight of mybox2 is " + mybox2.weight);
        }
    }
}

```

The output from this program is shown here:

```

Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

```

```

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076

```

**BoxWeight** inherits all of the characteristics of **Box** and adds to them the **weight** component. It is not necessary for **BoxWeight** to re-create all of the features found in **Box**. It can simply extend **Box** to meet its own purposes.

A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification. For example, the following class inherits **Box** and adds a color attribute:

```
// Here, Box is extended to include color.
class ColorBox extends Box {
    int color; // color of box

    ColorBox(double w, double h, double d, int c) {
        width = w;
        height = h;
        depth = d;
        color = c;
    }
}
```

Remember, once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes. Each subclass simply adds its own unique attributes. This is the essence of inheritance.

## A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations. For example, consider the following:

```

class RefDemo {
    public static void main(String[] args) {
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
        Box plainbox = new Box();
        double vol;

        vol = weightbox.volume();
        System.out.println("Volume of weightbox is " + vol);
        System.out.println("Weight of weightbox is " +
                           weightbox.weight);
        System.out.println();

        // assign BoxWeight reference to Box reference
        plainbox = weightbox;

        vol = plainbox.volume(); // OK, volume() defined in Box
        System.out.println("Volume of plainbox is " + vol);

        /* The following statement is invalid because plainbox
           does not define a weight member. */
        // System.out.println("Weight of plainbox is " + plainbox.weight);
    }
}

```

Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects. Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object.

It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is why **plainbox** can't access **weight** even when it refers to a **BoxWeight** object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out. It is not possible for a **Box** reference to access the **weight** field, because **Box** does not define one.

Although the preceding may seem a bit esoteric, it has some important practical applications—two of which are discussed later in this chapter.

## Using super

In the preceding examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box**. Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

**super** has two general forms. The first calls the superclass's constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

## Using super to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of **super**:

```
super(arg-list);
```

Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass's constructor.

To see how **super( )** is used, consider this improved version of the **BoxWeight** class:

```
// BoxWeight now uses super to initialize its Box attributes.  
class BoxWeight extends Box {  
    double weight; // weight of box  
  
    // initialize width, height, and depth using super()  
    BoxWeight(double w, double h, double d, double m) {  
        super(w, h, d); // call superclass constructor  
        weight = m;  
    }  
}
```

Here, **BoxWeight( )** calls **super( )** with the arguments **w**, **h**, and **d**. This causes the **Box** constructor to be called, which initializes **width**, **height**, and **depth** using these values. **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.

In the preceding example, **super( )** was called with three arguments. Since constructors can be overloaded, **super( )** can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments. For example, here is a complete implementation of **BoxWeight** that provides constructors for the various ways that a box can be constructed. In each case, **super( )** is called using the appropriate arguments. Notice that **width**, **height**, and **depth** have been made private within **Box**.

```
// A complete implementation of BoxWeight.
class Box {
    private double width;
    private double height;
    private double depth;

    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
    double weight; // weight of box
```

```
super(w, h, d); // call superclass constructor
weight = m;
}

// default constructor
BoxWeight() {
    super();
    weight = -1;
}

// constructor used when cube is created
BoxWeight(double len, double m) {
    super(len);
    weight = m;
}
}

class DemoSuper {
    public static void main(String[] args) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        BoxWeight mybox3 = new BoxWeight(); // default
        BoxWeight mycube = new BoxWeight(3, 2);
        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();

        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
        System.out.println();

        vol = mybox3.volume();
        System.out.println("Volume of mybox3 is " + vol);
        System.out.println("Weight of mybox3 is " + mybox3.weight);
        System.out.println();

        vol = myclone.volume();
    }
}
```

```

This program generates the following output:
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0      mycube is " + vol);
Weight of mybox2 is 0.076    mycube is " + mycube.weight);

} Volume of mybox3 is -1.0
} Weight of mybox3 is -1.0

Volume of myclone is 3000.0
Weight of myclone is 34.3

Volume of mycube is 27.0
Weight of mycube is 2.0

```

Pay special attention to this constructor in **BoxWeight**:

```

// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
}

```

Notice that **super()** is passed an object of type **BoxWeight**—not of type **Box**. This still invokes the constructor **Box(Box ob)**. As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. Of course, **Box** only has knowledge of its own members.

Let's review the key concepts behind **super()**. When a subclass calls **super()**, it is calling the constructor of its immediate superclass. Thus, **super()** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, **super()** must always be the first statement executed inside a subclass constructor.

## A Second Use for **super**

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

*super.member*

Here, *member* can be either a method or an instance variable.

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
// Using super to overcome name hiding.  
class A {  
    int i;  
}  
  
// Create a subclass by extending class A.  
  
class B extends A {  
    int i; // this i hides the i in A  
  
    B(int a, int b) {  
        super.i = a; // i in A  
        i = b; // i in B  
    }  
  
    void show() {  
        System.out.println("i in superclass: " + super.i);  
        System.out.println("i in subclass: " + i);  
    }  
}  
  
class UseSuper {  
    public static void main(String[] args) {  
        B subOb = new B(1, 2);  
  
        subOb.show();  
    }  
}
```

This program displays the following:

```
i in superclass: 1  
i in subclass: 2
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

## Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and it adds a field called **cost**, which holds the cost of shipping such a parcel.

```
// Extend BoxWeight to include shipping costs.  
  
// Start with Box.  
class Box {  
    private double width;  
    private double height;  
    private double depth;
```

```
// construct clone of an object
Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
}

// constructor used when all dimensions specified
Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
}

// constructor used when no dimensions specified
Box() {
    width = -1; // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1; // box
}

// constructor used when cube is created
Box(double len) {
    width = height = depth = len;
}

// compute and return volume
double volume() {
    return width * height * depth;
}

// Add weight.
class BoxWeight extends Box {
    double weight; // weight of box

    // construct clone of an object
    BoxWeight(BoxWeight ob) { // pass object to constructor
        super(ob);
        weight = ob.weight;
    }
}
```

```
// constructor used when cube is created
BoxWeight(double len, double m) {
    super(len);
    weight = m;
}
}

// Add shipping costs.
class Shipment extends BoxWeight {
    double cost;

    // construct clone of an object
    Shipment(Shipment ob) { // pass object to constructor
        super(ob);
        cost = ob.cost;
    }

    // constructor when all parameters are specified
    Shipment(double w, double h, double d,
             double m, double c) {
        super(w, h, d, m); // call superclass constructor
        cost = c;
    }

    // default constructor
    Shipment() {
        super();
        cost = -1;
    }

    // constructor used when cube is created
    Shipment(double len, double m, double c) {
        super(len, m);
        cost = c;
    }
}

class DemoShipment {
    public static void main(String[] args) {
        Shipment shipment1 =
            new Shipment(10, 20, 15, 10, 3.41);
        Shipment shipment2 =

```

```

        vol = shipment2.volume();
        System.out.println("Volume of shipment2 is " + vol);
        System.out.println("Weight of shipment2 is "
                           + shipment2.weight);
        System.out.println("Shipping cost: $" + shipment2.cost);
    }
}

System.out.println("Weight of shipment1 is "
                   + shipment1.weight);
System.out.println("Shipping cost: $" + shipment1.cost);
System.out.println("Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28

```

Because of inheritance, **Shipment** can make use of the previously defined classes of **Box** and **BoxWeight**, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

This example illustrates one other important point: **super()** always refers to the constructor in the closest superclass. The **super()** in **Shipment** calls the constructor in **BoxWeight**. The **super()** in **BoxWeight** calls the constructor in **Box**. In a class hierarchy, if a superclass constructor requires arguments, then all subclasses must pass those arguments “up the line.” This is true whether or not a subclass needs arguments of its own.

**NOTE** In the preceding program, the entire class hierarchy, including **Box**, **BoxWeight**, and **Shipment**, is shown all in one file. This is for your convenience only. In Java, all three classes could have been placed into their own files and compiled separately. In fact, using separate files is the norm, not the exception, in creating class hierarchies.

## When Constructors Are Executed

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed? For example, given a subclass called **B** and a superclass called **A**, is **A**’s constructor executed before **B**’s, or vice versa? The answer is that in a class hierarchy, constructors complete

their execution in order of derivation, from superclass to subclass. Further, since **super()** must be the first statement executed in a subclass's constructor, this order is the same whether or not **super()** is used. If **super()** is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are executed.

// Create a super class.
class A {
    A() {
        System.out.println("Inside A's constructor.");
    }
}

// Create a subclass by extending class A.
class B extends A {
    B() {
        System.out.println("Inside B's constructor.");
    }
}

// Create another subclass by extending B.
class C extends B {
    C() {
        System.out.println("Inside C's constructor.");
    }
}

class CallingCons {
    public static void main(String[] args) {
        C c = new C();
    }
}
```

The output from this program is shown here:

```
Inside A's constructor  
Inside B's constructor  
Inside C's constructor
```

As you can see, the constructors are executed in order of derivation.

If you think about it, it makes sense that constructors complete their execution in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

## Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called through its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

```
// Method overriding.  
class A {  
    int i, j;  
    A(int a, int b) {  
        i = a;  
        j = b;  
    }  
  
    // display i and j  
    void show() {  
        System.out.println("i and j: " + i + " " + j);  
    }  
}
```

```
class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // display k - this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}

class Override {
    public static void main(String[] args) {
        B subOb = new B(1, 2, 3);

        subOb.show(); // this calls show() in B
    }
}
```

The output produced by this program is shown here:

```
k: 3
```

When **show()** is invoked on an object of type **B**, the version of **show()** defined within **B** is used. That is, the version of **show()** inside **B** overrides the version declared in **A**.

If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show()** is invoked within the subclass's version. This allows all instance variables to be displayed.

```
class B extends A {  
    int k;  
  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
  
    void show() {  
        super.show(); // this calls A's show()  
        System.out.println("k: " + k);  
    }  
}
```

If you substitute this version of **A** into the previous program, you will see the following output:

```
i and j: 1 2  
k: 3
```

Here, **super.show()** calls the superclass version of **show()**.

Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

```
// Methods with differing type signatures are overloaded - not
// overridden.
class A {
    int i, j;

    A(int a, int b) {
        i = a;
        j = b;
    }

    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}

// Create a subclass by extending class A.
class B extends A {
    int k;

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // overload show()
    void show(String msg) {
        System.out.println(msg + k);
    }
}

class Override {
    public static void main(String[] args) {
        B subOb = new B(1, 2, 3);

        subOb.show("This is k: "); // this calls show() in B
        subOb.show(); // this calls show() in A
    }
}
```

The output produced by this program is shown here:

```
This is k: 3  
i and j: 1 2
```

The version of **show( )** in **B** takes a string parameter. This makes its type signature different from the one in **A**, which takes no parameters. Therefore, no overriding (or name hiding) takes place. Instead, the version of **show( )** in **B** simply overloads the version of **show( )** in **A**.

## Dynamic Method Dispatch

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
    void callme() {
        System.out.println("Inside A's callme method");
    }
}

class B extends A {
    // override callme()
    void callme() {
        System.out.println("Inside B's callme method");
    }
}

class C extends A {
    // override callme()
    void callme() {
        System.out.println("Inside C's callme method");
    }
}

class Dispatch {
    public static void main(String[] args) {
        A a = new A(); // object of type A
        B b = new B(); // object of type B
        C c = new C(); // object of type C

        A r; // obtain a reference of type A

        r = a; // r refers to an A object
        r.callme(); // calls A's version of callme

        r = b; // r refers to a B object
        r.callme(); // calls B's version of callme

        r = c; // r refers to a C object
        r.callme(); // calls C's version of callme
    }
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**. Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme( )**. As the output shows, the version of **callme( )** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme( )** method.

---

**NOTE** Readers familiar with C++ or C# will recognize that overridden methods in Java are similar to virtual functions in those languages.

## Why Overridden Methods?

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy that moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on

instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

## Applying Method Overriding

Let's look at a more practical example that uses method overriding. The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object. It also defines a method called **area()** that computes the area of an object. The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**. Each of these subclasses overrides **area()** so that it returns the area of a rectangle and a triangle, respectively.

```
// Using run-time polymorphism.
class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    double area() {
        System.out.println("Area for Figure is undefined.");
        return 0;
    }
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}

class FindAreas {
    public static void main(String[] args) {
```

```
Triangle t = new Triangle(10, 8);
Figure figref;

figref = r;
System.out.println("Area is " + figref.area());

figref = t;
System.out.println("Area is " + figref.area());

figref = f;
System.out.println("Area is " + figref.area());
}
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from **Figure**, then its area can be obtained by calling **area()**. The interface to this operation is the same no matter what type of figure is being used.

## Using Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable

to create a meaningful implementation for a method. This is the case with the class **Figure** used in the preceding example. The definition of **area( )** is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class **Triangle**. It has no meaning if **area( )** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the *abstract method*.

You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass or be declared **abstract** itself.

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
// A Simple demonstration of abstract.  
abstract class A {  
    abstract void callme();  
  
    // concrete methods are still allowed in abstract classes  
    void callmetoo() {  
        System.out.println("This is a concrete method.");  
    }  
}  
  
class B extends A {  
    void callme() {  
        System.out.println("B's implementation of callme.");  
    }  
}  
  
class AbstractDemo {  
    public static void main(String[] args) {  
        B b = new B();  
  
        b.callme();  
        b.callmetoo();  
    }  
}
```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo()**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example.

Using an abstract class, you can improve the **Figure** class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares **area()** as

abstract inside **Figure**. This, of course, means that all classes derived from **Figure** must override **area()**.

```
// Using abstract methods and classes.
abstract class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    // area is now an abstract method
    abstract double area();
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}

class AbstractAreas {
    public static void main(String[] args) {
        // Figure f = new Figure(10, 10); // illegal now
        Rectangle r = new Rectangle(10, 10).
```

```
    figref = t;
    System.out.println("Area is " + figref.area());ated
}
}
figref = r;
System.out.println("Area is " + figref.area());
```

As the comment inside **main()** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override **area()**. To prove this to yourself, try creating a subclass that does not override **area()**. You will receive a compile-time error.

Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**. The variable **figref** is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

## Using final with Inheritance

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

## Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {
    final void meth() {
        System.out.println("This is a final method.");
    }
}

class B extends A {
    void meth() { // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

Because **meth()** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it “knows” they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding*.

## Using **final** to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A {  
    //...  
}  
  
// The following class is illegal.  
class B extends A { // ERROR! Can't subclass A  
    //...  
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

---

**NOTE** Beginning with JDK 17, the ability to *seal* a class was added to Java. Sealing offers fine-grained control over inheritance. Sealing is described in [Chapter 17](#).

## Local Variable Type Inference and Inheritance

As explained in [Chapter 3](#), JDK 10 added local variable type inference to the Java language, which is supported by the context-sensitive keyword **var**. It is important to have a clear understanding of how type inference works within an inheritance hierarchy. Recall that a superclass reference can refer to a derived class object, and this feature is part of Java's support for polymorphism. However, it is critical to remember that when using local variable type inference, the inferred type of a variable is based on the declared type of its initializer. Therefore, if the initializer is of the superclass type, that will be the inferred type of the variable. It does not matter if the actual object being referred to by the initializer is an instance of a derived class. For example, consider this program:

```
// When working with inheritance, the inferred type is the declared
// type of the initializer, which may not be the most derived type of
// the object being referred to by the initializer.

class MyClass {
    // ...
}

class FirstDerivedClass extends MyClass {
    int x;
    // ...
}
```

```
class SecondDerivedClass extends FirstDerivedClass {
    int y;
    // ...
}

class TypeInferenceAndInheritance {

    // Return some type of MyClass object.
    static MyClass getObj(int which) {
        switch(which) {
            case 0: return new MyClass();
            case 1: return new FirstDerivedClass();
            default: return new SecondDerivedClass();
        }
    }

    public static void main(String[] args) {

        // Even though getObj() returns different types of
        // objects within the MyClass inheritance hierarchy,
        // its declared return type is MyClass. As a result,
        // in all three cases shown here, the type of the
        // variables is inferred to be MyClass, even though
        // different derived types of objects are obtained.

        // Here, getObj() returns a MyClass object.
        var mc = getObj(0);

        // In this case, a FirstDerivedClass object is returned.
        var mc2 = getObj(1);

        // Here, a SecondDerivedClass object is returned.
        var mc3 = getObj(2);

        // Because the types of both mc2 and mc3 are inferred
        // as MyClass (because the return type of getObj() is
        // MyClass), neither mc2 nor mc3 can access the fields
        // declared by FirstDerivedClass or SecondDerivedClass.
        // mc2.x = 10; // Wrong! MyClass does not have an x field.
        // mc3.y = 10; // Wrong! MyClass does not have a y field.
    }
}
```

In the program, a hierarchy is created that consists of three classes, at the top of which is **MyClass**. **FirstDerivedClass** is a subclass of **MyClass**, and **SecondDerivedClass** is a subclass of **FirstDerivedClass**. The program then uses type inference to create three variables, called **mc**, **mc2**, and **mc3**, by calling **getObj()**. The **getObj()** method has a return type of **MyClass** (the superclass) but returns objects of type **MyClass**, **FirstDerivedClass**, or **SecondDerivedClass**, depending on the argument that it is passed. As the output shows, the inferred type is determined by the return type of **getObj()**, not by the actual type of the object obtained. Thus, all three variables will be of type **MyClass**.

## The Object Class

There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**. That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

**Object** defines the following methods, which means that they are available in every object.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object <i>object</i> )	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled. (Deprecated for removal by JDK 18.)
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long <i>milliseconds</i> ) void wait(long <i>milliseconds</i> , int <i>nanoseconds</i> )	Waits on another thread of execution.

The methods **getClass( )**, **notify( )**, **notifyAll( )**, and **wait( )** are declared as **final**. You may override the others. These methods are described elsewhere in this book. However, notice two methods now: **equals( )** and **toString( )**. The **equals( )** method compares two objects. It returns **true** if the objects are equal, and **false** otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The **toString( )** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using **println( )**. Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

One last point: Notice the unusual syntax in the return type for **getClass( )**. This relates to Java's *generics* feature, which is described in [Chapter 14](#).

# CHAPTER

## 9

# Packages and Interfaces

---

This chapter examines two of Java’s most innovative features: packages and interfaces. *Packages* are containers for classes. They are used to keep the class name space compartmentalized. For example, a package allows you to create a class named **List**, which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions. As you will see in [Chapter 16](#), packages also play an important role with modules.

In previous chapters, you have seen how methods define the interface to the data in a class. Through the use of the **interface** keyword, Java allows you to fully abstract an interface from its implementation. Using **interface**, you can specify a set of methods that can be implemented by one or more classes. In its traditional form, the **interface** itself does not actually define any implementation. Although they are similar to abstract classes, **interfaces** have an additional capability: A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

## Packages

In the preceding chapters, the name of each sample class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the

name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name “Foobar” as a class name. Or, imagine the entire Internet community arguing over who first named a class “Espresso.”) Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are exposed only to other members of the same package. This allows your classes to have intimate knowledge of each other but not expose that knowledge to the rest of the world.

## Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven’t had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the **package** statement:

```
package pkg;
```

Here, *pkg* is the name of the package. For example, the following statement creates a package called **mypackage**:

```
package mypackage;
```

Typically, Java uses file system directories to store packages, and that is the approach assumed by the examples in this book. For example, the **.class** files for any classes you declare to be part of **mypackage** must be stored in a directory called **mypackage**. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[pkg2[pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package a.b.c;
```

needs to be stored in **a\b\c** in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

## Finding Packages and CLASSPATH

As just explained, packages are typically mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? As it relates to the examples in this chapter, the answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes. It is useful to point out that, beginning with JDK 9, a package can be part of a module, and thus found on the module path. However, a discussion of modules and module paths is deferred until [Chapter 16](#). For now, we will use only class paths.

For example, consider the following package specification:

```
package mypack;
```

In order for a program to find **mypad**, the program can be executed from a directory immediately above **mypad**, or the **CLASSPATH** must be set to include the path to **mypad**, or the **-classpath** option must specify the path to **mypad** when the program is run via **java**.

When the last two options are used, the class path *must not* include **mypad** itself. It must simply specify the *path to mypad*. For example, in a Windows environment, if the path to **mypad** is

C:\MyPrograms\Java\mypad

then the class path to **mypad** is

C:\MyPrograms\Java

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

## A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package mypack;

class Balance {
    String name;
    double bal;

    Balance(String n, double b) {
        name = n;
        bal = b;
    }

    void show() {
        if(bal<0)
            System.out.print("-> ");
        System.out.println(name + ": $" + bal);
    }
}

class AccountBalance {
    public static void main(String[] args) {
        Balance[] current = new Balance[3];

        current[0] = new Balance("K. J. Fielding", 123.23);
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);

        for(int i=0; i<3; i++) current[i].show();
    }
}
```

Call this file **AccountBalance.java** and put it in a directory called **mypack**.

Next, compile the file. Make sure that the resulting **.class** file is also in the **mypack** directory. Then, try executing the **AccountBalance** class, using the following command line:

```
java mypack.AccountBalance
```

Remember, you will need to be in the directory above **mypack** when you execute this command. (Alternatively, you can use one of the other two

options described in the preceding section to specify the path **mypack**.)

As explained, **AccountBalance** is now part of the package **mypack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

**AccountBalance** must be qualified with its package name.

## Packages and Member Access

In the preceding chapters, you learned about various aspects of Java's access control mechanism and its access modifiers. For example, you already know that access to a **private** member of a class is granted only to other members of that class. Packages add another dimension to access control. As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. As it relates to the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. [Table 9-1](#) sums up the interactions.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

**Table 9-1** Class Member Access

While Java’s access control mechanism may seem complicated, we can simplify it as follows. Anything declared **public** can be accessed from different classes and different packages. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

**Table 9-1** applies only to members of classes. A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible outside its package. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

---

**NOTE** The modules feature can also affect accessibility. Modules are described in [Chapter 16](#).

## An Access Example

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, **p1** and **p2**.

The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**. The first class defines four **int** variables in each of the legal protection modes. The variable **n** is declared with the

default protection, **n\_pri** is **private**, **n\_pro** is **protected**, and **n\_pub** is **public**.

Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.

The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**. This grants **Derived** access to every variable in **Protection** except for **n\_pri**, the **private** one. The third class, **SamePackage**, is not a subclass of **Protection**, but it is in the same package and also has access to all but **n\_pri**.

This is file **Protection.java**:

```
package p1;

public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;

    public Protection() {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

This is file **Derived.java**:

```
package p1;

class Derived extends Protection {
    Derived() {
        System.out.println("derived constructor");
        System.out.println("n = " + n);

        // class only
        // System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

This is file **SamePackage.java**:

```
package p1;

class SamePackage {
    SamePackage() {

        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);

        // class only
        // System.out.println("n_pri = " + p.n_pri);

        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

Following is the source code for the other package, **p2**. The two classes defined in **p2** cover the other two conditions that are affected by access control. The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**'s variables except for **n\_pri** (because it is **private**) and **n**, the variable declared with the default protection. Remember, the default only allows access from within the class or the

package, not extra-package subclasses. Finally, the class **OtherPackage** has access to only one variable, **n\_pub**, which was declared **public**.

This is file **Protection2.java**:

```
package p2;

class Protection2 extends p1.Protection {
    Protection2() {
        System.out.println("derived other package constructor");

        // class or package only
        // System.out.println("n = " + n);

        // class only
        // System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

This is file **OtherPackage.java**:

```
package p2;

class OtherPackage {
    OtherPackage() {
        p1.Protection p = new p1.Protection();
        System.out.println("other package constructor");

        // class or package only
        // System.out.println("n = " + p.n);

        // class only
        // System.out.println("n_pri = " + p.n_pri);

        // class, subclass or package only
        // System.out.println("n_pro = " + p.n_pro);

        System.out.println("n_pub = " + p.n_pub);
    }
}
```

If you want to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
    public static void main(String[] args) {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
    }
}
```

The test file for **p2** is shown next:

```
// Demo package p2.  
package p2;  
  
// Instantiate the various classes in p2.  
public class Demo {  
    public static void main(String[] args) {  
        Protection2 ob1 = new Protection2();  
        OtherPackage ob2 = new OtherPackage();  
    }  
}
```

## Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

```
import pkg1 [pkg2].(classname | *);
```

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (\*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

```
import java.util.Date;  
import java.io.*;
```

All of the standard Java SE classes included with Java begin with the name **java**. The basic language functions are stored in a package called **java.lang**. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

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

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an **import** statement:

```
import java.util.*;  
class MyDate extends Date {  
}
```

The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {  
}
```

In this version, **Date** is fully qualified.

As shown in [Table 9-1](#), when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code. For example, if you want the **Balance** class of the package **mypad** shown earlier to be available as a stand-alone class for general use outside of **mypad**, then you will need to declare it as **public** and put it into its own file, as shown here:

```
package mypack;

/* Now, the Balance class, its constructor, and its
   show() method are public. This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
    String name;
    double bal;

    public Balance(String n, double b) {
        name = n;
        bal = b;
    }

    public void show() {
        if(bal<0)

            System.out.print("--> ");
            System.out.println(name + ": $" + bal);
    }
}
```

As you can see, the **Balance** class is now **public**. Also, its constructor and its **show()** method are **public**, too. This means that they can be accessed by any type of code outside the **mypack** package. For example, here **TestBalance** imports **mypack** and is then able to make use of the **Balance** class:

```
import mypack.*;

class TestBalance {
    public static void main(String[] args) {

        /* Because Balance is public, you may use Balance
           class and call its constructor. */
        Balance test = new Balance("J. J. Jaspers", 99.88);

        test.show(); // you may also call show()
    }
}
```

As an experiment, remove the **public** specifier from the **Balance** class and then try compiling **TestBalance**. As explained, errors will result.

## Interfaces

Using the keyword **interface**, you can fully abstract a class's interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

To implement an interface, a class must provide the complete set of methods required by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

## Defining an Interface

An interface is defined much like a class. This is a simplified general form of an interface:

```
access interface name {  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
  
    type final-varname1 = value;  
    type final-varname2 = value;  
    //...  
    return-type method-nameN(parameter-list);  
    type final-varnameN = value;  
}
```

When no access modifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by code outside its package. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. *name* is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods. Each class that includes such an interface must implement all of the methods.

Before continuing, an important point needs to be made. JDK 8 added a feature to **interface** that made a significant change to its capabilities. Prior to JDK 8, an interface could not define any implementation whatsoever. This is the type of interface that the preceding simplified form shows, in which no method declaration supplies a body. Thus, prior to JDK 8, an interface could define only “what,” but not “how.” JDK 8 changed this. Beginning with JDK 8, it is possible to add a *default implementation* to an interface method. Furthermore, JDK 8 also added static interface methods, and beginning with JDK 9, an interface can include private methods. Thus, it is now possible for **interface** to specify some behavior. However, such methods constitute what are, in essence, special-use features, and the original intent behind **interface** still remains. Therefore, as a general rule, you will still often create and use interfaces in which no use is made of these new features. For this reason, we will begin by discussing the

interface in its traditional form. The newer interface features are described at the end of this chapter.

As the general form shows, variables can be declared inside interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

Here is an example of an interface definition. It declares a simple interface that contains one method called **callback( )** that takes a single integer parameter.

```
interface Callback {  
    void callback(int param);  
}
```

## Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods required by the interface. The general form of a class that includes the **implements** clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]] {  
    // class-body  
}
```

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is a small sample class that implements the **Callback** interface shown earlier:

```
class Client implements Callback {  
    // Implement Callback's interface  
    public void callback(int p) {  
  
        System.out.println("callback called with " + p);  
    }  
}
```

Notice that **callback( )** is declared using the **public** access modifier.

---

**REMEMBER** When you implement an interface method, it must be declared as **public**.

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **Client** implements **callback( )** and adds the method **nonIfaceMeth( )**:

```
class Client implements Callback {  
    // Implement Callback's interface  
    public void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
  
    void nonIfaceMeth() {  
        System.out.println("Classes that implement interfaces " +  
            "may also define other members, too.");  
    }  
}
```

## Accessing Implementations Through Interface References

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code that calls methods on them. The calling code can dispatch through an interface without having to know anything about the “callee.” This process is similar

to using a superclass reference to access a subclass object, as described in [Chapter 8](#).

The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface {
    public static void main(String[] args) {
        Callback c = new Client();
        c.callback(42);
    }
}
```

The output of this program is shown here:

```
callback called with 42
```

Notice that variable **c** is declared to be of the interface type **Callback**, yet it was assigned an instance of **Client**. Although **c** can be used to access the **callback( )** method, it cannot access any other members of the **Client** class. An interface reference variable has knowledge only of the methods declared by its **interface** declaration. Thus, **c** could not be used to access **nonIfaceMeth( )** since it is defined by **Client** but not **Callback**.

While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of **Callback**, shown here:

```
// Another implementation of Callback.
class AnotherClient implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
        System.out.println("Another version of callback");
        System.out.println("p squared is " + (p*p));
    }
}
```

Now, try the following class:

```
class TestIface2 {  
    public static void main(String[] args) {  
        Callback c = new Client();  
        AnotherClient ob = new AnotherClient();  
  
        c.callback(42);  
  
        c = ob; // c now refers to AnotherClient object  
        c.callback(42);  
    }  
}
```

The output from this program is shown here:

```
callback called with 42  
Another version of callback  
p squared is 1764
```

As you can see, the version of **callback()** that is called is determined by the type of object that **c** refers to at run time. While this is a very simple example, you will see another, more practical one shortly.

## Partial Implementations

If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as **abstract**. For example:

```
abstract class Incomplete implements Callback {  
    int a, b;  
  
    void show() {  
        System.out.println(a + " " + b);  
    }  
    //...  
}
```

Here, the class **Incomplete** does not implement **callback()** and must be declared as **abstract**. Any class that inherits **Incomplete** must implement **callback()** or be declared **abstract** itself.

## Nested Interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*. A nested interface can be declared as **public**, **private**, or **protected**. This differs from a top-level interface, which must either be declared as **public** or use the default access level, as previously described. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Here is an example that demonstrates a nested interface:

```
// A nested interface example.

// This class contains a member interface.
class A {
    // this is a nested interface
    public interface NestedIF {
        boolean isNotNegative(int x);
    }
}
```

```

// B implements the nested interface.
class B implements A.NestedIF {
    public boolean isNotNegative(int x) {
        return x < 0 ? false: true;
    }
}

class NestedIFDemo {
    public static void main(String[] args) {

        // use a nested interface reference
        A.NestedIF nif = new B();

        if(nif.isNotNegative(10))
            System.out.println("10 is not negative");
        if(nif.isNotNegative(-12))
            System.out.println("this won't be displayed");
    }
}

```

Notice that **A** defines a member interface called **NestedIF** and that it is declared **public**. Next, **B** implements the nested interface by specifying

implements A.NestedIF

Notice that the name is fully qualified by the enclosing class's name. Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**, this is legal.

## Applying Interfaces

To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called **Stack** that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be "growable." The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push( )** and **pop( )** define

the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**. This interface will be used by both stack implementations.

```
// Define an integer stack interface.  
interface IntStack {  
    void push(int item); // store an item  
    int pop(); // retrieve an item  
}
```

The following program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
    private int[] stck;
    private int tos;

    // allocate and initialize stack
    FixedStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    public void push(int item) {
        if(tos==stck.length-1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    public int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class IFTest {
    public static void main(String[] args) {
        FixedStack mystack1 = new FixedStack(5);
        FixedStack mystack2 = new FixedStack(8);

        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
    }
}
```

Following is another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```
System.out.println(mystack1.pop());  
System.out.println(mystack1.pop());  
System.out.println(mystack2.pop());  
}  
}
```

```
// Implement a "growable" stack.
class DynStack implements IntStack {
    private int[] stck;
    private int tos;

    // allocate and initialize stack
    DynStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    public void push(int item) {
        // if stack is full, allocate a larger stack
        if(tos==stck.length-1) {
            int[] temp = new int[stck.length * 2]; // double size
            for(int i=0; i<stck.length; i++) temp[i] = stck[i];
            stck = temp;
            stck[++tos] = item;
        }
        else
            stck[++tos] = item;
    }

    // Pop an item from the stack
    public int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class IFTest2 {
    public static void main(String[] args) {
        DynStack mystack1 = new DynStack(5);
        DynStack mystack2 = new DynStack(8);

        // these loops cause each stack to grow
        for(int i=0; i<12; i++) mystack1.push(i);
    }
}
```

```
        System.out.println("Stack in mystack2:");
        for(int i=0; i<20; i++)
            System.out.println(mystack2.pop());
    }
}
System.out.println(mystack1.pop());
```

The following class uses both the **FixedStack** and **DynStack** implementations. It does so through an interface reference. This means that calls to **push( )** and **pop( )** are resolved at run time rather than at compile time.

```
/* Create an interface variable and
   access stacks through it.
*/
class IFTest3 {
    public static void main(String[] args) {
        IntStack mystack; // create an interface reference variable
        DynStack ds = new DynStack(5);
        FixedStack fs = new FixedStack(8);

        mystack = ds; // load dynamic stack
        // push some numbers onto the stack
        for(int i=0; i<12; i++) mystack.push(i);

        mystack = fs; // load fixed stack
        for(int i=0; i<8; i++) mystack.push(i);

        mystack = ds;
        System.out.println("Values in dynamic stack:");
        for(int i=0; i<12; i++)
            System.out.println(mystack.pop());

        mystack = fs;
        System.out.println("Values in fixed stack:");
        for(int i=0; i<8; i++)
            System.out.println(mystack.pop());
    }
}
```

In this program, **mystack** is a reference to the **IntStack** interface. Thus, when it refers to **ds**, it uses the versions of **push( )** and **pop( )** defined by the **DynStack** implementation. When it refers to **fs**, it uses the versions of **push( )** and **pop( )** defined by **FixedStack**. As explained, these determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

## Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants. If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything. It is as if that class were importing the constant fields into the class name space as **final** variables. The next example uses this technique to implement an automated “decision maker”:

```
import java.util.Random;

interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}

class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)
            return NO;                  // 30%
        else if (prob < 60)
            return YES;                // 30%
        else if (prob < 75)
            return LATER;              // 15%
        else if (prob < 98)
            return SOON;               // 13%
        else
            return NEVER;             // 2%
    }
}

class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
            case NO:
                System.out.println("No");
                break;
            case YES:
                System.out.println("Yes");
                break;
            case MAYBE:
                System.out.println("Maybe");
                break;
            case LATER:
                System.out.println("Later");
                break;
        }
    }
}
```

```

    }
}

public static void main(String[] args) {
    Question q = new Question();

    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
}
}

```

Notice that this program makes use of one of Java's standard classes:

**Random**. This class provides pseudorandom numbers. It contains several methods that allow you to obtain random numbers in the form required by your program. In this example, the method **nextDouble()** is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the two classes, **Question** and **AskMe**, both implement the **SharedConstants** interface where **NO**, **YES**, **MAYBE**, **SOON**, **LATER**, and **NEVER** are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the results are different each time it is run.

```

Later
Soon
No
Yes

```

---

**NOTE** The technique of using an interface to define shared constants, as just described, is controversial. It is described here for completeness.

## Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for

all methods required by the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
    void meth1();
    void meth2();
}

// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
    void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
    public void meth1() {
        System.out.println("Implement meth1().");
    }

    public void meth2() {
        System.out.println("Implement meth2().");
    }

    public void meth3() {
        System.out.println("Implement meth3().");
    }
}

class IFExtend {
    public static void main(String[] args) {
        MyClass ob = new MyClass();

        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```

As an experiment, you might want to try removing the implementation for **meth1( )** in **MyClass**. This will cause a compile-time error. As stated

earlier, any class that implements an interface must implement all methods required by that interface, including any that are inherited from other interfaces.

## Default Interface Methods

As explained earlier, prior to JDK 8, an interface could not define any implementation whatsoever. This meant that for all previous versions of Java, the methods specified by an interface were abstract, containing no body. This is the traditional form of an interface and is the type of interface that the preceding discussions have used. The release of JDK 8 changed this by adding a new capability to **interface** called the *default method*. A default method lets you define a default implementation for an interface method. In other words, by use of a default method, it is possible for an interface method to provide a body, rather than being abstract. During its development, the default method was also referred to as an *extension method*, and you will likely see both terms used.

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. Recall that there must be implementations for all methods defined by an interface. In the past, if a new method were added to a popular, widely used interface, then the addition of that method would break existing code because no implementation would be found for that new method. The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided. Thus, the addition of a default method will not cause preexisting code to break.

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used. For example, an interface might define a group of methods that act on a sequence of elements. One of these methods might be called **remove( )**, and its purpose is to remove an element from the sequence. However, if the interface is intended to support both modifiable and nonmodifiable sequences, then **remove( )** is essentially optional because it won't be used by nonmodifiable sequences. In the past, a class that implemented a nonmodifiable sequence would have had to define an empty implementation of **remove( )**, even though it was not needed. Today, a default implementation for **remove( )** can be specified in the interface that

does nothing (or throws an exception). Providing this default prevents a class used for nonmodifiable sequences from having to define its own, placeholder version of **remove()**. Thus, by providing a default, the interface makes the implementation of **remove()** by a class optional.

It is important to point out that the addition of default methods does not change a key aspect of **interface**: its inability to maintain state information. An interface still cannot have instance variables, for example. Thus, the defining difference between an interface and a class is that a class can maintain state information, but an interface cannot. Furthermore, it is still not possible to create an instance of an interface by itself. It must be implemented by a class. Therefore, even though, beginning with JDK 8, an interface can define default methods, the interface must still be implemented by a class if an instance is to be created.

One last point: As a general rule, default methods constitute a special-purpose feature. Interfaces that you create will still be used primarily to specify *what* and not *how*. However, the inclusion of the default method gives you added flexibility.

## Default Method Fundamentals

An interface default method is defined similar to the way a method is defined by a **class**. The primary difference is that the declaration is preceded by the keyword **default**. For example, consider this simple interface:

```
public interface MyIF {  
    // This is a "normal" interface method declaration.  
    // It does NOT define a default implementation.  
    int getNumber();  
  
    // This is a default method. Notice that it provides  
    // a default implementation.  
    default String getString() {  
        return "Default String";  
    }  
}
```

**MyIF** declares two methods. The first, **getNumber()**, is a standard interface method declaration. It defines no implementation whatsoever. The second method is **getString()**, and it does include a default implementation. In this case, it simply returns the string "Default String". Pay special attention to the way **getString()** is declared. Its declaration is preceded by the **default** modifier. This syntax can be generalized. To define a default method, precede its declaration with **default**.

Because **getString()** includes a default implementation, it is not necessary for an implementing class to override it. In other words, if an implementing class does not provide its own implementation, the default is used. For example, the **MyIFImp** class shown next is perfectly valid:

```
// Implement MyIF.
class MyIFImp implements MyIF {
    // Only getNumber() defined by MyIF needs to be implemented.
    // getString() can be allowed to default.
    public int getNumber() {
        return 100;
    }
}
```

The following code creates an instance of **MyIFImp** and uses it to call both **getNumber()** and **getString()**:

```
// Use the default method.
class DefaultMethodDemo {
    public static void main(String[] args) {

        MyIFImp obj = new MyIFImp();

        // Can call getNumber(), because it is explicitly
        // implemented by MyIFImp:
        System.out.println(obj.getNumber());

        // Can also call getString(), because of default
        // implementation:
        System.out.println(obj.getString());
    }
}
```

The output is shown here:

```
100
Default String
```

As you can see, the default implementation of **getString( )** was automatically used. It was not necessary for **MyIFImp** to define it. Thus, for **getString( )**, implementation by a class is optional. (Of course, its implementation by a class will be *required* if the class uses **getString( )** for some purpose beyond that supported by its default.)

It is both possible and common for an implementing class to define its own implementation of a default method. For example, **MyIFImp2** overrides **getString( )**:

```
class MyIFImp2 implements MyIF {
    // Here, implementations for both getNumber( ) and getString( ) are provided.
    public int getNumber() {
        return 100;
    }

    public String getString() {
        return "This is a different string.";
    }
}
```

Now, when **getString( )** is called, a different string is returned.

## A More Practical Example

Although the preceding shows the mechanics of using default methods, it doesn't illustrate their usefulness in a more practical setting. To do this, let's once again return to the **IntStack** interface shown earlier in this chapter. For the sake of discussion, assume that **IntStack** is widely used and many programs rely on it. Further assume that we now want to add a method to **IntStack** that clears the stack, enabling the stack to be re-used. Thus, we want to evolve the **IntStack** interface so that it defines new functionality, but we don't want to break any preexisting code. In the past, this would be impossible, but with the inclusion of default methods, it is

now easy to do. For example, the **IntStack** interface can be enhanced like this:

```
interface IntStack {  
    void push(int item); // store an item  
    int pop(); // retrieve an item  
  
    // Because clear( ) has a default, it need not be  
    // implemented by a preexisting class that uses IntStack.  
    default void clear() {  
        System.out.println("clear() not implemented.");  
    }  
}
```

Here, the default behavior of **clear()** simply displays a message indicating that it is not implemented. This is acceptable because no preexisting class that implements **IntStack** would ever call **clear()** because it was not defined by the earlier version of **IntStack**. However, **clear()** can be implemented by a new class that implements **IntStack**. Furthermore, **clear()** needs to be defined by a new implementation only if it is used. Thus, the default method gives you

- a way to gracefully evolve interfaces over time, and
- a way to provide optional functionality without requiring that a class provide a placeholder implementation when that functionality is not needed.

One other point: In real-world code, **clear()** would have thrown an exception, rather than displaying an error message. Exceptions are described in the next chapter. After working through that material, you might want to try modifying **clear()** so that its default implementation throws an **UnsupportedOperationException**.

## Multiple Inheritance Issues

As explained earlier in this book, Java does not support the multiple inheritance of classes. Now that an interface can include default methods, you might be wondering if an interface can provide a way around this

restriction. The answer is, essentially, no. Recall that there is still a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot.

The preceding notwithstanding, default methods do offer a bit of what one would normally associate with the concept of multiple inheritance. For example, you might have a class that implements two interfaces. If each of these interfaces provides default methods, then some behavior is inherited from both. Thus, to a limited extent, default methods do support multiple inheritance of behavior. As you might guess, in such a situation, it is possible that a name conflict will occur.

For example, assume that two interfaces called **Alpha** and **Beta** are implemented by a class called **MyClass**. What happens if both **Alpha** and **Beta** provide a method called **reset()** for which both declare a default implementation? Is the version by **Alpha** or the version by **Beta** used by **MyClass**? Or, consider a situation in which **Beta** extends **Alpha**. Which version of the default method is used? Or, what if **MyClass** provides its own implementation of the method? To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

First, in all cases, a class implementation takes priority over an interface default implementation. Thus, if **MyClass** provides an override of the **reset()** default method, **MyClass**'s version is used. This is the case even if **MyClass** implements both **Alpha** and **Beta**. In this case, both defaults are overridden by **MyClass**'s implementation.

Second, in cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result. Continuing with the example, if **MyClass** implements both **Alpha** and **Beta**, but does not override **reset()**, then an error will occur.

In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore, continuing the example, if **Beta** extends **Alpha**, then **Beta**'s version of **reset()** will be used.

It is possible to explicitly refer to a default implementation in an inherited interface by using this form of **super**. Its general form is shown here:

*InterfaceName.super.methodName()*

For example, if **Beta** wants to refer to **Alpha**'s default for **reset( )**, it can use this statement:

```
Alpha.super.reset();
```

## Use static Methods in an Interface

Another capability added to **interface** by JDK 8 is the ability to define one or more **static** methods. Like **static** methods in a class, a **static** method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a **static** method. Instead, a **static** method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

*InterfaceName.staticMethodName*

Notice that this is similar to the way that a **static** method in a class is called.

The following shows an example of a **static** method in an interface by adding one to **MyIF**, shown in the previous section. The **static** method is **getDefaultNumber( )**. It returns zero.

```
public interface MyIF {
    // This is a "normal" interface method declaration.
    // It does NOT define a default implementation.
    int getNumber();

    // This is a default method. Notice that it provides
    // a default implementation.
    default String getString() {
        return "Default String";
    }

    // This is a static interface method.
    static int getDefaultNumber() {
        return 0;
    }
}
```

The **getDefaultNumber( )** method can be called as shown here:

```
int defNum = MyIF.getDefaultNumber();
```

As mentioned, no implementation or instance of **MyIF** is required to call **getDefaultNumber( )** because it is **static**.

One last point: **static** interface methods are not inherited by either an implementing class or a subinterface.

## Private Interface Methods

Beginning with JDK 9, an interface can include a private method. A private interface method can be called only by a default method or another private method defined by the same interface. Because a private interface method is specified **private**, it cannot be used by code outside the interface in which it is defined. This restriction includes subinterfaces because a private interface method is not inherited by a subinterface.

The key benefit of a private interface method is that it lets two or more default methods use a common piece of code, thus avoiding code duplication. For example, here is another version of the **IntStack** interface that has two default methods called **popNElements( )** and **skipAndPopNElements( )**. The first returns an array that contains the top

$N$  elements on the stack. The second skips a specified number of elements and then returns an array that contains the next  $N$  elements. Both use a private method called **getElements()** to obtain an array of the specified number of elements from the stack.

```
// Another version of IntStack that has a private interface
// method that is used by two default methods.
interface IntStack {
    void push(int item); // store an item
    int pop(); // retrieve an item

    // A default method that returns an array that contains
    // the top n elements on the stack.
    default int[] popNElements(int n) {
        // Return the requested elements.
        return getElements(n);
    }

    // A default method that returns an array that contains
    // the next n elements on the stack after skipping elements.
    default int[] skipAndPopNElements(int skip, int n) {

        // Skip the specified number of elements.
        getElements(skip);

        // Return the requested elements.
        return getElements(n);
    }

    // A private method that returns an array containing
    // the top n elements on the stack
    private int[] getElements(int n) {
        int[] elements = new int[n];

        for(int i=0; i < n; i++) elements[i] = pop();
        return elements;
    }
}
```

Notice that both `popNElements()` and `skipAndPopNElements()` use the private `getElements()` method to obtain the array to return. This prevents both methods from having to duplicate the same code sequence. Keep in mind that because `getElements()` is private, it cannot be called by code outside `IntStack`. Thus, its use is limited to the default methods inside `IntStack`. Also, because `getElements()` uses the `pop()` method to obtain stack elements, it will automatically call the implementation of `pop()` provided by the `IntStack` implementation. Thus, `getElements()` will work for any stack class that implements `IntStack`.

Although the private interface method is a feature that you will seldom need, in those cases in which you *do* need it, you will find it quite useful.

## Final Thoughts on Packages and Interfaces

Although the examples we've included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment. Virtually all real programs that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

# CHAPTER

## 10

# Exception Handling

---

This chapter examines Java’s exception-handling mechanism. An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java’s exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

## Exception-Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a

**try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

This is the general form of an exception-handling block:

```
try {  
    // block of code to monitor for errors  
}  
  
catch (ExceptionType1 exOb) {  
    // exception handler for ExceptionType1  
}  
  
catch (ExceptionType2 exOb) {  
    // exception handler for ExceptionType2  
}  
// ...  
finally {  
    // block of code to be executed after try block ends  
}
```

Here, *ExceptionType* is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

---

**NOTE** There is another form of the **try** statement that supports *automatic resource management*. This form of **try**, called **try-with-resources**, is described in [Chapter 13](#) in the context of managing files because files are some of the most commonly used resources.

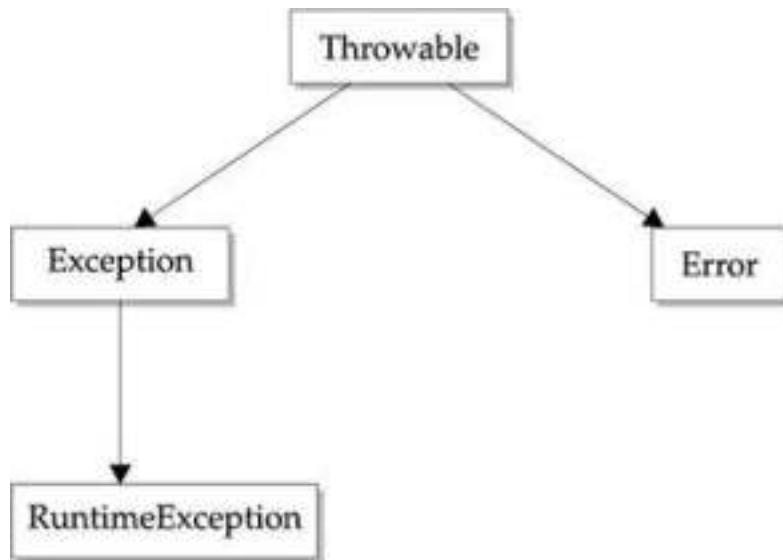
## Exception Types

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two

distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

The top-level exception hierarchy is shown here:



## Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {  
    public static void main(String[] args) {  
        int d = 0;  
        int a = 42 / d;  
    }  
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:

```
java.lang.ArithmetricException: / by zero  
at Exc0.main(Exc0.java:4)
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, 4, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmetricException**, which more specifically describes what type of error happened. As discussed later in this chapter, Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated. One other point: The exact output you see when running this and other sample programs in this chapter that use Java's built-in exceptions may vary slightly from what is shown because of differences between JDks.

The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from **main()**:

```
class Excl {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String[] args) {
        Excl.subroutine();
    }
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmetricException: / by zero
at Excl.subroutine(Excl.java:4)
at Excl.main(Excl.java:7)
```

As you can see, the bottom of the stack is **main**'s line 7, which is the call to **subroutine()**, which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

## Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause that processes the **ArithmetricException** generated by the division-by-zero error:

```
class Exc2 {
    public static void main(String[] args) {
        int d, a;

        try { // monitor a block of code.
            d = 0;
            a = 42 / d;
            System.out.println("This will not be printed.");
        } catch (ArithmaticException e) { // catch divide-by-zero error
            System.out.println("Division by zero.");
        }

        System.out.println("After catch statement.");
    }
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Notice that the call to **println()** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not “called,” so execution never “returns” to the **try** block from a **catch**. Thus, the line “This will not be printed.” is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.

A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements, described shortly). The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement.

The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program, each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other,

and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

```
// Handle an exception and move on.  
import java.util.Random;  
  
class HandleError {  
    public static void main(String[] args) {  
        int a=0, b=0, c=0;  
        Random r = new Random();  
  
        for(int i=0; i<32000; i++) {  
            try {  
                b = r.nextInt();  
                c = r.nextInt();  
                a = 12345 / (b/c);  
            } catch (ArithmetricException e) {  
                System.out.println("Division by zero.");  
                a = 0; // set a to zero and continue  
            }  
            System.out.println("a: " + a);  
        }  
    }  
}
```

## Displaying a Description of an Exception

**Throwable** overrides the **toString()** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println()** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmetricException e) {  
    System.out.println("Exception: " + e);  
    a = 0; // set a to zero and continue  
}
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmetricException: / by zero
```

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

## Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block. The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultipleCatches {
    public static void main(String[] args) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int[] c = { 1 };
            c[42] = 99;
        } catch(ArithmetricException e) {
            System.out.println("Divide by 0: " + e);
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index oob: " + e);
        }
        System.out.println("After try/catch blocks.");
    }
}
```

This program will cause a division-by-zero exception if it is started with no command-line arguments, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

Here is the output generated by running it both ways:

```
C:\>java MultipleCatches
a = 0
Divide by 0: java.lang.ArithmetricException: / by zero
After try/catch blocks.

C:\>java MultipleCatches TestArg
a = 1
Array index oob:  java.lang.ArrayIndexOutOfBoundsException:
Index 42 out of bounds for length 1
After try/catch blocks.
```

When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

```
/* This program contains an error.

A subclass must come before its superclass in
a series of catch statements. If not,
unreachable code will be created and a
compile-time error will result.

*/
class SuperSubCatch {
    public static void main(String[] args) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch(Exception e) {
            System.out.println("Generic Exception catch.");
        }
        /* This catch is never reached because
           ArithmeticException is a subclass of Exception. */
        catch(ArithmeticException e) { // ERROR - unreachable
            System.out.println("This is never reached.");
        }
    }
}
```

If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable because the exception has already been caught. Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**. This means that the second **catch** statement will never execute. To fix the problem, reverse the order of the **catch** statements.

## Nested try Statements

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the

nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested **try** statements:

```
// An example of nested try statements.
class NestTry {
    public static void main(String[] args) {
        try {
            int a = args.length;

            /* If no command-line args are present,
               the following statement will generate
               a divide-by-zero exception. */
            int b = 42 / a;

            System.out.println("a = " + a);

            try { // nested try block
                /* If one command-line arg is used,
                   then a divide-by-zero exception
                   will be generated by the following code. */
                if(a==1) a = a/(a-a); // division by zero

                /* If two command-line args are used,
                   then generate an out-of-bounds exception. */
                if(a==2) {
                    int[] c = { 1 };
                    c[42] = 99; // generate an out-of-bounds exception
                }
            } catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index out-of-bounds: " + e);
            }

        } catch(ArithmetricException e) {
            System.out.println("Divide by 0: " + e);
        }
    }
}
```

As you can see, this program nests one **try** block within another. The program works as follows. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested **try** block. Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner **try** block. Here are sample runs that illustrate each case:

```
C:\>java NestTry
Divide by 0: java.lang.ArithmetricException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmetricException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
    java.lang.ArrayIndexOutOfBoundsException:
        Index 42 out of bounds for length 1
```

Nesting of **try** statements can occur in less obvious ways when method calls are involved. For example, you can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method. Here is the previous program recoded so that the nested **try** block is moved inside the method **nesttry()**:

```
/* Try statements can be implicitly nested via
   calls to methods. */
class MethNestTry {
    static void nesttry(int a) {
        try { // nested try block
            /* If one command-line arg is used,
               then a divide-by-zero exception
               will be generated by the following code. */
            if(a==1) a = a/(a-a); // division by zero

            /* If two command-line args are used,
               then generate an out-of-bounds exception. */
            if(a==2) {
                int[] c = { 1 };
                c[42] = 99; // generate an out-of-bounds exception
            }
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out-of-bounds: " + e);
        }
    }

    public static void main(String[] args) {
        try {
            int a = args.length;

            /* If no command-line args are present,
               the following statement will generate
               a divide-by-zero exception. */
            int b = 42 / a;
            System.out.println("a = " + a);

            nesttry(a);
        } catch(ArithmaticException e) {
            System.out.println("Divide by 0: " + e);
        }
    }
}
```

The output of this program is identical to that of the preceding example.

## **throw**

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

```
throw ThrowableInstance;
```

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter in a **catch** clause or creating one with the **new** operator.

The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }

    public static void main(String[] args) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

This program gets two chances to deal with the same error. First, `main()` sets up an exception context and then calls `demoproc()`. The `demoproc()` method then sets up another exception-handling context and immediately throws a new instance of `NullPointerException`, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

```
throw new NullPointerException("demo");
```

Here, `new` is used to construct an instance of `NullPointerException`. Many of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to `print()`

or `println()`. It can also be obtained by a call to `getMessage()`, which is defined by **Throwable**.

## throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
    // body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
    static void throwOne() {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String[] args) {
        throwOne();
    }
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try/catch** statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String[] args) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Here is the output generated by running this sample program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

## **finally**

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.

**finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an

exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

Here is a sample program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
    // Throw an exception out of the method.
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("inside finally");
        }
    }
}
```

```
    } finally {
        System.out.println("procA's finally");
    }
}

// Return from within a try block.
static void procB() {
    try {
        System.out.println("inside procB");
        return;
    } finally {
        System.out.println("procB's finally");
    }
}

// Execute a try block normally.
static void procC() {
    try {
        System.out.println("inside procC");
    } finally {
        System.out.println("procC's finally");
    }
}

public static void main(String[] args) {
    try {
        procA();
    } catch (Exception e) {
        System.out.println("Exception caught");
    }

    procB();
    procC();
}
}
```

In this example, **procA()** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB()**'s **try** statement is exited via a **return** statement. The **finally** clause is executed

before `procB()` returns. In `procC()`, the `try` statement executes normally, without error. However, the `finally` block is still executed.

---

**REMEMBER** If a `finally` block is associated with a `try`, the `finally` block will be executed upon conclusion of the `try`.

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

## Java's Built-in Exceptions

Inside the standard package `java.lang`, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. As previously explained, these exceptions need not be included in any method's `throws` list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in `java.lang` are listed in [Table 10-1](#). [Table 10-2](#) lists those exceptions defined by `java.lang` that must be included in a method's `throws` list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*. In addition to the exceptions in `java.lang`, Java defines several more that relate to its other standard packages.

Exception	Meaning
ArithmaticException	Arithmatic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out of bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalCallerException	A method cannot be legally executed by the calling code.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out of bounds.
LayerInstantiationException	A module layer cannot be created.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotFoundException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

**Table 10-1** Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the <b>Cloneable</b> interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

**Table 10-2** Java’s Checked Exceptions Defined in `java.lang`

## Creating Your Own Exception Subclasses

Although Java’s built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). Your subclasses don’t need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them. They are shown in [Table 10-3](#). You may also wish to override one or more of these methods in exception classes that you create.

Method	Description
final void addSuppressed(Throwable exc)	Adds <i>exc</i> to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the <b>try-with-resources</b> statement.
Throwable fillInStackTrace( )	Returns a <b>Throwable</b> object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, <b>null</b> is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of <b>StackTraceElement</b> . The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The <b>StackTraceElement</b> class gives your program access to information about each element in the trace, such as its method name.
final Throwable[ ] getSuppressed( )	Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. Suppressed exceptions are primarily generated by the <b>try-with-resources</b> statement.
Throwable initCause(Throwable causeExc)	Associates <i>causeExc</i> with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream stream)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter stream)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement[ ] elements)	Sets the stack trace to the elements passed in <i>elements</i> . This method is for specialized applications, not normal use.

**Table 10-3** The Methods Defined by **Throwable**

String <b>toString()</b>	Returns a <b>String</b> object containing a description of the exception. This method is called by <b>println()</b> when outputting a <b>Throwable</b> object.
<b>Exception()</b>	
<b>Exception(String msg)</b>	

The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Although specifying a description when an exception is created is often useful, sometimes it is better to override **toString()**. Here's why: The version of **toString()** defined by **Throwable** (and inherited by **Exception**) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding **toString()**, you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

The following example declares a new subclass of **Exception** and then uses that subclass to signal an error condition in a method. It overrides the **toString()** method, allowing a carefully tailored description of the exception to be displayed.

```
// This program creates a custom exception type.
class MyException extends Exception {
    private int detail;

    MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}

class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
            throw new MyException(a);
        System.out.println("Normal exit");
    }
}

public static void main(String[] args) {
    try {
        compute(1);
        compute(20);
    } catch (MyException e) {
        System.out.println("Caught " + e);
    }
}
```

This example defines a subclass of **Exception** called **MyException**. This subclass is quite simple: It has only a constructor plus an overridden **toString()** method that displays the value of the exception. The **ExceptionDemo** class defines a method named **compute()** that throws a **MyException** object. The exception is thrown when **compute()**'s integer parameter is greater than 10. The **main()** method sets up an exception handler for **MyException**, then calls **compute()** with a legal value (less than 10) and an illegal one to show both paths through the code. Here is the result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

## Chained Exceptions

A number of years ago, a feature was incorporated into the exception subsystem: *chained exceptions*. The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an **ArithmaticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an **ArithmaticException**, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

To allow chained exceptions, two constructors and two methods were added to **Throwable**. The constructors are shown here:

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

In the first form, *causeExc* is the exception that causes the current exception. That is, *causeExc* is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

The chained exception methods supported by **Throwable** are **getCause()** and **initCause()**. These methods are shown in [Table 10-3](#) and are repeated here for the sake of discussion.

```
Throwable getCause()
Throwable initCause(Throwable causeExc)
```

The **getCause( )** method returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned. The **initCause( )** method associates *causeExc* with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created. However, the cause exception can be set only once. This means that you can call **initCause( )** only once for each exception object. Furthermore, if the cause exception was set by a constructor, then you can't set it again using **initCause()**. In general, **initCause()** is used to set a cause for legacy exception classes that don't support the two additional constructors described earlier.

Here is an example that illustrates the mechanics of handling chained exceptions:

```
// Demonstrate exception chaining.
class ChainExcDemo {
    static void demoproc() {

        // create an exception
        NullPointerException e =
            new NullPointerException("top layer");

        // add a cause
        e.initCause(new ArithmeticException("cause"));

        throw e;
    }

    public static void main(String[] args) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            // display top level exception
            System.out.println("Caught: " + e);

            // display cause exception
            System.out.println("Original cause: " +
                e.getCause());
        }
    }
}
```

The output from the program is shown here:

```
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmetricException: cause
```

In this example, the top-level exception is **NullPointerException**. To it is added a cause exception, **ArithmetricException**. When the exception is thrown out of **demoproc()**, it is caught by **main()**. There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling **getCause()**.

Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design.

Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

## Three Additional Exception Features

Beginning with JDK 7, three interesting and useful features have been part of the exception system. The first automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of the **try** statement called **try-with-resources**, and is described in [Chapter 13](#) when files are introduced. The second feature is called *multi-catch*, and the third is sometimes referred to as *final rethrow* or *more precise rethrow*. These two features are described here.

The multi-catch feature allows two or more exceptions to be caught by the same **catch** clause. It is not uncommon for two or more exception handlers to use the same code sequence even though they respond to different exceptions. Instead of having to catch each exception type individually, you can use a single **catch** clause to handle all of the exceptions without code duplication.

To use a multi-catch, separate each exception type in the **catch** clause with the OR operator. Each multi-catch parameter is implicitly **final**. (You can explicitly specify **final**, if desired, but it is not necessary.) Because each multi-catch parameter is implicitly **final**, it can't be assigned a new value.

Here is a **catch** statement that uses the multi-catch feature to catch both **ArithmetiException** and **ArrayIndexOutOfBoundsException**:

```
catch(ArithmetiException | ArrayIndexOutOfBoundsException  
e) {
```

The following program shows the multi-catch feature in action:

```
// Demonstrate the multi-catch feature.  
class MultiCatch {  
    public static void main(String[] args) {  
        int a=10, b=0;  
        int[] vals = { 1, 2, 3 };  
  
        try {  
            int result = a / b; // generate an ArithmetiException  
  
//          vals[10] = 19; // generate an ArrayIndexOutOfBoundsException  
  
            // This catch clause catches both exceptions.  
        } catch(ArithmetiException | ArrayIndexOutOfBoundsException e) {  
            System.out.println("Exception caught: " + e);  
        }  
  
        System.out.println("After multi-catch.");  
    }  
}
```

The program will generate an **ArithmetiException** when the division by zero is attempted. If you comment out the division statement and remove the comment symbol from the next line, an **ArrayIndexOutOfBoundsException** is generated. Both exceptions are caught by the single **catch** statement.

The more precise rethrow feature restricts the type of exceptions that can be rethrown to only those checked exceptions that the associated **try** block throws, that are not handled by a preceding **catch** clause, and that are a subtype or supertype of the parameter. Although this capability might not be needed often, it is now available for use. For the more precise rethrow feature to be in force, the **catch** parameter must be either effectively **final**,

which means that it must not be assigned a new value inside the **catch** block, or explicitly declared **final**.

## Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of **try**, **throw**, and **catch** as clean ways to handle errors and unusual boundary conditions in your program's logic. Instead of using error return codes to indicate failure, use Java's exception handling capabilities. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.

One last point: Java's exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.

# CHAPTER

## 11

# Multithreaded Programming

---

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

You are almost certainly acquainted with multitasking because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. For many readers, process-based multitasking is the more familiar form. A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a website. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context

switching from one process to another is also costly. Threads, on the other hand, are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java's direct control. However, multithreaded multitasking is.

Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time to a minimum. This is especially important for the interactive, networked environment in which Java operates because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though most of the time the program is idle, waiting for input. Multithreading helps you reduce this idle time because another thread can run when one is waiting.

If you have programmed for operating systems such as Windows, then you are already familiar with multithreaded programming. However, the fact that Java manages threads makes multithreading especially convenient because many of the details are handled for you.

## The Java Thread Model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems often use an approach called an *event loop* with *polling*. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is

ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the program. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a single-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running. This is an example of how only being able to do one thing at a time, something common to all single-threaded systems, whatever approach they use, is an inefficient way to use the CPU.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

As most readers know, over the past few years, multicore systems have become commonplace. Of course, single-core systems are still in use. It is important to understand that Java's multithreading features work in both types of systems. In a single-core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized. However, in multicore systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.

Java has straddled these two CPU architectures with the thread model by implementing Java threads, each as a wrapper around an OS thread. Such Java threads are known as platform threads. For most developers, platform threads, when used correctly, give great scalability and performance characteristics. However, for some high-throughput server applications, particularly ones managing many connections simultaneously like a heavily trafficked web application, this model can lead to performance bottlenecks. For these special kinds of applications, JDK 21 has added a new way to map Java threads onto OS threads. Java threads using this new feature are called *virtual threads*. When an application uses virtual threads, the JVM is

freed from tying each virtual thread one-to-one with an OS thread, offering greater efficiencies for highly multithreaded applications. While most developers will continue to use the default platform thread model, for those who do wish to explore virtual threads, fortunately, the programming model for the new virtual threads is the same as for platform threads. We'll cover virtual threads at the end of the chapter.

---

**NOTE** In addition to the multithreading features described in this chapter, you will also want to explore the Fork/Join Framework. It provides a powerful means of creating multithreaded applications that automatically scale to make best use of multicore environments. The Fork/Join Framework is part of Java's support for *parallel programming*, which is the name commonly given to the techniques that optimize some types of algorithms for parallel execution in systems that have more than one CPU. For a discussion of the Fork/Join Framework and other concurrency utilities, see [Chapter 29](#). Java's traditional multithreading capabilities are described here.

Threads exist in several states. Here is a general description: A thread can be *running*. It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended*, which temporarily halts its activity. A suspended thread can then be *resumed*, allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

## Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*. The rules that determine when a *context switch* takes place are simple:

- *A thread can voluntarily relinquish control.* This occurs when explicitly yielding, sleeping, or when blocked. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

- A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking*.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For some operating systems, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

---

**CAUTION** Portability problems can arise from the differences in the way that operating systems context-switch threads of equal priority.

## Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor*. The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

In Java, there is no class “Monitor”; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

# Messaging

After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with some other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

## The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

The **Thread** class defines several methods that help manage threads. Several of those used in this chapter are shown here:

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Thus far, all the examples in this book have used a single thread of execution. The remainder of this chapter explains how to use **Thread** and

**Runnable** to create and manage threads, beginning with the one thread that all Java programs have: the main thread.

## The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread()**, which is a **public static** member of **Thread**. Its general form is shown here:

```
static Thread currentThread()
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let's begin by reviewing the following example:

```
// Controlling the main Thread.
class CurrentThreadDemo {
    public static void main(String[] args) {
        Thread t = Thread.currentThread();

        System.out.println("Current thread: " + t);

        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);
```

```
try {
    for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
    }
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted");
}
}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread()**, and this reference is stored in the local variable **t**. Next, the program displays information about the thread. The program then calls **setName()** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep()** method. The argument to **sleep()** specifies the delay period in milliseconds. Notice the **try/catch** block around this loop. The **sleep()** method in **Thread** might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Notice the output produced when `t` is used as an argument to `println()`. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is `main`. Its priority is 5, which is the default value, and `main` is also the name of the group of threads to which this thread belongs. A *thread group* is a data structure that controls the state of a collection of threads as a whole. After the name of the

thread is changed, **t** is again output. This time, the new name of the thread is displayed.

Let's look more closely at the methods defined by **Thread** that are used in the program. The **sleep()** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

```
static void sleep(long milliseconds) throws InterruptedException
```

The number of milliseconds to suspend is specified in *milliseconds*. This method may throw an **InterruptedException**.

The **sleep()** method has a second form, shown next, that allows you to specify the period in terms of milliseconds and nanoseconds:

```
static void sleep(long milliseconds, int nanoseconds) throws  
InterruptedException
```

This second form is useful only in environments that allow timing periods as short as nanoseconds.

As the preceding program shows, you can set the name of a thread by using **setName()**. You can obtain the name of a thread by calling **getName()** (but note that this is not shown in the program). These methods are members of the **Thread** class and are declared like this:

```
final void setName(String threadName)  
final String getName()
```

Here, *threadName* specifies the name of the thread.

## Creating a Thread

In the most general sense, you create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class itself.

The following two sections look at each method, in turn.

# Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

```
public void run( )
```

Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.

After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName)
```

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** initiates a call to **run( )**. The **start( )** method is shown here:

```
void start( )
```

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
    Thread t;
```

```
NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
}

// This is the entry point for the second thread.
public void run() {
    try {
        for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            Thread.sleep(500);
        }
    } catch (InterruptedException e) {
        System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
}

}

class ThreadDemo {
    public static void main(String[] args) {
        NewThread nt = new NewThread(); // create a new thread

        nt.t.start(); // Start the thread

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing **this** as the first argument indicates that you want the new thread to call the **run( )** method on the **this** object. Inside **main( )**, **start( )** is called, which starts the thread of execution beginning at the **run( )** method. This causes the child thread's **for** loop to begin. Next, the main thread enters its **for** loop. Both threads continue running, sharing the CPU in single-core systems, until their loops finish. The output produced by this program is as follows.(Your output may vary based upon the specific execution environment.)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4

Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

As mentioned earlier, in a multithreaded program, it is often useful for the main thread to be the last thread to finish running. The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread. Shortly, you will see a better way to wait for a thread to finish.

## Extending Thread

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new

thread. As before, a call to **start()** begins execution of the new thread. Here is the preceding program rewritten to extend **Thread**:

```
// Create a second thread by extending Thread
class NewThread extends Thread {

    NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ExtendThread {
    public static void main(String[] args) {
        NewThread nt = new NewThread(); // create a new thread
        nt.start(); // start the thread
    }
}
```

```

try {
    for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
    }
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}

```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.

Notice the call to **super()** inside **NewThread**. This invokes the following form of the **Thread** constructor:

```
public Thread(String threadName)
```

Here, *threadName* specifies the name of the thread.

## Choosing an Approach

At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point. The **Thread** class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must* be overridden is **run()**. This is, of course, the same method required when you implement **Runnable**. Many Java programmers feel that classes should be extended only when they are being enhanced or adapted in some way. So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**. Also, by implementing **Runnable**, your thread class does not need to inherit **Thread**, making it free to inherit a different class. Ultimately, which approach to use is up to you. However, throughout the rest of this chapter, we will create threads by using classes that implement **Runnable**.

## Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
    }
}
```

```
// This is the entry point for thread.
public void run() {
    try {
        for(int i = 5; i > 0; i--) {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        System.out.println(name + "Interrupted");
    }
    System.out.println(name + " exiting.");
}
}

class MultiThreadDemo {
    public static void main(String[] args) {
        NewThread nt1 = new NewThread("One");
        NewThread nt2 = new NewThread("Two");
        NewThread nt3 = new NewThread("Three");

        // Start the threads.
        nt1.t.start();
        nt2.t.start();
        nt3.t.start();

        try {
            // wait for other threads to end
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Main thread exiting.");
    }
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2

Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

As you can see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main()**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

## Using **isAlive()** and **join()**

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling **sleep()** within **main()**, with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished. First, you can call **isAlive()** on the thread. This method is defined by **Thread**, and its general form is shown here:

```
final boolean isAlive()
```

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here:

```
final void join() throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Here is an improved version of the preceding example that uses **join()** to ensure that the main thread is the last to finish. It also demonstrates the **isAlive( )** method.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
    }
}
```

```
// This is the entry point for thread.
public void run() {
    try {
        for(int i = 5; i > 0; i--) {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
}
}

class DemoJoin {
    public static void main(String[] args) {
        NewThread nt1 = new NewThread("One");
        NewThread nt2 = new NewThread("Two");
        NewThread nt3 = new NewThread("Three");

        // Start the threads.
        nt1.t.start();
        nt2.t.start();
        nt3.t.start();

        System.out.println("Thread One is alive: "
                           + nt1.t.isAlive());
        System.out.println("Thread Two is alive: "
                           + nt2.t.isAlive());
        System.out.println("Thread Three is alive: "
                           + nt3.t.isAlive());
        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            nt1.t.join();
            nt2.t.join();
            nt3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
    }
}
```

Sample output from this program is shown here. (Your output may vary based upon the specific execution environment.)

```
+ nt1.t.isAlive();
+ nt2.t.isAlive());
New thread: Thread[One,5,main]  e is alive: "
New thread: Thread[Two,5,main]  live());
New thread: Thread[Three,5,main]
Thread One is alive: true      exiting.");
Thread Two is alive: true
Thread Three is alive: true
} Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

As you can see, after the calls to **join()** return, the threads have stopped executing.

## Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, over a given period of time,

higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

In theory, threads of equal priority should get equal access to the CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others. You should think of the thread priority as a polite suggestion to the thread scheduler as to how to prioritize your threads at times when the system resources are getting stretched. But the thread scheduler will likely have other factors to consider as well. So you should experiment with your application in a range of conditions if you are using thread priorities and not depend on it as if it were a strict rule.

To set a thread's priority, use the `setPriority()` method, which is a member of **Thread**. This is its general form:

```
final void setPriority(int level)
```

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN\_PRIORITY** and **MAX\_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM\_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**.

You can obtain the current priority setting by calling the `getPriority()` method of **Thread**, shown here:

```
final int getPriority()
```

Implementations of Java may have radically different behavior when it comes to scheduling. Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively

giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

## Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*. As you will see, Java provides unique, language-level support for it.

Key to synchronization is the concept of the monitor. A *monitor* is an object that is used as a mutually exclusive lock. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

## Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call()**. The **call()** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call()** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.

The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run( )** method. The **run( )** method of **Caller** calls the **call()** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

```
// This program is not synchronized.  
class Callme {  
    void call(String msg) {  
        System.out.print("[ " + msg);  
        try {  
            Thread.sleep(1000);  
        } catch(InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
        System.out.println("] ");  
    }  
}  
  
class Caller implements Runnable {  
    String msg;  
    Callme target;  
    Thread t;  
  
    public Caller(Callme targ, String s) {  
        target = targ;  
        msg = s;  
        t = new Thread(this);  
    }  
  
    public void run() {  
        target.call(msg);  
    }  
}  
  
class Synch {  
    public static void main(String[] args) {  
        Callme target = new Callme();  
        Caller ob1 = new Caller(target, "Hello");  
        Caller ob2 = new Caller(target, "Synchronized");  
        Caller ob3 = new Caller(target, "World");
```

```
// Start the threads.  
ob1.t.start();  
ob2.t.start();  
ob3.t.start();  
  
// wait for threads to end  
try {  
    ob1.t.join();  
    ob2.t.join();  
    ob3.t.join();  
} catch(InterruptedException e) {  
    System.out.println("Interrupted");  
}  
}  
}  
}
```

Here is the output produced by this program:

```
[Hello [Synchronized [World]  
]  
]  
]
```

As you can see, by calling **sleep()**, the **call()** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to prevent all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition*, because the three threads are racing each other to complete the method. This example used **sleep()** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next.

To fix the preceding program, you must *serialize* access to **call()**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call()**'s definition with the keyword **synchronized**, as shown here:

```
class Callme {  
    synchronized void call(String msg) {  
        ...  
    }  
}
```

This prevents other threads from entering `call()` while another thread is using it. After `synchronized` has been added to `call()`, the output of the program is as follows:

```
[Hello]  
[Synchronized]  
[World]
```

Any time that you have a method, or group of methods, that changes the internal state of an object in a multithreaded situation, you should consider using the `synchronized` keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

## The `synchronized` Statement

While creating `synchronized` methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following: Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use `synchronized` methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add `synchronized` to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a `synchronized` block.

This is the general form of the `synchronized` statement:

```
synchronized(objRef) {  
    // statements to be synchronized  
}
```

Here, *objRef* is a reference to the object being synchronized. A synchronized block ensures that a call to a synchronized method that is a

member of *objRef*'s *class* occurs only after the current thread has successfully entered *objRef*'s monitor.

Here is an alternative version of the preceding example, using a synchronized block within the **run( )** method:

```
// This program uses a synchronized block.
class Callme {
    void call(String msg) {
        System.out.print("[ " + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("] ");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
    }

    // synchronize calls to call()
    public void run() {
        synchronized(target) { // synchronized block
            target.call(msg);
        }
    }
}
```

```

        }
    }
}

class Synch1 {
    public static void main(String[] args) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");

        // Start the threads.
        ob1.t.start();
        ob2.t.start();
        ob3.t.start();

        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch(InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}

```

Here, the `call()` method is not modified by `synchronized`. Instead, the `synchronized` statement is used inside `Caller's run()` method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

## Interthread Communication

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interthread communication. As you will see, this is especially easy in Java.

As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete, logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interthread communication mechanism via the **wait()**, **notify()**, and **notifyAll()** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- **wait()** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify()** or **notifyAll()**.
- **notify()** wakes up a thread that called **wait()** on the same object.
- **notifyAll()** wakes up all the threads that called **wait()** on the same object. One of the threads will be granted access.

These methods are declared within **Object**, as shown here:

```
final void wait() throws InterruptedException  
final void notify()  
final void notifyAll()
```

Additional forms of **wait()** exist that allow you to specify a period of time to wait.

Before working through an example that illustrates interthread communication, an important point needs to be made. Although **wait()**

normally waits until **notify()** or **notifyAll()** is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a *spurious wakeup*. In this case, a waiting thread resumes without **notify()** or **notifyAll()** having been called. (In essence, the thread resumes for no apparent reason.) Because of this remote possibility, the Java API documentation recommends that calls to **wait()** should take place within a loop that checks the condition on which the thread is waiting. The following example shows this technique.

Let's now work through an example that uses **wait()** and **notify()**. To begin, consider the following sample program that incorrectly implements a simple form of the producer/consumer problem. It consists of four classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

```
// An incorrect implementation of a producer and consumer.
class Q {
    int n;

    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }

    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}

class Producer implements Runnable {
    Q q;
    Thread t;
```

```
Producer(Q q) {
    this.q = q;
    t = new Thread(this, "Producer");
}

public void run() {
    int i = 0;

    while(true) {
        q.put(i++);
    }
}
}

class Consumer implements Runnable {
    Q q;
    Thread t;

    Consumer(Q q) {
        this.q = q;
        t = new Thread(this, "Consumer");
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PC {
    public static void main(String[] args) {
        Q q = new Q();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);

        // Start the threads.
        p.t.start();
        c.t.start();

        System.out.println("Press Control-C to stop.");
    }
}
```

Although the **put( )** and **get( )** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything prevent the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use **wait( )** and **notify( )** to signal in both directions, as shown here:

```
// A correct implementation of a producer and consumer.
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        while(!valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }

        System.out.println("Got: " + n);
        valueSet = false;
        notify();
        return n;
    }

    synchronized void put(int n) {
        while(valueSet)
            try {
                wait();
            } catch(InterruptedException e) {
                System.out.println("InterruptedException caught");
            }

        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
    }
}

class Producer implements Runnable {
    Q q;
    Thread t;
```

```
Producer(Q q) {
    this.q = q;
    t = new Thread(this, "Producer");
}

public void run() {
    int i = 0;

    while(true) {
        q.put(i++);
    }
}
}

class Consumer implements Runnable {
    Q q;
    Thread t;

    Consumer(Q q) {
        this.q = q;
        t = new Thread(this, "Consumer");
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PCFixed {
    public static void main(String[] args) {
        Q q = new Q();
        Producer p = new Producer(q);
        Consumer c = new Consumer(q);

        // Start the threads.
        p.t.start();
        c.t.start();

        System.out.println("Press Control-C to stop.");
    }
}
```

Inside `get()`, `wait()` is called. This causes its execution to suspend until **Producer** notifies you that some data is ready. When this happens, execution inside `get()` resumes. After the data has been obtained, `get()` calls `notify()`. This tells **Producer** that it is okay to put more data in the queue. Inside `put()`, `wait()` suspends execution until **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and `notify()` is called. This tells **Consumer** that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

## Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is *deadlock*, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

- In general, it occurs unpredictably, when the two threads time-slice in just the right way.
- It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of

events than just described.)

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, **A** and **B**, with methods **foo( )** and **bar( )**, respectively, which pause briefly before trying to call a method in the other class. The main class, named **Deadlock**, creates an **A** and a **B** instance, and then calls **deadlockStart( )** to start a second thread that sets up the deadlock condition. The **foo( )** and **bar( )** methods use **sleep( )** as a way to force the deadlock condition to occur.

```
// An example of deadlock.
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered A.foo");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("A Interrupted");
        }
        System.out.println(name + " trying to call B.last()");
        b.last();
    }
}
```

```
synchronized void last() {
    System.out.println("Inside A.last");
}
}

class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");

        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("B Interrupted");
        }

        System.out.println(name + " trying to call A.last()");
        a.last();
    }

    synchronized void last() {
        System.out.println("Inside B.last");
    }
}

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();
    Thread t;

    Deadlock() {
        Thread.currentThread().setName("MainThread");
        t = new Thread(this, "RacingThread");
    }

    void deadlockStart() {
        t.start();
        a.foo(b); // get lock on a in this thread.
        System.out.println("Back in main thread");
    }
}
```

When you run this program, you will see the output shown here, although whether **A.foo()** or **B.bar()** executes first will vary based on the specific execution environment.

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

Because the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC. You will see that **RacingThread** owns the monitor on **b**, while it is waiting for the monitor on **a**. At the same time, **MainThread** owns **a** and is waiting to get **b**. This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

## Suspending, Resuming, and Stopping Threads

Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once it's suspended, restarting the thread is also a simple matter.

The mechanisms to suspend, stop, and resume threads differ between early versions of Java, such as Java 1.0, and more modern versions, beginning with Java 2. Prior to Java 2, a program used **suspend( )**, **resume( )**, and **stop( )**, which are methods defined by **Thread**, to pause, restart, and stop the execution of a thread, respectively. Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs. Here's why. The **suspend( )** method of the **Thread** class was deprecated by Java 2 several years ago, and will be removed in a future release. This was done because **suspend( )** can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

The **resume( )** method is also deprecated. It does not cause problems, but it cannot be used without the **suspend( )** method as its counterpart.

The **stop( )** method of the **Thread** class, too, was deprecated by Java 2, and will be removed in a future release. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that **stop( )** causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.

Because you can't now use the **suspend( )**, **resume( )**, or **stop( )** methods to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the **run( )** method periodically checks to determine whether that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running," the **run( )** method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

The following example illustrates how the **wait( )** and **notify( )** methods that are inherited from **Object** can be used to control the execution of a thread. Let us consider its operation. The **NewThread** class contains a **boolean** instance variable named **suspendFlag**, which is used to control the execution of the thread. It is initialized to **false** by the constructor. The **run( )** method contains a **synchronized** statement block that checks **suspendFlag**. If that variable is **true**, the **wait( )** method is invoked to suspend the execution of the thread. The **mysuspend( )** method sets **suspendFlag** to **true**. The **myresume( )** method sets **suspendFlag** to **false** and invokes **notify( )** to wake up the thread. Finally, the **main( )** method has been modified to invoke the **mysuspend( )** and **myresume( )** methods.

```
// Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }

    synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}
```

```
class SuspendResume {
    public static void main(String[] args) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");

        ob1.t.start(); // Start the thread
        ob2.t.start(); // Start the thread

        try {
            Thread.sleep(1000);
            ob1.mysuspend();
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");
            ob2.mysuspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }

        System.out.println("Main thread exiting.");
    }
}
```

When you run the program, you will see the threads suspend and resume. Later in this book, you will see more examples that use the modern mechanism of thread control. Although this mechanism may not appear as

simple to use as the old way, nevertheless, it is the way required to ensure that run-time errors don't occur. It is the approach that *must* be used for all new code.

## Obtaining a Thread's State

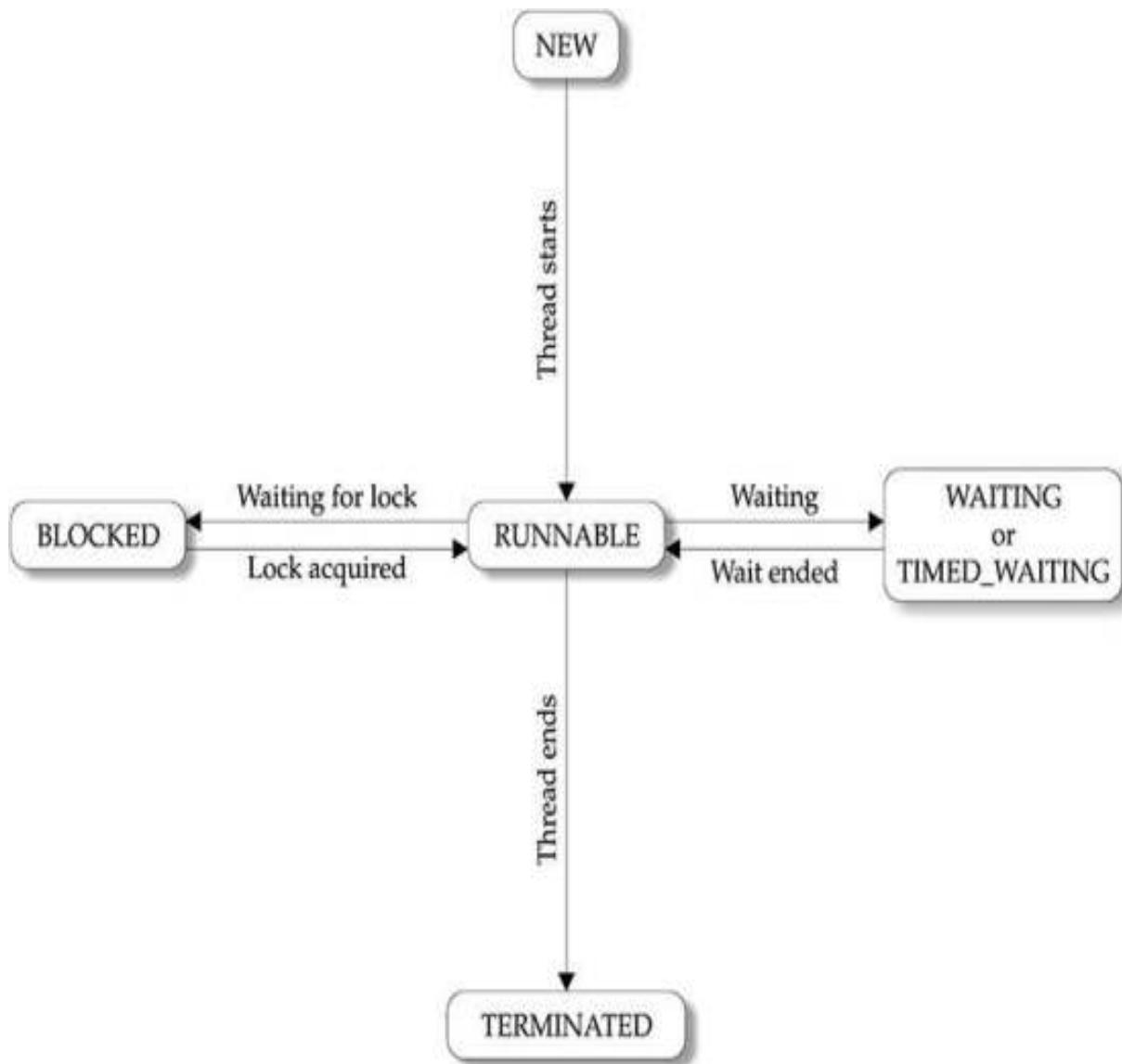
As mentioned earlier in this chapter, a thread can exist in a number of different states. You can obtain the current state of a thread by calling the `getState( )` method defined by **Thread**. It is shown here:

```
Thread.State getState( )
```

It returns a value of type **Thread.State** that indicates the state of the thread at the time at which the call was made. **State** is an enumeration defined by **Thread**. (An enumeration is a list of named constants. It is discussed in detail in [Chapter 12](#).) Here are the values that can be returned by `getState( )`:

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called <code>sleep( )</code> . This state is also entered when a timeout version of <code>wait( )</code> or <code>join( )</code> is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of <code>wait( )</code> or <code>join( )</code> .

[Figure 11-1](#) diagrams how the various thread states relate.



**Figure 11-1 Thread states**

Given a **Thread** instance, you can use **getState()** to obtain the state of a thread. For example, the following sequence determines if a thread called **thrd** is in the **RUNNABLE** state at the time **getState()** is called:

```

Thread.State ts = thrd.getState();

if(ts == Thread.State.RUNNABLE) // ...
  
```

It is important to understand that a thread's state may change after the call to **getState()**. Thus, depending on the circumstances, the state obtained by

calling `getState()` may not reflect the actual state of the thread only a moment later. For this (and other) reasons, `getState()` is not intended to provide a means of synchronizing threads. It's primarily used for debugging or for profiling a thread's run-time characteristics.

## Using a Factory Method to Create and Start a Thread

In some cases, it is not necessary to separate the creation of a thread from the start of its execution. In other words, sometimes it is convenient to create and start a thread at the same time. One way to do this is to use a static factory method. A *factory method* is a method that returns an object of a class. Typically, factory methods are *static* methods of a class. They are used for a variety of reasons, such as to set an object to some initial state prior to use, to configure a specific type of object, or in some cases to enable an object to be reused. As it relates to creating and starting a thread, a factory method will create the thread, call `start()` on the thread, and then return a reference to the thread. With this approach, you can create and start a thread through a single method call, thus streamlining your code.

For example, assuming the **ThreadDemo** program shown near the start of this chapter, adding the following factory method to **NewThread** enables you to create and start a thread in a single step:

```
// A factory method that creates and starts a thread.
public static NewThread createAndStart() {
    NewThread myThrd = new NewThread();
    myThrd.t.start();
    return myThrd;
}
```

Using `createAndStart()`, you can now replace the sequence

```
NewThread nt = new NewThread(); // create a new thread
nt.t.start(); // Start the thread
```

with

```
NewThread nt = NewThread.createAndStart();
```

Now the thread is created and started in one step.

In cases in which you don't need to keep a reference to the executing thread, you can sometimes create and start a thread with one line of code, without the use of a factory method. For example, again assuming the **ThreadDemo** program, the following creates and starts a **NewThread** thread:

```
new NewThread().t.start();
```

However, in real-world applications, you will usually need to keep a reference to the thread, so the factory method is often a good choice.

## Virtual Threads

As mentioned at the beginning of this chapter, the Java platform implements Java threads as wrappers around the underlying OS's native threads. In this model, when the OS runs out of threads, so does Java! For high-throughput, high-scale applications such as application servers, which generally like to use a new thread for each new incoming request, this can be a performance bottleneck as to the number of requests the server can handle at the same time. JDK 21 has introduced a new way to map Java threads onto OS threads in which many Java threads can share the same OS thread. This is called *virtual threading*. By scheduling work on the OS in a more flexible way, high-scale server applications can perform better because they can handle more concurrent requests gracefully. The number of virtual Java threads an application can create is no longer limited to the number of OS threads available to Java.

Fortunately, the programming model for using platform threads and virtual threads is the same except for how the Java threads are first created and managed when the JVM terminates. All the examples so far have created threads that are platform threads, since these are the kind of threads that work well for most applications. You can determine what kind of thread is in use by the **Thread** class method

**boolean isVirtual()**

which is **false** for platform threads and **true** for virtual threads.

In order to create a virtual thread in Java, you can use the static **Thread** method **ofVirtual()**, from which you can obtain a **Thread.Builder** object from which to create virtual threads. For example, the code

```
Runnable myTask = // a task within the run() method  
Thread myVirtualThread = Thread.ofVirtual().start(myTask);
```

shows a **Runnable** task being run on a virtual thread. The virtual thread is an instance of the Java **Thread** class you already learned about in this chapter, and so it has the same life cycle and API methods for managing execution. It is important to note that if you want to use virtual threads, you have to explicitly choose this API to do so.

The companion to **ofVirtual()** is the static **Thread** method **ofPlatform()**, which creates a **Thread.Builder** object from which you can create traditional platform threads. However, since platform threads are the default type of threads, the APIs that you have already seen create platform threads, such as

```
Runnable myTask = // a task within the run() method  
Thread myThread = new Thread(task, "myTask");
```

or the default no argument constructor of the **Thread** class.

While virtual threads “look like” platform threads for the most part, they differ in that they are not named by default (their name variable is the empty string) and the JVM does not wait for them to finish their work when it exits.

If a virtual thread is waiting, the Java platform can schedule its OS thread to be useful for another virtual thread that needs attention. In this way, the Java platform can use OS threads more efficiently for applications that have tasks spending significant time waiting on network connections or blocking I/O. For applications mostly doing continuous processing, like large algorithmic calculations, virtual threads will likely not bring much benefit. So virtual threads, while easy to use because they “look like” traditional threads, bring performance benefits only for certain types of applications and will likely require judicious use together with performance testing to realize their full potential.

# Using Multithreading

The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads. With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it. Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program! One last point: To create compute-intensive applications that can automatically scale to make use of the available processors in a multicore system, consider using the Fork/Join Framework, which is described in [Chapter 29](#).

## CHAPTER

# 12

# Enumerations, Autoboxing, and Annotations

---

This chapter examines three features that were not originally part of Java, but over time each has become a near indispensable aspect of Java programming: enumerations, autoboxing, and annotations. Originally added by JDK 5, each is a feature upon which Java programmers have come to rely because each offers a streamlined approach to handling common programming tasks. This chapter also discusses Java's type wrappers and introduces reflection.

## Enumerations

In its simplest form, an *enumeration* is a list of named constants that define a new data type and its legal values. Thus, an enumeration object can hold only a value that was declared in the list. Other values are not allowed. In other words, an enumeration gives you a way to explicitly specify the only values that a data type can legally have. Enumerations are commonly used to define a set of values that represent a collection of items. For example, you might use an enumeration to represent the error codes that can result from some operation, such as success, failed, or pending; or a list of the states that a device might be in, such as running, stopped, or paused. In early versions of Java, such values were defined using **final** variables, but enumerations offer a far superior approach.

Although Java enumerations might, at first glance, appear similar to enumerations in other languages, this similarity may be only skin deep

because, in Java, an enumeration defines a class type. By making enumerations into classes, the capabilities of the enumeration are greatly expanded. For example, in Java, an enumeration can have constructors, methods, and instance variables. Because of their power and flexibility, enumerations are widely used throughout the Java API library.

## Enumeration Fundamentals

An enumeration is created using the **enum** keyword. For example, here is a simple enumeration that lists various apple varieties:

```
// An enumeration of apple varieties.  
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap, Cortland  
}
```

The identifiers **Jonathan**, **GoldenDel**, and so on are called *enumeration constants*. Each is implicitly declared as a public, static final member of **Apple**. Furthermore, their type is the type of the enumeration in which they are declared, which is **Apple** in this case. Thus, in the language of Java, these constants are called *self-typed*, in which “self” refers to the enclosing enumeration.

Once you have defined an enumeration, you can create a variable of that type. However, even though enumerations define a class type, you do not instantiate an **enum** using **new**. Instead, you declare and use an enumeration variable in much the same way as you do one of the primitive types. For example, this declares **ap** as a variable of enumeration type **Apple**:

```
Apple ap;
```

Because **ap** is of type **Apple**, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns **ap** the value **RedDel**:

```
ap = Apple.RedDel;
```

Notice that the symbol **RedDel** is preceded by **Apple**.

Two enumeration constants can be compared for equality by using the `=` relational operator. For example, this statement compares the value in `ap` with the **GoldenDel** constant:

```
if (ap == Apple.GoldenDel) // ...
```

An enumeration value can also be used to control a **switch** statement. Of course, all of the **case** statements must use constants from the same **enum** as that used by the **switch** expression. For example, this **switch** is perfectly valid:

```
// Use an enum to control a switch statement.
switch(ap) {
    case Jonathan:
        // ...
    case Winesap:
        // ...
```

Notice that in the **case** statements, the names of the enumeration constants are used without being qualified by their enumeration type name. That is, **Winesap**, not **Apple.Winesap**, is used. This is because the type of the enumeration in the **switch** expression has already implicitly specified the **enum** type of the **case** constants. There is no need to qualify the constants in the **case** statements with their **enum** type name. In fact, attempting to do so will cause a compilation error.

When an enumeration constant is displayed, such as in a **println( )** statement, its name is output. For example, given the statement

```
System.out.println(Apple.Winesap);
```

the name **Winesap** is displayed.

The following program puts together all of the pieces and demonstrates the **Apple** enumeration:

```
// An enumeration of apple varieties.  
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap, Cortland  
}  
  
class EnumDemo {  
    public static void main(String[] args)  
    {  
        Apple ap;  
  
        ap = Apple.RedDel;  
  
        // Output an enum value.  
        System.out.println("Value of ap: " + ap);  
        System.out.println();  
  
        ap = Apple.GoldenDel;  
  
        // Compare two enum values.  
        if(ap == Apple.GoldenDel)  
            System.out.println("ap contains GoldenDel.\n");  
  
        // Use an enum to control a switch statement.  
        switch(ap) {  
            case Jonathan:  
                System.out.println("Jonathan is red.");  
                break;  
            case GoldenDel:  
                System.out.println("Golden Delicious is yellow.");  
                break;  
            case RedDel:  
                System.out.println("Red Delicious is red.");  
                break;  
            case Winesap:  
                System.out.println("Winesap is red.");  
                break;  
            case Cortland:  
                System.out.println("Cortland is red.");  
                break;  
        }  
    }  
}
```

- The output from the program is shown here:

```
Value of ap: RedDel  
ap contains GoldenDel.  
Golden Delicious is yellow.
```

## The values( ) and valueOf( ) Methods

All enumerations automatically contain two predefined methods: **values( )** and **valueOf( )**. Their general forms are shown here:

```
public static enum-type [ ] values()  
public static enum-type valueOf(String str)
```

The **values( )** method returns an array that contains a list of the enumeration constants. The **valueOf( )** method returns the enumeration constant whose value corresponds to the string passed in *str*. In both cases, *enum-type* is the type of the enumeration. For example, in the case of the **Apple** enumeration shown earlier, the return type of **Apple.valueOf("Winesap")** is **Winesap**.

The following program demonstrates the **values( )** and **valueOf( )** methods:

```
// Use the built-in enumeration methods.

// An enumeration of apple varieties.
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
    public static void main(String[] args)
    {
        Apple ap;

        System.out.println("Here are all Apple constants:");

        // use values()
        Apple[] allapples = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);

        System.out.println();

        // use valueOf()
        ap = Apple.valueOf("Winesap");
        System.out.println("ap contains " + ap);
    }
}
```

The output from the program is shown here:

```
Here are all Apple constants:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap contains Winesap
```

Notice that this program uses a for-each style **for** loop to cycle through the array of constants obtained by calling **values()**. For the sake of

illustration, the variable **allapples** was created and assigned a reference to the enumeration array. However, this step is not necessary because the **for** could have been written as shown here, eliminating the need for the **allapples** variable:

```
for(Apple a : Apple.values())
    System.out.println(a);
```

Now, notice how the value corresponding to the name **Winesap** was obtained by calling **valueOf()**:

```
ap = Apple.valueOf("Winesap");
```

As explained, **valueOf()** returns the enumeration value associated with the name of the constant represented as a string.

## Java Enumerations Are Class Types

As mentioned, a Java enumeration is a class type. Although you don't instantiate an **enum** using **new**, it otherwise has much the same capabilities as other classes. The fact that **enum** defines a class gives the Java enumeration extraordinary power. For example, you can give it constructors, add instance variables and methods, and even implement interfaces.

It is important to understand that each enumeration constant is an object of its enumeration type. Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration. For example, consider the following version of **Apple**:

```
// Use an enum constructor, instance variable, and method.
enum Apple {
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

    private int price; // price of each apple

    // Constructor
    Apple(int p) { price = p; }

    int getPrice() { return price; }
}

class EnumDemo3 {
    public static void main(String[] args)
    {
        Apple ap;

        // Display price of Winesap.
        System.out.println("Winesap costs " +
                           Apple.Winesap.getPrice() +
                           " cents.\n");

        // Display all apples and prices.
        System.out.println("All apple prices:");
        for(Apple a : Apple.values())
            System.out.println(a + " costs " + a.getPrice() +
                               " cents.");
    }
}
```

The output is shown here:

```
Winesap costs 15 cents.

All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

This version of **Apple** adds three things. The first is the instance variable **price**, which is used to hold the price of each variety of apple. The second is the **Apple** constructor, which is passed the price of an apple. The third is the method **getPrice()**, which returns the value of **price**.

When the variable **ap** is declared in **main()**, the constructor for **Apple** is called once for each constant that is specified. Notice how the arguments to the constructor are specified, by putting them inside parentheses after each constant, as shown here:

```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15),  
Cortland(8);
```

These values are passed to the **p** parameter of **Apple()**, which then assigns this value to **price**. Again, the constructor is called once for each constant.

Because each enumeration constant has its own copy of **price**, you can obtain the price of a specified type of apple by calling **getPrice()**. For example, in **main()**, the price of a Winesap is obtained by the following call:

```
Apple.Winesap.getPrice()
```

The prices of all varieties are obtained by cycling through the enumeration using a **for** loop. Because there is a copy of **price** for each enumeration constant, the value associated with one constant is separate and distinct from the value associated with another constant. This is a powerful concept, which is only available when enumerations are implemented as classes, as Java does.

Although the preceding example contains only one constructor, an **enum** can offer two or more overloaded forms, just as can any other class. For example, this version of **Apple** provides a default constructor that initializes the price to **-1**, to indicate that no price data is available:

```
// Use an enum constructor.  
enum Apple {  
    Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);  
  
    private int price; // price of each apple  
  
    // Constructor  
    Apple(int p) { price = p; }  
  
    // Overloaded constructor  
    Apple() { price = -1; }  
  
    int getPrice() { return price; }  
}
```

Notice that in this version, **RedDel** is not given an argument. This means that the default constructor is called, and **RedDel**'s price variable is given the value **-1**.

Here are two restrictions that apply to enumerations. First, an enumeration can't inherit another class. Second, an **enum** cannot be a superclass. This means that an **enum** can't be extended. Otherwise, **enum** acts much like any other class type. The key is to remember that each of the enumeration constants is an object of the class in which it is defined.

## Enumerations Inherit Enum

Although you can't inherit a superclass when declaring an **enum**, all enumerations automatically inherit one: **java.lang.Enum**. This class defines several methods that are available for use by all enumerations. The **Enum** class is described in detail in Part II, but three of its methods warrant a discussion at this time.

You can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its *ordinal value*, and it is retrieved by calling the **ordinal()** method, shown here:

```
final int ordinal()
```

It returns the ordinal value of the invoking constant. Ordinal values begin at zero. Thus, in the **Apple** enumeration, **Jonathan** has an ordinal value of 0,

**GoldenDel** has an ordinal value of 1, **RedDel** has an ordinal value of 2, and so on.

You can compare the ordinal value of two constants of the same enumeration by using the **compareTo( )** method. It has this general form:

```
final int compareTo(enum-type e)
```

Here, *enum-type* is the type of the enumeration, and *e* is the constant being compared to the invoking constant. Remember, both the invoking constant and *e* must be of the same enumeration. If the invoking constant has an ordinal value less than *e*'s, then **compareTo( )** returns a negative value. If the two ordinal values are the same, then zero is returned. If the invoking constant has an ordinal value greater than *e*'s, then a positive value is returned.

You can compare for equality an enumeration constant with any other object by using **equals( )**, which overrides the **equals( )** method defined by **Object**. Although **equals( )** can compare an enumeration constant to any other object, those two objects will be equal only if they both refer to the same constant, within the same enumeration. Simply having ordinal values in common will not cause **equals( )** to return true if the two constants are from different enumerations.

Remember, you can compare two enumeration references for equality by using `= =`.

The following program demonstrates the **ordinal( )**, **compareTo( )**, and **equals( )** methods:

```
// Demonstrate ordinal(), compareTo(), and equals().  
  
// An enumeration of apple varieties.  
enum Apple {  
    Jonathan, GoldenDel, RedDel, Winesap, Cortland  
}  
  
class EnumDemo4 {  
    public static void main(String[] args)  
    {  
        Apple ap, ap2, ap3;  
  
        // Obtain all ordinal values using ordinal().  
        System.out.println("Here are all apple constants" +  
                           " and their ordinal values: ");  
        for(Apple a : Apple.values())  
            System.out.println(a + " " + a.ordinal());  
  
        ap = Apple.RedDel;  
        ap2 = Apple.GoldenDel;  
        ap3 = Apple.RedDel;  
  
        System.out.println();  
  
        // Demonstrate compareTo() and equals()  
        if(ap.compareTo(ap2) < 0)  
            System.out.println(ap + " comes before " + ap2);  
  
        if(ap.compareTo(ap2) > 0)  
            System.out.println(ap2 + " comes before " + ap);  
  
        if(ap.compareTo(ap3) == 0)  
            System.out.println(ap + " equals " + ap3);  
  
        System.out.println();  
  
        if(ap.equals(ap2))  
            System.out.println("Error!");  
  
        if(ap.equals(ap3))  
            System.out.println(ap + " equals " + ap3);
```

```
    if(ap == ap3)
        System.out.println(ap + " == " + ap3);

    }
}
```

The output from the program is shown here:

```
Here are all apple constants and their ordinal values:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel comes before RedDel
RedDel equals RedDel

RedDel equals RedDel
RedDel == RedDel
```

## Another Enumeration Example

Before moving on, we will look at a different example that uses an **enum**. In [Chapter 9](#), we created an automated “decision maker” program. In that version, variables called **NO**, **YES**, **MAYBE**, **LATER**, **SOON**, and **NEVER** were declared within an interface and used to represent the possible answers. While there is nothing technically wrong with that approach, using an enumeration is a better choice. Here is an improved version of that program that uses an **enum** called **Answers** to define the answers. You should compare this version to the original in [Chapter 9](#).

```
// An improved version of the "Decision Maker"
// program from Chapter 9. This version uses an
// enum, rather than interface variables, to
// represent the answers.

import java.util.Random;

// An enumeration of the possible answers.
enum Answers {
    NO, YES, MAYBE, LATER, SOON, NEVER
}

class Question {
    Random rand = new Random();
    Answers ask() {
        int prob = (int) (100 * rand.nextDouble());

        if (prob < 15)
            return Answers.MAYBE; // 15%
        else if (prob < 30)
            return Answers.NO; // 15%
        else if (prob < 60)
            return Answers.YES; // 30%
```

```
        else if (prob < 75)
            return Answers.LATER; // 15%
        else if (prob < 98)
            return Answers.SOON; // 13%
        else
            return Answers.NEVER; // 2%
    }
}

class AskMe {
    static void answer(Answers result) {
        switch(result) {
            case NO:
                System.out.println("No");
                break;
            case YES:
                System.out.println("Yes");
                break;
            case MAYBE:
                System.out.println("Maybe");
                break;
            case LATER:
                System.out.println("Later");
                break;
            case SOON:
                System.out.println("Soon");
                break;
            case NEVER:
                System.out.println("Never");
                break;
        }
    }

    public static void main(String[] args) {
        Question q = new Question();
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}
```

# Type Wrappers

As you know, Java uses primitive types (also called simple types), such as **int** or **double**, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, you can't pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides *type wrappers*, which are classes that encapsulate a primitive type within an object. The type wrapper classes are described in detail in Part II, but they are introduced here because they relate directly to Java's autoboxing feature.

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy. Each is briefly examined next.

## Character

**Character** is a wrapper around a **char**. The constructor for **Character** is

`Character(char ch)`

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

However, beginning with JDK 9, the **Character** constructor was deprecated, and beginning with JDK 16, it has been deprecated for removal. Today, it is strongly recommended that you use the static method **valueOf()** to obtain a **Character** object. It is shown here:

`static Character valueOf(char ch)`

It returns a **Character** object that wraps *ch*.

To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here:

```
char charValue()
```

It returns the encapsulated character.

## Boolean

**Boolean** is a wrapper around **boolean** values. It defines these constructors:

```
Boolean(boolean boolValue)  
Boolean(String boolString)
```

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

However, beginning with JDK 9, the **Boolean** constructors were deprecated, and beginning with JDK 16, they have been deprecated for removal. Today, it is strongly recommended that you use the static method **valueOf( )** to obtain a **Boolean** object. It has the two versions shown here:

```
static Boolean valueOf(boolean boolValue)  
static Boolean valueOf(String boolString)
```

Each returns a **Boolean** object that wraps the indicated value.

To obtain a **boolean** value from a **Boolean** object, use **booleanValue( )**, shown here:

```
boolean booleanValue()
```

It returns the **boolean** equivalent of the invoking object.

## The Numeric Type Wrappers

By far, the most commonly used type wrappers are those that represent numeric values. These are **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**.

All of the numeric type wrappers inherit the abstract class **Number**. **Number** declares methods that return the value of an object in each of the different number formats. These methods are shown here:

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

For example, **doubleValue( )** returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for **Integer**:

```
Integer(int num)
Integer(String str)
```

If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.

However, beginning with JDK 9, the numeric type-wrapper constructors were deprecated, and beginning with JDK 16, they have been deprecated for removal. Today, it is strongly recommended that you use one of the **valueOf( )** methods to obtain a wrapper object. The **valueOf( )** method is a static member of all of the numeric wrapper classes, and all numeric classes support forms that convert a numeric value or a string into an object. For example, here are two of the forms supported by **Integer**:

```
static Integer valueOf(int val)
static Integer valueOf(String valStr) throws NumberFormatException
```

Here, *val* specifies an integer value and *valStr* specifies a string that represents a properly formatted numeric value in string form. Each returns an **Integer** object that wraps the specified value. Here is an example:

```
Integer iOb = Integer.valueOf(100);
```

After this statement executes, the value 100 is represented by an **Integer** instance. Thus, **iOb** wraps the value 100 within an object. In addition to the forms of **valueOf()** just shown, the integer wrappers, **Byte**, **Short**, **Integer**, and **Long**, also supply a form that lets you specify a radix.

All of the type wrappers override **toString()**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println()**, for example, without having to convert it into its primitive type.

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value:

```
// Demonstrate a type wrapper.
class Wrap {
    public static void main(String[] args) {

        Integer iOb = Integer.valueOf(100);

        int i = iOb.intValue();

        System.out.println(i + " " + iOb); // displays 100 100
    }
}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue()** and stores the result in **i**.

The process of encapsulating a value within an object is called *boxing*. Thus, in the program, this line boxes the value 100 into an **Integer**:

```
Integer iOb = Integer.valueOf(100);
```

The process of extracting a value from a type wrapper is called *unboxing*. For example, the program unboxes the value in **iOb** with this statement:

```
int i = iOb.intValue();
```

The same general procedure used by the preceding program to box and unbox values has been available for use since the original version of Java. However, today, Java provides a more streamlined approach, which is described next.

## Autoboxing

Modern versions of Java have included two important features: *autoboxing* and *auto-unboxing*. Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue()** or **doubleValue()**.

Autoboxing and auto-unboxing greatly streamline the coding of several algorithms, removing the tedium of manually boxing and unboxing values. They also help prevent errors. Moreover, they are very important to generics, which operate only on objects. Finally, autoboxing makes working with the Collections Framework (described in Part II) much easier.

With autoboxing, it is not necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct an **Integer** object that has the value 100:

```
Integer iOb = 100; // autobox an int
```

Notice that the object is not explicitly boxed. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox **iOb**, you can use this line:

```
int i = iOb; // auto-unbox
```

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

```
// Demonstrate autoboxing/unboxing.
class AutoBox {
    public static void main(String[] args) {
        Integer iOb = 100; // autobox an int
        int i = iOb; // auto-unbox
        System.out.println(i + " " + iOb); // displays 100 100
    }
}
```

## Autoboxing and Methods

In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method. For example, consider this:

```
// Autoboxing/unboxing takes place with
// method parameters and return values.

class AutoBox2 {
    // Take an Integer parameter and return
    // an int value;
    static int m(Integer v) {
        return v ; // auto-unbox to int
    }

    public static void main(String[] args) {
        // Pass an int to m() and assign the return value
        // to an Integer. Here, the argument 100 is autoboxed
        // into an Integer. The return value is also autoboxed
        // into an Integer.
        Integer iOb = m(100);

        System.out.println(iOb);
    }
}
```

This program displays the following result:

```
100
```

In the program, notice that **m()** specifies an **Integer** parameter and returns an **int** result. Inside **main()**, **m()** is passed the value 100. Because **m()** is expecting an **Integer**, this value is automatically boxed. Then, **m()** returns the **int** equivalent of its argument. This causes **v** to be auto-unboxed. Next, this **int** value is assigned to **iOb** in **main()**, which causes the **int** return value to be autoboxed.

## Autoboxing/Unboxing Occurs in Expressions

In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
// Autoboxing/unboxing occurs inside expressions.

class AutoBox3 {
    public static void main(String[] args) {

        Integer iOb, iOb2;
        int i;

        iOb = 100;
        System.out.println("Original value of iOb: " + iOb);

        // The following automatically unboxes iOb,
        // performs the increment, and then reboxes
        // the result back into iOb.
        ++iOb;
        System.out.println("After ++iOb: " + iOb);

        // Here, iOb is unboxed, the expression is
        // evaluated, and the result is reboxed and
        // assigned to iOb2.
        iOb2 = iOb + (iOb / 3);
        System.out.println("iOb2 after expression: " + iOb2);

        // The same expression is evaluated, but the
        // result is not reboxed.
        i = iOb + (iOb / 3);
        System.out.println("i after expression: " + i);

    }
}
```

The output is shown here:

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

In the program, pay special attention to this line:

```
++iOb;
```

This causes the value in **iOb** to be incremented. It works like this: **iOb** is unboxed, the value is incremented, and the result is reboxed.

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

```
class AutoBox4 {  
    public static void main(String[] args) {  
  
        Integer iOb = 100;  
        Double dOb = 98.6;  
  
        dOb = dOb + iOb;  
        System.out.println("dOb after expression: " + dOb);  
    }  
}
```

The output is shown here:

```
dOb after expression: 198.6
```

As you can see, both the **Double** object **dOb** and the **Integer** object **iOb** participated in the addition, and the result was reboxed and stored in **dOb**.

Because of auto-unboxing, you can use **Integer** numeric objects to control a **switch** statement. For example, consider this fragment:

```
Integer iOb = 2;  
  
switch(iOb) {  
    case 1: System.out.println("one");  
    break;  
    case 2: System.out.println("two");  
    break;  
    default: System.out.println("error");  
}
```

When the **switch** expression is evaluated, **iOb** is unboxed and its **int** value is obtained.

As the examples in the programs show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy. In the early days of Java, such code would have involved casts and calls to methods such as `intValue()`.

## Autoboxing/Unboxing Boolean and Character Values

As described earlier, Java also supplies wrappers for `boolean` and `char`. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.

class AutoBox5 {
    public static void main(String[] args) {

        // Autobox/unbox a boolean.
        Boolean b = true;

        // Below, b is auto-unboxed when used in
        // a conditional expression, such as an if.
        if(b) System.out.println("b is true");

        // Autobox/unbox a char.
        Character ch = 'x'; // box a char
        char ch2 = ch; // unbox a char

        System.out.println("ch2 is " + ch2);
    }
}
```

The output is shown here:

```
b is true
ch2 is x
```

The most important thing to notice about this program is the auto-unboxing of **b** inside the **if** conditional expression. As you should recall, the

conditional expression that controls an **if** must evaluate to type **boolean**. Because of auto-unboxing, the **boolean** value contained within **b** is automatically unboxed when the conditional expression is evaluated. Thus, with autoboxing/unboxing, a **Boolean** object can be used to control an **if** statement.

Because of auto-unboxing, a **Boolean** object can now also be used to control any of Java's loop statements. When a **Boolean** is used as the conditional expression of a **while**, **for**, or **do/while**, it is automatically unboxed into its **boolean** equivalent. For example, this is perfectly valid code:

```
Boolean b;  
// ...  
while(b) { // ...
```

## Autoboxing/Unboxing Helps Prevent Errors

In addition to the convenience that it offers, autoboxing/unboxing can also help prevent errors. For example, consider the following program:

```
// An error produced by manual unboxing.  
class UnboxingError {  
    public static void main(String[] args) {  
  
        Integer iOb = 1000; // autobox the value 1000  
  
        int i = iOb.byteValue(); // manually unbox as byte !!!  
  
        System.out.println(i); // does not display 1000 !  
    }  
}
```

This program displays not the expected value of 1000, but –24! The reason is that the value inside **iOb** is manually unboxed by calling **byteValue()**, which causes the truncation of the value stored in **iOb**, which is 1000. This results in the garbage value of –24 being assigned to **i**. Auto-unboxing prevents this type of error because the value in **iOb** will always auto-unbox into a value compatible with **int**.

In general, because autoboxing always creates the proper object, and auto-unboxing always produces the proper value, there is no way for the process to produce the wrong type of object or value. In the rare instances where you want a type different than that produced by the automated process, you can still manually box and unbox values. Of course, the benefits of autoboxing/unboxing are lost. In general, you should employ autoboxing/unboxing. It is the way that modern Java code is written.

## A Word of Warning

Because of autoboxing and auto-unboxing, some might be tempted to use objects such as **Integer** or **Double** exclusively, abandoning primitives altogether. For example, with autoboxing/unboxing, it is possible to write code like this:

```
// A bad use of autoboxing/unboxing!
Double a, b, c;

a = 10.0;
b = 4.0;

c = Math.sqrt(a*a + b*b);

System.out.println("Hypotenuse is " + c);
```

In this example, objects of type **Double** hold values that are used to calculate the hypotenuse of a right triangle. Although this code is technically correct and does, in fact, work properly, it is a very bad use of autoboxing/unboxing. It is far less efficient than the equivalent code written using the primitive type **double**. The reason is that each autobox and auto-unbox adds overhead that is not present if the primitive type is used.

In general, you should restrict your use of the type wrappers to only those cases in which an object representation of a primitive type is required. Autoboxing/unboxing was not added to Java as a “back door” way of eliminating the primitive types.

## Annotations

Java provides a feature that enables you to embed supplemental information into a source file. This information, called an *annotation*, does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged. However, this information can be used by various tools during both development and deployment. For example, an annotation might be processed by a source-code generator. The term *metadata* is also used to refer to this feature, but the term *annotation* is the most descriptive and more commonly used.

## Annotation Basics

An annotation is created through a mechanism based on the **interface**. Let's begin with an example. Here is the declaration for an annotation called **MyAnno**:

```
// A simple annotation type.  
@interface MyAnno {  
    String str();  
    int val();  
}
```

First, notice the @ that precedes the keyword **interface**. This tells the compiler that an annotation type is being declared. Next, notice the two members **str()** and **val()**. All annotations consist solely of method declarations. However, you don't provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields, as you will see.

An annotation cannot include an **extends** clause. However, all annotation types automatically extend the **Annotation** interface. Thus, **Annotation** is a super-interface of all annotations. It is declared within the **java.lang.annotation** package. It overrides **hashCode()**, **equals()**, and **toString()**, which are defined by **Object**. It also specifies **annotationType()**, which returns a **Class** object that represents the invoking annotation.

Once you have declared an annotation, you can use it to annotate something. Initially, annotations could be used only on declarations, and that is where we will begin. (JDK 8 added the ability to annotate type use, and this is described later in this chapter. However, the same basic

techniques apply to both kinds of annotations.) Any type of declaration can have an annotation associated with it. For example, classes, methods, fields, parameters, and **enum** constants can be annotated. Even an annotation can be annotated. In all cases, the annotation precedes the rest of the declaration.

When you apply an annotation, you give values to its members. For example, here is an example of **MyAnno** being applied to a method declaration:

```
// Annotate a method.  
@MyAnno(str = "Annotation Example", val = 100)  
public static void myMeth() { // ...}
```

This annotation is linked with the method **myMeth()**. Look closely at the annotation syntax. The name of the annotation, preceded by an **@**, is followed by a parenthesized list of member initializations. To give a member a value, that member's name is assigned a value. Therefore, in the example, the string "Annotation Example" is assigned to the **str** member of **MyAnno**. Notice that no parentheses follow **str** in this assignment. When an annotation member is given a value, only its name is used. Thus, annotation members look like fields in this context.

## Specifying a Retention Policy

Before exploring annotations further, it is necessary to discuss *annotation retention policies*. A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within the **java.lang.annotation.RetentionPolicy** enumeration. They are **SOURCE**, **CLASS**, and **RUNTIME**.

An annotation with a retention policy of **SOURCE** is retained only in the source file and is discarded during compilation.

An annotation with a retention policy of **CLASS** is stored in the **.class** file during compilation. However, it is not available through the JVM during run time.

An annotation with a retention policy of **RUNTIME** is stored in the **.class** file during compilation and is available through the JVM during run time. Thus, **RUNTIME** retention offers the greatest annotation persistence.

---

**NOTE** An annotation on a local variable declaration is not retained in the `.class` file.

A retention policy is specified for an annotation by using one of Java's built-in annotations: `@Retention`. Its general form is shown here:

`@Retention(retention-policy)`

Here, *retention-policy* must be one of the previously discussed enumeration constants. If no retention policy is specified for an annotation, then the default policy of **CLASS** is used.

The following version of **MyAnno** uses `@Retention` to specify the **RUNTIME** retention policy. Thus, **MyAnno** will be available to the JVM during program execution.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}
```

## Obtaining Annotations at Run Time by Use of Reflection

Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of **RUNTIME**, then they can be queried at run time by any Java program through the use of *reflection*. Reflection is the feature that enables information about a class to be obtained at run time. The reflection API is contained in the **java.lang.reflect** package. There are a number of ways to use reflection, and we won't examine them all here. We will, however, walk through a few examples that apply to annotations.

The first step to using reflection is to obtain a **Class** object that represents the class whose annotations you want to obtain. **Class** is one of Java's built-in classes and is defined in **java.lang**. It is described in detail in Part II. There are various ways to obtain a **Class** object. One of the easiest is to call **getClass()**, which is a method defined by **Object**. Its general form is shown here:

```
final Class<?> getClass()
```

It returns the **Class** object that represents the invoking object.

---

**NOTE** Notice the <?> that follows **Class** in the declaration of **getClass( )** just shown. This is related to Java's generics feature. **getClass( )** and several other reflection-related methods discussed in this chapter make use of generics. Generics are described in [Chapter 14](#). However, an understanding of generics is not needed to grasp the fundamental principles of reflection.

After you have obtained a **Class** object, you can use its methods to obtain information about the various items declared by the class, including its annotations. If you want to obtain the annotations associated with a specific item declared within a class, you must first obtain an object that represents that item. For example, **Class** supplies (among others) the **getMethod( )**, **getField( )**, and **getConstructor( )** methods, which obtain information about a method, field, and constructor, respectively. These methods return objects of type **Method**, **Field**, and **Constructor**.

To understand the process, let's work through an example that obtains the annotations associated with a method. To do this, you first obtain a **Class** object that represents the class, and then you call **getMethod( )** on that **Class** object, specifying the name of the method. **getMethod( )** has this general form:

```
Method getMethod(String methName, Class<?> ... paramTypes)
```

The name of the method is passed in *methName*. If the method has arguments, then **Class** objects representing those types must also be specified by *paramTypes*. Notice that *paramTypes* is a varargs parameter. This means that you can specify as many parameter types as needed, including zero. **getMethod( )** returns a **Method** object that represents the method. If the method can't be found, **NoSuchMethodException** is thrown.

From a **Class**, **Method**, **Field**, or **Constructor** object, you can obtain a specific annotation associated with that object by calling **getAnnotation( )**. Its general form is shown here:

```
<A extends Annotation> getAnnotation(Class<A> annoType)
```

Here, *annoType* is a **Class** object that represents the annotation in which you are interested. The method returns a reference to the annotation. Using this reference, you can obtain the values associated with the annotation's members. The method returns **null** if the annotation is not found, which will be the case if the annotation does not have **RUNTIME** retention.

Here is a program that assembles all of the pieces shown earlier and uses reflection to display the annotation associated with a method:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {

    // Annotate a method.
    @MyAnno(str = "Annotation Example", val = 100)
    public static void myMeth() {
        Meta ob = new Meta();

        // Obtain the annotation for this method
        // and display the values of the members.
    }
}
```

```

try {
    // First, get a Class object that represents
    // this class.
    Class<?> c = ob.getClass();

    // Now, get a Method object that represents
    // this method.
    Method m = c.getMethod("myMeth");

    // Next, get the annotation for this class.
    MyAnno anno = m.getAnnotation(MyAnno.class);

    // Finally, display the values.
    System.out.println(anno.str() + " " + anno.val());
} catch (NoSuchMethodException exc) {
    System.out.println("Method Not Found.");
}
}

public static void main(String[] args) {
    myMeth();
}
}

```

The output from the program is shown here:

Annotation Example 100

This program uses reflection as described to obtain and display the values of **str** and **val** in the **MyAnno** annotation associated with **myMeth()** in the **Meta** class. There are two things to pay special attention to. First, in the line

```
MyAnno anno = m.getAnnotation(MyAnno.class);
```

notice the expression **MyAnno.class**. This expression evaluates to a **Class** object of type **MyAnno**, the annotation. This construct is called a *class literal*. You can use this type of expression whenever a **Class** object of a known class is needed. For example, this statement could have been used to obtain the **Class** object for **Meta**:

```
Class<?> c = Meta.class;
```

Of course, this approach only works when you know the class name of an object in advance, which might not always be the case. In general, you can obtain a class literal for classes, interfaces, primitive types, and arrays. (Remember, the `<?>` syntax relates to Java's generics feature. It is described in [Chapter 14](#).)

The second point of interest is the way the values associated with `str` and `val` are obtained when they are output by the following line:

```
System.out.println(anno.str() + " " + anno.val());
```

Notice that they are invoked using the method-call syntax. This same approach is used whenever the value of an annotation member is required.

## A Second Reflection Example

In the preceding example, `myMeth()` has no parameters. Thus, when `getMethod()` was called, only the name `myMeth` was passed. However, to obtain a method that has parameters, you must specify class objects representing the types of those parameters as arguments to `getMethod()`. For example, here is a slightly different version of the preceding program:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {

    // myMeth now has two arguments.
    @MyAnno(str = "Two Parameters", val = 19)
    public static void myMeth(String str, int i)
    {
        Meta ob = new Meta();

        try {
            Class<?> c = ob.getClass();

            // Here, the parameter types are specified.
            Method m = c.getMethod("myMeth", String.class, int.class);

            MyAnno anno = m.getAnnotation(MyAnno.class);

            System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

    public static void main(String[] args) {
        myMeth("test", 10);
    }
}
```

The output from this version is shown here:

Two Parameters 19

In this version, **myMeth( )** takes a **String** and an **int** parameter. To obtain information about this method, **getMethod( )** must be called as shown here:

```
Method m = c.getMethod("myMeth", String.class, int.class);
```

Here, the **Class** objects representing **String** and **int** are passed as additional arguments.

## Obtaining All Annotations

You can obtain all annotations that have **RUNTIME** retention that are associated with an item by calling **getAnnotations( )** on that item. It has this general form:

```
Annotation[ ] getAnnotations()
```

It returns an array of the annotations. **getAnnotations( )** can be called on objects of type **Class**, **Method**, **Constructor**, and **Field**, among others.

Here is another reflection example that shows how to obtain all annotations associated with a class and with a method. It declares two annotations. It then uses those annotations to annotate a class and a method.

```
// Show all annotations for a class and a method.
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
    String description();
}

@What(description = "An annotation test class")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {

    @What(description = "An annotation test method")
    @MyAnno(str = "Testing", val = 100)
    public static void myMeth() {
        Meta2 ob = new Meta2();

        try {
            Annotation[] annos = ob.getClass().getAnnotations();

            // Display all annotations for Meta2.
            System.out.println("All annotations for Meta2:");
            for(Annotation a : annos)
                System.out.println(a);

            System.out.println();

            // Display all annotations for myMeth.
            Method m = ob.getClass( ).getMethod("myMeth");
            annos = m.getAnnotations();
        }
    }
}
```

```
        System.out.println("All annotations for myMeth:");
        for(Annotation a : annos)
            System.out.println(a);

    } catch (NoSuchMethodException exc) {
        System.out.println("Method Not Found.");
    }
}

public static void main(String[] args) {
    myMeth();
}
}
```

The output is shown here:

```
All annotations for Meta2:
@What(description=An annotation test class)
@MyAnno(str=Meta2, val=99)
```

```
All annotations for myMeth:
@What(description=An annotation test method)
@MyAnno(str=Testing, val=100)
```

The program uses **getAnnotations()** to obtain an array of all annotations associated with the **Meta2** class and with the **myMeth()** method. As explained, **getAnnotations()** returns an array of **Annotation** objects. Recall that **Annotation** is a super-interface of all annotation interfaces and that it overrides **toString()** in **Object**. Thus, when a reference to an **Annotation** is output, its **toString()** method is called to generate a string that describes the annotation, as the preceding output shows.

## The AnnotatedElement Interface

The methods **getAnnotation()** and **getAnnotations()** used by the preceding examples are defined by the **AnnotatedElement** interface, which is defined in **java.lang.reflect**. This interface supports reflection for

annotations and is implemented by the classes **Method**, **Field**, **Constructor**, **Class**, and **Package**, among others.

In addition to **getAnnotation( )** and **getAnnotations( )**, **AnnotatedElement** defines several other methods. Two have been available since annotations were initially added to Java. The first is **getDeclaredAnnotations( )**, which has this general form:

```
Annotation[ ] getDeclaredAnnotations( )
```

It returns all non-inherited annotations present in the invoking object. The second is **isAnnotationPresent( )**, which has this general form:

```
default boolean isAnnotationPresent(Class<? extends Annotation>  
annoType)
```

It returns **true** if the annotation specified by *annoType* is associated with the invoking object. It returns **false** otherwise. To these, JDK 8 added **getDeclaredAnnotation( )**, **getAnnotationsByType( )**, and **getDeclaredAnnotationsByType( )**. Of these, the last two automatically work with a repeated annotation.(Repeated annotations are discussed at the end of this chapter.)

## Using Default Values

You can give annotation members default values that will be used if no value is specified when the annotation is applied. A default value is specified by adding a **default** clause to a member's declaration. It has this general form:

```
type member( ) default value ;
```

Here, *value* must be of a type compatible with *type*.

Here is **@MyAnno** rewritten to include default values:

```
// An annotation type declaration that includes defaults.  
@Retention(RetentionPolicy.RUNTIME)  
@interface MyAnno {  
    String str() default "Testing";  
    int val() default 9000;  
}
```

This declaration gives a default value of "Testing" to **str** and 9000 to **val**. This means that neither value needs to be specified when **@MyAnno** is used. However, either or both can be given values if desired. Therefore, following are the four ways that **@MyAnno** can be used:

```
@MyAnno() // both str and val default  
@MyAnno(str = "some string") // val defaults  
@MyAnno(val = 100) // str defaults  
@MyAnno(str = "Testing", val = 100) // no defaults
```

The following program demonstrates the use of default values in an annotation:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration that includes defaults.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str() default "Testing";
    int val() default 9000;
}

class Meta3 {

    // Annotate a method using the default values.
    @MyAnno()
    public static void myMeth() {
        Meta3 ob = new Meta3();

        // Obtain the annotation for this method
        // and display the values of the members.
        try {
            Class<?> c = ob.getClass();

            Method m = c.getMethod("myMeth");

```



The output is shown here:

```
Testing 9000
```

## Marker Annotations

A *marker* annotation is a special kind of annotation that contains no members. Its sole purpose is to mark an item. Thus, its presence as an annotation is sufficient. The best way to determine if a marker annotation is present is to use the method **isAnnotationPresent()**, which is defined by the **AnnotatedElement** interface.

Here is an example that uses a marker annotation. Because a marker interface contains no members, simply determining whether it is present or absent is sufficient.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A marker annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {

    // Annotate a method using a marker.
    // Notice that no ( ) is needed.
    @MyMarker
    public static void myMeth() {
        Marker ob = new Marker();

        try {
            Method m = ob.getClass().getMethod("myMeth");

            // Determine if the annotation is present.
            if(m.isAnnotationPresent(MyMarker.class))
                System.out.println("MyMarker is present.");

        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

    public static void main(String[] args) {
        myMeth();
    }
}
```

The output, shown here, confirms that **@MyMarker** is present:

```
MyMarker is present.
```

In the program, notice that you do not need to follow **@MyMarker** with parentheses when it is applied. Thus, **@MyMarker** is applied simply by using its name, like this:

`@MyMarker`

It is not wrong to supply an empty set of parentheses, but they are not needed.

## Single-Member Annotations

A *single-member* annotation contains only one member. It works like a normal annotation, except that it allows a shorthand form of specifying the value of the member. When only one member is present, you can simply specify the value for that member when the annotation is applied—you don't need to specify the name of the member. However, in order to use this shorthand, the name of the member must be **value**.

Here is an example that creates and uses a single-member annotation:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A single-member annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
    int value(); // this variable name must be value
}

class Single {

    // Annotate a method using a single-member annotation.
    @MySingle(100)
    public static void myMeth() {
        Single ob = new Single();

        try {
            Method m = ob.getClass().getMethod("myMeth");

            MySingle anno = m.getAnnotation(MySingle.class);

            System.out.println(anno.value()); // displays 100
        } catch (NoSuchMethodException exc) {
            System.out.println("Method Not Found.");
        }
    }

    public static void main(String[] args) {
        myMeth();
    }
}
```

As expected, this program displays the value 100. In the program, **@MySingle** is used to annotate **myMeth()**, as shown here:

```
@MySingle(100)
```

Notice that **value** = need not be specified.

You can use the single-value syntax when applying an annotation that has other members, but those other members must all have default values. For example, here the value **xyz** is added, with a default value of zero:

```
@interface SomeAnno {  
    int value();  
    int xyz() default 0;  
}
```

In cases in which you want to use the default for **xyz**, you can apply **@SomeAnno**, as shown next, by simply specifying the value of **value** by using the single-member syntax:

```
@SomeAnno (88)
```

In this case, **xyz** defaults to zero, and **value** gets the value 88. Of course, to specify a different value for **xyz** requires that both members be explicitly named, as shown here:

```
@SomeAnno (value = 88, xyz = 99)
```

Remember, whenever you are using a single-member annotation, the name of that member must be **value**.

## The Built-In Annotations

Java defines many built-in annotations. Most are specialized, but nine are general purpose. Of these, four are imported from **java.lang.annotation**: **@Retention**, **@Documented**, **@Target**, and **@Inherited**. Five—**@Override**, **@Deprecated**, **@FunctionalInterface**, **@SafeVarargs**, and **@SuppressWarnings**—are included in **java.lang**. Each is described here.

---

**NOTE** **java.lang.annotation** also includes the annotations **Repeatable** and **Native**. **Repeatable** supports repeatable annotations, as described later in this chapter. **Native** annotates a field that can be accessed by native code.

### **@Retention**

**@Retention** is designed to be used only as an annotation to another annotation. It specifies the retention policy as described earlier in this

chapter.

## **@Documented**

The **@Documented** annotation is a marker interface that tells a tool that an annotation is to be documented. It is designed to be used only as an annotation to an annotation declaration.

## **@Target**

The **@Target** annotation specifies the types of items to which an annotation can be applied. It is designed to be used only as an annotation to another annotation. **@Target** takes one argument, which is an array of constants of the **ElementType** enumeration. This argument specifies the types of declarations to which the annotation can be applied. The constants are shown here along with the type of declaration to which they correspond:

Target Constant	Annotation Can Be Applied To
ANNOTATION_TYPE	Another annotation
CONSTRUCTOR	Constructor
FIELD	Field
LOCAL_VARIABLE	Local variable
METHOD	Method
MODULE	Module
PACKAGE	Package
PARAMETER	Parameter
RECORD_COMPONENT	A component of a record (added by JDK 16)
TYPE	Class, interface, or enumeration
TYPE_PARAMETER	Type parameter
TYPE_USE	Type use

You can specify one or more of these values in a **@Target** annotation. To specify multiple values, you must specify them within a braces-

delimited list. For example, to specify that an annotation applies only to fields and local variables, you can use this **@Target** annotation:

```
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```

If you don't use **@Target**, then the annotation can be used on any declaration. For this reason, it is often a good idea to explicitly specify the target or targets so as to clearly indicate the intended uses of an annotation.

## **@Inherited**

**@Inherited** is a marker annotation that can be used only on another annotation declaration. Furthermore, it affects only annotations that will be used on class declarations. **@Inherited** causes the annotation for a superclass to be inherited by a subclass. Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with **@Inherited**, then that annotation will be returned.

## **@Override**

**@Override** is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a superclass. If it doesn't, a compile-time error will result. It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

## **@Deprecated**

**@Deprecated** indicates that a declaration is obsolete and not recommended for use. Beginning with JDK 9, **@Deprecated** also allows you to specify the Java version in which the deprecation occurred and whether the deprecated element is slated for removal.

## **@FunctionalInterface**

**@FunctionalInterface** is a marker annotation designed for use on interfaces. It indicates that the annotated interface is a functional interface. A *functional interface* is an interface that contains one and only one abstract method. Functional interfaces are used by lambda expressions.

(See [Chapter 15](#) for details on functional interfaces and lambda expressions.) If the annotated interface is not a functional interface, a compilation error will be reported. It is important to understand that **@FunctionalInterface** is not needed to create a functional interface. Any interface with exactly one abstract method is, by definition, a functional interface. Thus, **@FunctionalInterface** is purely informational.

## **@SafeVarargs**

**@SafeVarargs** is a marker annotation that can be applied to methods and constructors. It indicates that no unsafe actions related to a varargs parameter occur. It is used to suppress unchecked warnings on otherwise safe code as it relates to non-reifiable vararg types and parameterized array instantiation. (A non-reifiable type is, essentially, a generic type. Generics are described in [Chapter 14](#).) It must be applied only to vararg methods or constructors. Methods must also be **static**, **final**, or **private**.

## **@SuppressWarnings**

**@SuppressWarnings** specifies that one or more warnings that might be issued by the compiler are to be suppressed. The warnings to suppress are specified by name, in string form.

# Type Annotations

As mentioned earlier, annotations were originally allowed only on declarations. However, for modern versions of Java, annotations can also be specified in most cases in which a type is used. This expanded aspect of annotations is called *type annotation*. For example, you can annotate the return type of a method, the type of **this** within a method, a cast, array levels, an inherited class, and a **throws** clause. You can also annotate generic types, including generic type parameter bounds and generic type arguments. (See [Chapter 14](#) for a discussion of generics.)

Type annotations are important because they enable tools to perform additional checks on code to help prevent errors. Understand that, as a general rule, **javac** will not perform these checks, itself. A separate tool is used for this purpose, although such a tool might operate as a compiler plug-in.

A type annotation must include `ElementType.TYPE_USE` as a target. (Recall that valid annotation targets are specified using the `@Target` annotation, as previously described.) A type annotation applies to the type that the annotation precedes. For example, assuming some type annotation called `@TypeAnno`, the following is legal:

```
void myMeth() throws @TypeAnno NullPointerException { // ...}
```

Here, `@TypeAnno` annotates `NullPointerException` in the `throws` clause.

You can also annotate the type of `this` (called the *receiver*). As you know, `this` is an implicit argument to all instance methods, and it refers to the invoking object. To annotate its type requires the use of another feature that was not originally part of Java. Beginning with JDK 8, you can explicitly declare `this` as the first parameter to a method. In this declaration, the type of `this` must be the type of its class; for example:

```
class SomeClass {
    int myMeth(SomeClass this, int i, int j) { // ...}
```

Here, because `myMeth()` is a method defined by `SomeClass`, the type of `this` is `SomeClass`. Using this declaration, you can now annotate the type of `this`. For example, again assuming that `@TypeAnno` is a type annotation, the following is legal:

```
int myMeth(@TypeAnno SomeClass this, int i, int j) { // ...}
```

It is important to understand that it is not necessary to declare `this` unless you are annotating it. (If `this` is not declared, it is still implicitly passed, as it always has been.) Also, explicitly declaring `this` does not change any aspect of the method's signature because `this` is implicitly declared, by default. Again, you will declare `this` only if you want to apply a type annotation to it. If you do declare `this`, it *must* be the first parameter.

The following program shows a number of the places that a type annotation can be used. It defines several annotations, of which several are for type annotation. The names and targets of the annotations are shown here:

Annotation	Target
@TypeAnno	ElementType.TYPE_USE
@MaxLen	ElementType.TYPE_USE
@NotZeroLen	ElementType.TYPE_USE
@Unique	ElementType.TYPE_USE
@What	ElementType.TYPE_PARAMETER
@EmptyOK	ElementType.FIELD
@Recommended	ElementType.METHOD

Notice that **@EmptyOK**, **@Recommended**, and **@What** are not type annotations. They are included for comparison purposes. Of special interest is **@What**, which is used to annotate a generic type parameter declaration. The comments in the program describe each use.

```
// Demonstrate several type annotations.  
import java.lang.annotation.*;  
import java.lang.reflect.*;  
  
// A marker annotation that can be applied to a type.  
@Target(ElementType.TYPE_USE)  
@interface TypeAnno { }  
  
// Another marker annotation that can be applied to a type.  
@Target(ElementType.TYPE_USE)  
@interface NotZeroLen { }  
  
// Still another marker annotation that can be applied to a type.  
@Target(ElementType.TYPE_USE)  
@interface Unique { }  
  
// A parameterized annotation that can be applied to a type.  
@Target(ElementType.TYPE_USE)  
@interface MaxLen {  
    int value();  
}  
  
// An annotation that can be applied to a type parameter.  
@Target(ElementType.TYPE_PARAMETER)  
@interface What {  
    String description();  
}  
  
// An annotation that can be applied to a field declaration.  
@Target(ElementType.FIELD)  
@interface EmptyOK { }  
  
// An annotation that can be applied to a method declaration.  
@Target(ElementType.METHOD)  
@interface Recommended { }  
  
// Use an annotation on a type parameter.  
class TypeAnnoDemo<@What(description = "Generic data type") T> {  
    // Has a type annotation on a constructor
```

```
// This annotates the field test.  
@EmptyOK String test;  
  
// Use a type annotation to annotate this (the receiver).  
public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {  
    return 10;  
}  
  
// Annotate the return type.  
public @TypeAnno Integer f2(int j, int k) {  
    return j+k;  
}  
  
// Annotate the method declaration.  
public @Recommended Integer f3(String str) {  
    return str.length() / 2;  
}  
  
// Use a type annotation with a throws clause.  
public void f4() throws @TypeAnno NullPointerException {  
    // ...  
}  
  
// Annotate array levels.  
String @MaxLen(10) [] @NotZeroLen [] w;  
  
// Annotate the array element type.  
@TypeAnno Integer[] vec;  
  
public static void myMeth(int i) {  
  
    // Use a type annotation on a type argument.  
    TypeAnnoDemo<@TypeAnno Integer> ob =  
        new TypeAnnoDemo<@TypeAnno Integer>();  
  
    // Use a type annotation with new.  
    @Unique TypeAnnoDemo<Integer> ob2 = new @Unique TypeAnnoDemo<Integer>();  
  
    Object x = Integer.valueOf(10);  
    Integer y;  
  
    // Use a type annotation on a cast
```

Although what most of the annotations in the preceding program refer to is clear, four uses require a bit of discussion. The first is the annotation of a method return type versus the annotation of a method declaration. In the program, pay special attention to these two method declarations:

```
myMeth(10);
// Annotate the return type.
public @TypeAnno Integer f2(int j, int k) {
    return j+k;
}

// Annotate the method declaration.
public @Recommended Integer f3(String str) {
    return str.length() / 2;
}
```

Notice that in both cases, an annotation precedes the method's return type (which is **Integer**). However, the two annotations annotate two different things. In the first case, the **@TypeAnno** annotation annotates **f2()**'s return type. This is because **@TypeAnno** has its target specified as **ElementType.TYPE\_USE**, which means that it can be used to annotate type uses. In the second case, **@Recommended** annotates the method declaration itself. This is because **@Recommended** has its target specified as **ElementType.METHOD**. As a result, **@Recommended** applies to the declaration, not the return type. Therefore, the target specification is used to eliminate what, at first glance, appears to be ambiguity between the annotation of a method declaration and the annotation of the method's return type.

One other thing about annotating a method return type: You cannot annotate a return type of **void**.

The second point of interest are the field annotations, shown here:

```
// Annotate the type (in this case String), not the field.
@TypeAnno String str;

// This annotates the field test.
@EmptyOK String test;
```

Here, **@TypeAnno** annotates the type **String**, but **@EmptyOK** annotates the field **test**. Even though both annotations precede the entire declaration,

their targets are different, based on the target element type. If the annotation has the **ElementType.TYPE\_USE** target, then the type is annotated. If it has **ElementType.FIELD** as a target, then the field is annotated. Thus, the situation is similar to that just described for methods, and no ambiguity exists. The same mechanism also disambiguates annotations on local variables.

Next, notice how **this** (the receiver) is annotated here:

```
public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {
```

Here, **this** is specified as the first parameter and is of type **TypeAnnoDemo** (which is the class of which **f()** is a member). As explained, an instance method declaration can explicitly specify the **this** parameter for the sake of applying a type annotation.

Finally, look at how array levels are annotated by the following statement:

```
String @MaxLen(10) [] @NotZeroLen [] w;
```

In this declaration, **@MaxLen** annotates the type of the first level, and **@NotZeroLen** annotates the type of the second level. In the declaration

```
@TypeAnno Integer[] vec;
```

the element type **Integer** is annotated.

## Repeating Annotations

Beginning with JDK 8, an annotation can be repeated on the same element. This is called *repeating annotations*. For an annotation to be repeatable, it must be annotated with the **@Repeatable** annotation, defined in **java.lang.annotation**. Its **value** field specifies the *container* type for the repeatable annotation. The container is specified as an annotation for which the **value** field is an array of the repeatable annotation type. Thus, to create a repeatable annotation, you must create a container annotation and then specify that annotation type as an argument to the **@Repeatable** annotation.

To access the repeated annotations using a method such as **getAnnotation()**, you will use the container annotation, not the repeatable annotation. The following program shows this approach. It converts the version of **MyAnno** shown previously into a repeatable annotation and demonstrates its use.

```
// Demonstrate a repeated annotation.

import java.lang.annotation.*;
import java.lang.reflect.*;

// Make MyAnno repeatable.
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface MyAnno {
    String str() default "Testing";
    int val() default 9000;
}

// This is the container annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos {
    MyAnno[] value();
}

class RepeatAnno {

    // Repeat MyAnno on myMeth().
    @MyAnno(str = "First annotation", val = -1)
    @MyAnno(str = "Second annotation", val = 100)
    public static void myMeth(String str, int i)
```

```

{
    RepeatAnno ob = new RepeatAnno();

    try {
        Class<?> c = ob.getClass();

        // Obtain the annotations for myMeth().
        Method m = c.getMethod("myMeth", String.class, int.class);

        // Display the repeated MyAnno annotations.
        Annotation anno = m.getAnnotation(MyRepeatedAnnos.class);
        System.out.println(anno);

    } catch (NoSuchMethodException exc) {
        System.out.println("Method Not Found.");
    }
}

public static void main(String[] args) {
    myMeth("test", 10);
}
}

```

The output is shown here:

```
@MyRepeatedAnnos(value=@MyAnno(val=-1, str="First annotation"),
@MyAnno(val=100, str="Second annotation"))
```

As explained, in order for **MyAnno** to be repeatable, it must be annotated with the **@Repeatable** annotation, which specifies its container annotation. The container annotation is called **MyRepeatedAnnos**. The program accesses the repeated annotations by calling **getAnnotation()**, passing in the class of the container annotation, not the repeatable annotation, itself. As the output shows, the repeated annotations are separated by a comma. They are not returned individually.

Another way to obtain the repeated annotations is to use one of the methods in **AnnotatedElement** that can operate directly on a repeated annotation. These are **getAnnotationsByType()** and

**getDeclaredAnnotationsByType( ).** Here, we will use the former. It is shown here:

```
default <T extends Annotation> T[ ]
getAnnotationsByType(Class<T> annoType)
```

It returns an array of the annotations of *annoType* associated with the invoking object. If no annotations are present, the array will be of zero length. Here is an example. Assuming the preceding program, the following sequence uses **getAnnotationsByType( )** to obtain the repeated **MyAnno** annotations:

```
Annotation[] annos = m.getAnnotationsByType(MyAnno.class);
for(Annotation a : annos)
    System.out.println(a);
```

Here, the repeated annotation type, which is **MyAnno**, is passed to **getAnnotationsByType( )**. The returned array contains all of the instances of **MyAnno** associated with **myMeth( )**, which, in this example, is two. Each repeated annotation can be accessed via its index in the array. In this case, each **MyAnno** annotation is displayed via a for-each loop.

## Some Restrictions

There are a number of restrictions that apply to annotation declarations. First, no annotation can inherit another. Second, all methods declared by an annotation must be without parameters. Furthermore, they must return one of the following:

- A primitive type, such as **int** or **double**
- An object of type **String** or **Class**
- An object of an **enum** type
- An object of another annotation type
- An array of a legal type.

Annotations cannot be generic. In other words, they cannot take type parameters. (Generics are described in [Chapter 14](#).) Finally, annotation methods cannot specify a **throws** clause.

# CHAPTER

## 13

# I/O, Try-with-Resources, and Other Topics

---

This chapter introduces one of Java’s most important packages, **java.io**, which supports Java’s basic I/O (input/output) system, including file I/O. Support for I/O comes from Java’s core API libraries, not from language keywords. For this reason, an in-depth discussion of this topic is found in Part II of this book, which examines several of Java’s API packages. Here, the foundation of this important subsystem is introduced so that you can see how it fits into the larger context of the Java programming and execution environment. This chapter also examines the **try-with-resources** statement and several more Java keywords: **transient**, **volatile**, **instanceof**, **native**, **strictfp**, and **assert**. It concludes by discussing static import and describing another use for the **this** keyword.

## I/O Basics

As you may have noticed while reading the preceding 12 chapters, not much use has been made of I/O in the sample programs. In fact, aside from **print( )** and **println( )**, none of the I/O methods have been used significantly. The reason is simple: Most real applications of Java are not text-based, console programs. Rather, they are either graphically oriented programs that rely on one of Java’s graphical user interface (GUI) frameworks, such as Swing, for user interaction, or they are web applications. Although text-based, console programs are excellent as teaching examples, they do not, as a general rule, constitute an important

use for Java in the real world. Also, Java's support for console I/O is limited and somewhat awkward to use—even in simple sample programs. Text-based console I/O is just not that useful in real-world Java programming.

The preceding paragraph notwithstanding, Java does provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master. A general overview of I/O is presented here. A detailed description is found in [Chapters 22](#) and [23](#).

## Streams

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the `java.io` package.

---

**NOTE** In addition to the stream-based I/O defined in `java.io`, Java also provides buffer- and channel-based I/O, which are defined in `java.nio` and its subpackages. They are described in [Chapter 23](#).

## Byte Streams and Character Streams

Java defines two types of I/O streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated.

Although old code that doesn't use character streams is becoming increasingly rare, it may still be encountered from time to time. As a general rule, old code should be updated to take advantage of character streams where appropriate.

One other point: At the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

An overview of both byte-oriented streams and character-oriented streams is presented in the following sections.

## The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers. The non-deprecated byte stream classes in **java.io** are shown in [Table 13-1](#). A few of these classes are discussed later in this section. Others are described in Part II of this book. Remember, to use the stream classes, you must import **java.io**.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements <b>InputStream</b>
FilterOutputStream	Implements <b>OutputStream</b>
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains <b>print( )</b> and <b>println( )</b>
PushbackInputStream	Input stream that allows bytes to be returned to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

**Table 13-1** The Non-DDeprecated Byte Stream Classes in **java.io**

The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are **read( )** and **write( )**, which, respectively, read and write bytes of data. Each has a form that is abstract and must be overridden by derived stream classes.

## The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes in **java.io** are shown in [Table 13-2](#).

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains <code>print()</code> and <code>println()</code>
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

**Table 13-2** The Character Stream I/O Classes in **java.io**

The abstract classes **Reader** and **Writer** define several key methods that the other stream classes implement. Two of the most important methods are **read( )** and **write( )**, which read and write characters of data, respectively. Each has a form that is abstract and must be overridden by derived stream classes.

## The Predefined Streams

As you know, all Java programs automatically import the **java.lang** package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment. For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system. **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**. This means that they can be used by any other part of your program and without reference to a specific **System** object.

**System.out** refers to the standard output stream. By default, this is the console. **System.in** refers to standard input, which is the keyboard by default. **System.err** refers to the standard error stream, which also is the console by default. However, these streams may be redirected to any compatible I/O device.

**System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they are typically used to read and write characters from and to the console. As you will see, you can wrap these within character-based streams, if desired.

The preceding chapters have been using **System.out** in their examples. You can use **System.err** in much the same way. As explained in the next section, use of **System.in** is a little more complicated.

## Reading Console Input

In the early days of Java, the only way to perform console input was to use a byte stream. Today, using a byte stream to read console input is still often acceptable, such as when used in sample programs. However, for commercial applications, the preferred method of reading console input is

to use a character-oriented stream. This makes your program easier to internationalize and maintain.

In Java, console input is accomplished (either directly or indirectly) by reading from **System.in**. One way to obtain a character-based stream that is attached to the console is to wrap **System.in** in a **BufferedReader**. The **BufferedReader** class supports a buffered input stream. A commonly used constructor is shown here:

`BufferedReader(Reader inputReader)`

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters.

Beginning with JDK 17, the precise way you obtain an **InputStreamReader** linked to **System.in** has changed. In the past, it was common to use the following **InputStreamReader** constructor for this purpose:

`InputStreamReader(InputStream inputStream)`

Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream*. Thus, the following line of code shows a common approach used in the past for creating a **BufferedReader** connected to the keyboard:

```
BufferedReader br = new BufferedReader(new  
InputStreamReader(System.in));
```

After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.

However, beginning with JDK 17, it is now recommended to explicitly specify the charset associated with the console when creating the **InputStreamReader**. A *charset* defines the way that bytes are mapped to characters. Normally, when a charset is not specified, the default charset of the JVM is used. However, in the case of the console, the charset used for console input may differ from this default charset. Thus, it is now recommended that this form of **InputStreamReader** constructor be used:

### `InputStreamReader(InputStream inputStream, Charset charset)`

For `charset`, use the charset associated with the console. This charset is returned by `charset()`, which is a new method added by JDK 17 to the **Console** class. (See [Chapter 22](#).) You obtain a **Console** object by calling `System.console()`. It returns a reference to the console, or `null` if no console is present. Therefore, today the following sequence shows one way to wrap `System.in` in a **BufferedReader**:

```
Console con = System.console(); // get the console
if(con==null) return; // if no console present, return

BufferedReader br = new
    BufferedReader(new InputStreamReader(System.in, con.charset()));
```

Of course, in cases in which you know that a console will be present, the sequence can be shortened to:

```
BufferedReader br = new
    BufferedReader(new InputStreamReader(System.in,
                                         System.console().charset()));
```

Because a console is (obviously) required to run the examples in this book, this is the form we will use.

One other point: It is also possible to obtain a **Reader** that is already associated with the console by use of the `reader()` method defined by **Console**. However, we will use the **InputStreamReader** approach as just described because it explicitly demonstrates the way that byte streams and character streams can interact.

## Reading Characters

To read a character from a **BufferedReader**, use `read()`. The version of `read()` that we will be using is

```
int read() throws IOException
```

Each time that `read()` is called, it reads a character from the input stream and returns it as an integer value. It returns `-1` when an attempt is made to

read at the end of the stream. As you can see, it can throw an **IOException**.

The following program demonstrates **read( )** by reading characters from the console until the user types a "q." Notice that any I/O exceptions that might be generated are simply thrown out of **main( )**. Such an approach is common when reading from the console in simple sample programs such as those shown in this book, but in more sophisticated applications, you can handle the exceptions explicitly.

```
// Use a BufferedReader to read characters from the console.  
import java.io.*;  
  
class BRRead {  
    public static void main(String[] args) throws IOException  
    {  
        char c;  
        BufferedReader br = new BufferedReader(new  
            InputStreamReader(System.in, System.console().charset()));  
  
        System.out.println("Enter characters, 'q' to quit.");  
        // read characters  
  
        do {  
            c = (char) br.read();  
            System.out.println(c);  
        } while(c != 'q');  
    }  
}
```

Here is a sample run:

```
Enter characters, 'q' to quit.  
123abcq  
1  
2  
3  
a  
b  
c  
q
```

This output may look a little different from what you expected because **System.in** is line buffered, by default. This means that no input is actually passed to the program until you press ENTER. As you can guess, this does not make **read( )** particularly valuable for interactive console input.

## Reading Strings

To read a string from the keyboard, use the version of **readLine( )** that is a member of the **BufferedReader** class. Its general form is shown here:

```
String readLine() throws IOException
```

As you can see, it returns a **String** object.

The following program demonstrates **BufferedReader** and the **readLine( )** method; the program reads and displays lines of text until you enter the word "stop":

```
// Read a string from console using a BufferedReader.  
import java.io.*;  
  
class BRReadLines {  
    public static void main(String[] args) throws IOException  
    {  
        // create a BufferedReader using System.in  
        BufferedReader br = new BufferedReader(new  
            InputStreamReader(System.in, System.console().charset()));  
  
        String str;  
        System.out.println("Enter lines of text.");  
        System.out.println("Enter 'stop' to quit.");  
        do {  
            str = br.readLine();  
            System.out.println(str);  
        } while(!str.equals("stop"));  
    }  
}
```

The next example creates a tiny text editor. It creates an array of **String** objects and then reads in lines of text, storing each line in the array. It will

read up to 100 lines or until you enter "stop." It uses a **BufferedReader** to read from the console.

```
// A tiny editor.  
import java.io.*;  
  
class TinyEdit {  
    public static void main(String[] args) throws IOException  
    {  
        // create a BufferedReader using System.in  
        BufferedReader br = new BufferedReader(new  
            InputStreamReader(System.in, System.console().charset()));  
  
        String[] str = new String[100];  
        System.out.println("Enter lines of text.");  
        System.out.println("Enter 'stop' to quit.");  
        for(int i=0; i<100; i++) {  
            str[i] = br.readLine();  
            if(str[i].equals("stop")) break;  
        }  
        System.out.println("\nHere is your file:");  
        // display the lines  
        for(int i=0; i<100; i++) {  
            if(str[i].equals("stop")) break;  
            System.out.println(str[i]);  
        }  
    }  
}
```

Here is a sample run:

```
Enter lines of text.  
Enter 'stop' to quit.  
This is line one.  
This is line two.  
Java makes working with strings easy.  
Just create String objects.  
stop  
Here is your file:  
This is line one.  
This is line two.  
Java makes working with strings easy.  
Just create String objects.
```

## Writing Console Output

Console output is most easily accomplished with **print( )** and **println( )**, described earlier, which are used in most of the examples in this book. These methods are defined by the class **PrintStream** (which is the type of object referenced by **System.out**). Even though **System.out** is a byte stream, using it for simple program output is still acceptable. However, a character-based alternative is described in the next section.

Because **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write( )**. Thus, **write( )** can be used to write to the console. The simplest form of **write( )** defined by **PrintStream** is shown here:

```
void write(int byteval)
```

This method writes the byte specified by *byteval*. Although *byteval* is declared as an integer, only the low-order eight bits are written. Here is a short example that uses **write( )** to output the character "A" followed by a newline to the screen:

```
// Demonstrate System.out.write().
class WriteDemo {
    public static void main(String[] args) {
        int b;

        b = 'A';
        System.out.write(b);
        System.out.write('\n');
    }
}
```

You will not often use **write( )** to perform console output (although doing so might be useful in some situations) because **print( )** and **println( )** are substantially easier to use.

## The PrintWriter Class

Although using **System.out** to write to the console is acceptable, its use is probably best for debugging purposes or for sample programs, such as those found in this book. For real-world programs, the recommended method of writing to the console when using Java is through a **PrintWriter** stream. **PrintWriter** is one of the character-based classes. Using a character-based class for console output makes internationalizing your program easier.

**PrintWriter** defines several constructors. The one we will use is shown here:

```
PrintWriter(InputStream outputStream, boolean flushingOn)
```

Here, *outputStream* is an object of type **OutputStream**, and *flushingOn* controls whether Java flushes the output stream every time a **println( )** method (among others) is called. If *flushingOn* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.

**PrintWriter** supports the **print( )** and **println( )** methods. Thus, you can use these methods in the same way as you used them with **System.out**. If an argument is not a simple type, the **PrintWriter** methods call the object's **toString( )** method and then display the result.

To write to the console by using a **PrintWriter**, specify **System.out** for the output stream and automatic flushing. For example, this line of code creates a **PrintWriter** that is connected to console output:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

The following application illustrates using a **PrintWriter** to handle console output:

```
// Demonstrate PrintWriter
import java.io.*;

public class PrintWriterDemo {
    public static void main(String[] args) {
        PrintWriter pw = new PrintWriter(System.out, true);

        pw.println("This is a string");
        int i = -7;
        pw.println(i);
        double d = 4.5e-7;
        pw.println(d);
    }
}
```

The output from this program is shown here:

```
This is a string
-7
4.5E-7
```

Remember, there is nothing wrong with using **System.out** to write simple text output to the console when you are learning Java or debugging your programs. However, using a **PrintWriter** makes your real-world applications easier to internationalize. Because no advantage is gained by using a **PrintWriter** in the sample programs shown in this book, we will continue to use **System.out** to write to the console.

## Reading and Writing Files

Java provides a number of classes and methods that allow you to read and write files. Before we begin, it is important to state that the topic of file I/O is quite large, and file I/O is examined in detail in Part II. The purpose of this section is to introduce the basic techniques that read from and write to a file. Although byte streams are used, these techniques can be adapted to the character-based streams.

Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. Although both classes support additional constructors, the following are the forms that we will be using:

```
FileInputStream(String fileName) throws FileNotFoundException  
FileOutputStream(String fileName) throws FileNotFoundException
```

Here, *fileName* specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then

**FileNotFoundException** is thrown. For output streams, if the file cannot be opened or created, then **FileNotFoundException** is thrown.

**FileNotFoundException** is a subclass of **IOException**. When an output file is opened, any preexisting file by the same name is destroyed.

---

**NOTE** In situations in which a security manager is present, several of the file classes, including **FileInputStream** and **FileOutputStream**, will throw a **SecurityException** if a security violation occurs when attempting to open a file. By default, applications run via **java** do not use a security manager. For that reason, the I/O examples in this book do not need to watch for a possible **SecurityException**. However, other types of applications may use the security manager, and file I/O performed by such an application could generate a **SecurityException**. In that case, you will need to appropriately handle this exception. Be aware that JDK 17 deprecates the security manager for removal.

When you are done with a file, you must close it. This is done by calling the **close( )** method, which is implemented by both **FileInputStream** and **FileOutputStream**. It is shown here:

```
void close( ) throws IOException
```

Closing a file releases the system resources allocated to the file, allowing them to be used by another file. Failure to close a file can result in

“memory leaks” because of unused resources remaining allocated.

---

**NOTE** The `close()` method is specified by the **AutoCloseable** interface in **java.lang**.

**AutoCloseable** is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes, including **FileInputStream** and **OutputStream**.

Before moving on, it is important to point out that there are two basic approaches that you can use to close a file when you are done with it. The first is the traditional approach, in which `close()` is called explicitly when the file is no longer needed. This is the approach used by all versions of Java prior to JDK 7 and is, therefore, found in all pre-JDK 7 legacy code. The second is to use the **try-with-resources** statement added by JDK 7, which automatically closes a file when it is no longer needed. In this approach, no explicit call to `close()` is executed. Since you may still encounter pre-JDK 7 legacy code, it is important that you know and understand the traditional approach. Furthermore, the traditional approach could still be the best approach in some situations. Therefore, we will begin with it. The automated approach is described in the following section.

To read from a file, you can use a version of `read()` that is defined within **FileInputStream**. The one that we will use is shown here:

```
int read() throws IOException
```

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. `read()` returns `-1` when an attempt is made to read at the end of the stream. It can throw an **IOException**.

The following program uses `read()` to input and display the contents of a file that contains ASCII text. The name of the file is specified as a command-line argument.

```
/* Display a text file.  
To use this program, specify the name  
of the file that you want to see.  
For example, to see a file called TEST.TXT,  
use the following command line.  
  
    java ShowFile TEST.TXT  
*/  
  
import java.io.*;  
  
class ShowFile {  
    public static void main(String[] args)  
    {  
        int i;  
        FileInputStream fin;
```

```
// First, confirm that a filename has been specified.  
if(args.length != 1) {  
    System.out.println("Usage: ShowFile filename");  
    return;  
}  
  
// Attempt to open the file.  
try {  
    fin = new FileInputStream(args[0]);  
} catch(FileNotFoundException e) {  
    System.out.println("Cannot Open File");  
    return;  
}  
  
// At this point, the file is open and can be read.  
// The following reads characters until EOF is encountered.  
try {  
    do {  
        i = fin.read();  
        if(i != -1) System.out.print((char) i);  
    } while(i != -1);  
} catch(IOException e) {  
    System.out.println("Error Reading File");  
}  
  
// Close the file.  
try {  
    fin.close();  
} catch(IOException e) {  
    System.out.println("Error Closing File");  
}  
}
```

In the program, notice the **try/catch** blocks that handle the I/O errors that might occur. Each I/O operation is monitored for exceptions, and if an exception occurs, it is handled. Be aware that in simple programs or sample code, it is common to see I/O exceptions simply thrown out of **main( )**, as was done in the earlier console I/O examples. Also, in some real-world code, it can be helpful to let an exception propagate to a calling routine to

let the caller know that an I/O operation failed. However, most of the file I/O examples in this book handle all I/O exceptions explicitly, as shown, for the sake of illustration.

Although the preceding example closes the file stream after the file is read, there is a variation that is often useful. The variation is to call **close()** within a **finally** block. In this approach, all of the methods that access the file are contained within a **try** block, and the **finally** block is used to close the file. This way, no matter how the **try** block terminates, the file is closed. Assuming the preceding example, here is how the **try** block that reads the file can be recoded:

```
try {
    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);

} catch(IOException e) {
    System.out.println("Error Reading File");
} finally {
    // Close file on the way out of the try block.
    try {
        fin.close();
    } catch(IOException e) {
        System.out.println("Error Closing File");
    }
}
```

Although not an issue in this case, one advantage to this approach in general is that if the code that accesses a file terminates because of some non-I/O related exception, the file is still closed by the **finally** block.

Sometimes it's easier to wrap the portions of a program that open the file and access the file within a single **try** block (rather than separating the two) and then use a **finally** block to close the file. For example, here is another way to write the **ShowFile** program:

```
/* Display a text file.  
To use this program, specify the name  
of the file that you want to see.  
For example, to see a file called TEST.TXT,  
use the following command line.  
  
java ShowFile TEST.TXT  
  
This variation wraps the code that opens and  
accesses the file within a single try block.  
The file is closed by the finally block.  
*/  
  
import java.io.*;  
  
class ShowFile {  
    public static void main(String[] args)  
    {  
        int i;  
        FileInputStream fin = null;  
  
        // First, confirm that a filename has been specified.  
        if(args.length != 1) {  
            System.out.println("Usage: ShowFile filename");  
            return;  
        }  
  
        // The following code opens a file, reads characters until EOF  
        // is encountered, and then closes the file via a finally block.  
        try {  
            fin = new FileInputStream(args[0]);  
  
            do {  
                i = fin.read();  
                if(i != -1) System.out.print((char) i);  
            } while(i != -1);  
        }  
    }  
}
```

```
    } catch(FileNotFoundException e) {
        System.out.println("File Not Found.");
    } catch(IOException e) {
        System.out.println("An I/O Error Occurred");
    } finally {
        // Close file in all cases.
        try {
            if(fin != null) fin.close();
        } catch(IOException e) {
            System.out.println("Error Closing File");
        }
    }
}
```

In this approach, notice that **fin** is initialized to **null**. Then, in the **finally** block, the file is closed only if **fin** is not **null**. This works because **fin** will be non-**null** only if the file is successfully opened. Thus, **close( )** is not called if an exception occurs while opening the file.

It is possible to make the **try/catch** sequence in the preceding example a bit more compact. Because **FileNotFoundException** is a subclass of **IOException**, it need not be caught separately. For example, here is the sequence recoded to eliminate catching **FileNotFoundException**. In this case, the standard exception message, which describes the error, is displayed.

```

try {
    fin = new FileInputStream(args[0]);

    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);

} catch(IOException e) {
    System.out.println("I/O Error: " + e);
} finally {
    // Close file in all cases.
    try {
        if(fin != null) fin.close();
    } catch(IOException e) {
        System.out.println("Error Closing File");
    }
}

```

In this approach, any error, including an error opening the file, is simply handled by the single **catch** statement. Because of its compactness, this approach is used by many of the I/O examples in this book. Be aware, however, that this approach is not appropriate in cases in which you want to deal separately with a failure to open a file, such as might be caused if a user mistypes a filename. In such a situation, you might want to prompt for the correct name, for example, before entering a **try** block that accesses the file.

To write to a file, you can use the **write( )** method defined by **FileOutputStream**. Its simplest form is shown here:

```
void write(int byteval) throws IOException
```

This method writes the byte specified by *byteval* to the file. Although *byteval* is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an **IOException** is thrown. The next example uses **write( )** to copy a file:

```
/* Copy a file.  
To use this program, specify the name  
of the source file and the destination file.  
For example, to copy a file called FIRST.TXT  
to a file called SECOND.TXT, use the following  
command line.  
  
    java CopyFile FIRST.TXT SECOND.TXT  
*/  
  
import java.io.*;  
  
class CopyFile {  
    public static void main(String[] args) throws IOException  
    {  
        int i;  
        FileInputStream fin = null;  
        FileOutputStream fout = null;  
  
        // First, confirm that both files have been specified.  
        if(args.length != 2) {  
            System.out.println("Usage: CopyFile from to");  
            return;  
        }  
  
        // Copy a File.  
        try {  
            // Attempt to open the files.  
            fin = new FileInputStream(args[0]);  
            fout = new FileOutputStream(args[1]);  
  
            do {  
                i = fin.read();  
                if(i != -1) fout.write(i);  
            } while(i != -1);  
  
        } catch(IOException e) {  
            System.out.println("I/O Error: " + e);  
        } finally {  
            try {  
                if(fin != null) fin.close();  
            } catch(IOException e2) {  
            }  
        }  
    }  
}
```

```

        try {
            if(fout != null) fout.close();
        } catch(IOException e2) {
            System.out.println("Error Closing Output File");
        }
    }
}

```

In the program, notice that two separate **try** blocks are used when closing the files. This ensures that both files are closed, even if the call to **fin.close()** throws an exception.

In general, notice that all potential I/O errors are handled in the preceding two programs by the use of exceptions. This differs from some computer languages that use error codes to report file errors. Not only do exceptions make file handling cleaner, but they also enable Java to easily differentiate the end-of-file condition from file errors when input is being performed.

## Automatically Closing a File

In the preceding section, the sample programs have made explicit calls to **close()** to close a file once it is no longer needed. As mentioned, this is the way files were closed when using versions of Java prior to JDK 7.

Although this approach is still valid and useful, JDK 7 added a feature that offers another way to manage resources, such as file streams, by automating the closing process. This feature, sometimes referred to as *automatic resource management*, or *ARM* for short, is based on an expanded version of the **try** statement. The principal advantage of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed. As explained, forgetting to close a file can result in memory leaks and could lead to other problems.

Automatic resource management is based on an expanded form of the **try** statement. Here is its general form:

```
try (resource-specification) {  
    // use the resource  
}
```

Typically, *resource-specification* is a statement that declares and initializes a resource, such as a file stream. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the **try** block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. (Thus, there is no need to call **close()** explicitly.) Of course, this form of **try** can also include **catch** and **finally** clauses. This form of **try** is called the *try-with-resources* statement.

---

**NOTE** Beginning with JDK 9, it is also possible for the resource specification of the **try** to consist of a variable that has been declared and initialized earlier in the program. However, that variable must be *effectively final*, which means that it has not been assigned a new value after being given its initial value.

The **try-with-resources** statement can be used only with those resources that implement the **AutoCloseable** interface defined by **java.lang**. This interface defines the **close()** method. **AutoCloseable** is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes. Thus, **try-with-resources** can be used when working with streams, including file streams.

As a first example of automatically closing a file, here is a reworked version of the **ShowFile** program that uses it:

```
/* This version of the ShowFile program uses a try-with-resources
   statement to automatically close a file after it is no longer needed.
*/
import java.io.*;

class ShowFile {
    public static void main(String[] args)
    {
        int i;

        // First, confirm that a filename has been specified.
        if(args.length != 1) {
            System.out.println("Usage: ShowFile filename");
            return;
        }

        // The following code uses a try-with-resources statement to open
        // a file and then automatically close it when the try block is left.
        try(FileInputStream fin = new FileInputStream(args[0])) {

            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(FileNotFoundException e) {
            System.out.println("File Not Found.");
        } catch(IOException e) {
            System.out.println("An I/O Error Occurred");
        }
    }
}
```

In the program, pay special attention to how the file is opened within the **try** statement:

```
try(FileInputStream fin = new FileInputStream(args[0])) {
```

Notice how the resource-specification portion of the **try** declares a **FileInputStream** called **fin**, which is then assigned a reference to the file opened by its constructor. Thus, in this version of the program, the variable **fin** is local to the **try** block, being created when the **try** is entered. When the **try** is left, the stream associated with **fin** is automatically closed by an implicit call to **close()**. You don't need to call **close()** explicitly, which means that you can't forget to close the file. This is a key advantage of using **try-with-resources**.

It is important to understand that a resource declared in the **try** statement is implicitly **final**. This means that you can't assign to the resource after it has been created. Also, the scope of the resource is limited to the **try-with-resources** statement.

Before moving on, it is useful to mention that beginning with JDK 10, you can use local variable type inference to specify the type of the resource declared in a **try-with-resources** statement. To do so, specify the type as **var**. When this is done, the type of the resource is inferred from its initializer. For example, the **try** statement in the preceding program can now be written like this:

```
try(var fin = new FileInputStream(args[0])) {
```

Here, **fin** is inferred to be of type **FileInputStream** because that is the type of its initializer. Because a number of readers will be working in Java environments that predate JDK 10, the **try-with-resource** statements in the remainder of this book will not make use of type inference so that the code works for as many readers as possible. Of course, going forward, you should consider using type inference in your own code.

You can manage more than one resource within a single **try** statement. To do so, simply separate each resource specification with a semicolon. The following program shows an example. It reworks the **CopyFile** program shown earlier so that it uses a single **try-with-resources** statement to manage both **fin** and **fout**.

```
/* A version of CopyFile that uses try-with-resources.  
 It demonstrates two resources (in this case files) being  
 managed by a single try statement.  
 */  
  
import java.io.*;  
  
class CopyFile {  
    public static void main(String[] args) throws IOException  
    {  
        int i;  
  
        // First, confirm that both files have been specified.  
        if(args.length != 2) {  
            System.out.println("Usage: CopyFile from to");  
            return;  
        }  
  
        // Open and manage two files via the try statement.  
        try (FileInputStream fin = new FileInputStream(args[0]);  
             FileOutputStream fout = new FileOutputStream(args[1]))  
        {  
  
            do {  
                i = fin.read();  
                if(i != -1) fout.write(i);  
            } while(i != -1);  
  
        } catch(IOException e) {  
            System.out.println("I/O Error: " + e);  
        }  
    }  
}
```

In this program, notice how the input and output files are opened within the **try** block:

```
try (FileInputStream fin = new FileInputStream(args[0]);  
     FileOutputStream fout = new FileOutputStream(args[1]))  
{  
    // ...
```

After this **try** block ends, both **fin** and **fout** will have been closed. If you compare this version of the program to the previous version, you will see that it is much shorter. The ability to streamline source code is a side-benefit of automatic resource management.

There is one other aspect to **try-with-resources** that needs to be mentioned. In general, when a **try** block executes, it is possible that an exception inside the **try** block will lead to another exception that occurs when the resource is closed in a **finally** clause. In the case of a “normal” **try** statement, the original exception is lost, being preempted by the second exception. However, when using **try-with-resources**, the second exception is *suppressed*. It is not, however, lost. Instead, it is added to the list of suppressed exceptions associated with the first exception. The list of suppressed exceptions can be obtained by using the **getSuppressed()** method defined by **Throwable**.

Because of the benefits that the **try-with-resources** statement offers, it will be used by many, but not all, of the sample programs in this edition of this book. Some of the examples will still use the traditional approach to closing a resource. There are several reasons for this. First, you may encounter legacy code that still relies on the traditional approach. It is important that all Java programmers be fully versed in, and comfortable with, the traditional approach when maintaining this older code. Second, it is possible that some programmers will continue to work in a pre-JDK 7 environment for a period of time. In such situations, the expanded form of **try** is not available. Finally, there may be cases in which explicitly closing a resource is more appropriate than the automated approach. For these reasons, some of the examples in this book will continue to use the traditional approach, explicitly calling **close()**. In addition to illustrating the traditional technique, these examples can also be compiled and run by all readers in all environments.

---

**REMEMBER** A few examples in this book use the traditional approach to closing files as a means of illustrating this technique, which is widely used in legacy code. However, for new code, you will usually want to use the automated approach supported by the **try-with-resources** statement just described.

## The transient and volatile Modifiers

Java defines two interesting type modifiers: **transient** and **volatile**. These modifiers are used to handle somewhat specialized situations.

When an instance variable is declared as **transient**, its value need not persist when an object is stored. For example:

```
class T {  
    transient int a; // will not persist  
    int b; // will persist  
}
```

Here, if an object of type **T** is written to a persistent storage area, the contents of **a** would not be saved, but the contents of **b** would.

The **volatile** modifier tells the compiler that the variable modified by **volatile** can be changed unexpectedly by other parts of your program. One of these situations involves multithreaded programs. In a multithreaded program, sometimes two or more threads share the same variable. For efficiency considerations, each thread can keep its own, private copy of such a shared variable. The real (or *master*) copy of the variable is updated at various times, such as when a **synchronized** method is entered. While this approach works fine, it may be inefficient at times. In some cases, all that really matters is that the master copy of a variable always reflects its current state. To ensure this, simply specify the variable as **volatile**, which tells the compiler that it must always use the master copy of a **volatile** variable (or, at least, always keep any private copies up to date with the master copy, and vice versa). Also, accesses to the shared variable must be executed in the precise order indicated by the program.

## Introducing instanceof

Sometimes, knowing the type of an object during run time is useful. For example, you might have one thread of execution that generates various types of objects, and another thread that processes these objects. In this situation, it might be useful for the processing thread to know the type of each object when it receives it. Another situation in which knowledge of an object's type at run time is important involves casting. In Java, an invalid cast causes a run-time error. Many invalid casts can be caught at compile time. However, casts involving class hierarchies can produce invalid casts

that can be detected only at run time. For example, a superclass called A can produce two subclasses, called B and C. Thus, casting a B object into type A or casting a C object into type A is legal, but casting a B object into type C (or vice versa) isn't legal. Because an object of type A can refer to objects of either B or C, how can you know, at run time, what type of object is actually being referred to before attempting the cast to type C? It could be an object of type A, B, or C. If it is an object of type B, a run-time exception will be thrown. Java provides the run-time operator **instanceof** to answer this question.

Before we begin, it is necessary to state that **instanceof** was significantly enhanced by JDK 17 with a powerful new feature based on pattern matching. Here, the traditional form of **instanceof** is introduced. The enhanced form is covered in [Chapter 17](#).

The traditional **instanceof** operator has this general form:

*objref instanceof type*

Here, *objref* is a reference to an instance of a class, and *type* is a class type. If *objref* is of the specified type or can be cast into the specified type, then the **instanceof** operator evaluates to **true**. Otherwise, its result is **false**. Thus, **instanceof** is the means by which your program can obtain run-time type information about an object.

The following program demonstrates **instanceof**:

```
// Demonstrate instanceof operator.
class A {
    int i, j;
}

class B {
    int i, j;
}

class C extends A {
```

```
        int k;
    }

    class D extends A {
        int k;
    }

    class InstanceOf {
        public static void main(String[] args) {
            A a = new A();
            B b = new B();
            C c = new C();
            D d = new D();
            if(a instanceof A)
                System.out.println("a is instance of A");
            if(b instanceof B)
                System.out.println("b is instance of B");
            if(c instanceof C)
                System.out.println("c is instance of C");
            if(c instanceof A)
                System.out.println("c can be cast to A");

            if(a instanceof C)
                System.out.println("a can be cast to C");

            System.out.println();

            // compare types of derived types
            A ob;

            ob = d; // A reference to d
            System.out.println("ob now refers to d");
            if(ob instanceof D)
                System.out.println("ob is instance of D");

            System.out.println();

            ob = c; // A reference to c
            System.out.println("ob now refers to c");

            if(ob instanceof D)
                System.out.println("ob can be cast to D");
            else
                System.out.println("ob cannot be cast to D");

            if(ob instanceof A)
                System.out.println("ob can be cast to A");
```

The output from this program is shown here:

```
System.out.println();
a is instance of A
b is instance of B      ist to Object
c is instance of C      may be cast to Object");
c can be cast to A

ob now refers to d      may be cast to Object");
ob is instance of D      may be cast to Object");

ob now refers to c      may be cast to Object");
} ob cannot be cast to D
ob can be cast to A

a may be cast to Object
b may be cast to Object
c may be cast to Object
d may be cast to Object
```

The **instanceof** operator isn't needed by most simple programs because, often, you know the type of object with which you are working. However, it can be very useful when you're writing generalized routines that operate on objects of a complex class hierarchy or that are created from code outside your direct control. As you will see, the pattern matching enhancements described in [Chapter 17](#) streamline its use.

## strictfp

With the creation of Java 2 several years ago, the floating-point computation model was relaxed slightly. Specifically, the new model did not require the truncation of certain intermediate values that occur during a computation. This prevented overflow or underflow in some cases. By modifying a class, a method, or interface with **strictfp**, you could ensure that floating-point calculations (and thus all truncations) took place precisely as they did in earlier versions of Java. When a class was modified by **strictfp**, all the methods in the class were also modified by **strictfp** automatically. However, beginning with JDK 17, all floating-point computations are now strict, and **strictfp** is obsolete and no longer required. Its use will now generate a warning message.

For versions of Java prior to JDK 17, the following example illustrates **strictfp**. It tells Java to use the original floating-point model for calculations in all methods defined within **MyClass**:

```
strictfp class MyClass { //...
```

Frankly, most programmers never needed to use **strictfp** because it affected only a very small class of problems.

---

**REMEMBER** Beginning with JDK 17, **strictfp** has been rendered obsolete and its use will now generate a warning message.

## Native Methods

Although it is rare, occasionally you may want to call a subroutine that is written in a language other than Java. Typically, such a subroutine exists as executable code for the CPU and environment in which you are working—that is, native code. For example, you may want to call a native code subroutine to achieve faster execution time. Or, you may want to use a specialized, third-party library, such as a statistical package. However, because Java programs are compiled to bytecode, which is then interpreted (or compiled on the fly) by the Java run-time system, it would seem impossible to call a native code subroutine from within your Java program. Fortunately, this conclusion is false. Java provides the **native** keyword, which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method.

To declare a native method, precede the method with the **native** modifier, but do not define any body for the method. For example:

```
public native int meth();
```

After you declare a native method, you must write the native method and follow a rather complex series of steps to link it with your Java code. Consult the Java documentation for current details.

---

**NOTE** JDK 21 introduces a more sophisticated API for using code libraries written in languages other than Java code. The new `java.lang.foreign.*` package contains an API that gives greater control over managing memory and calling functions in such libraries.

## Using assert

Another interesting keyword is **assert**. It is used during program development to create an *assertion*, which is a condition that should be true during the execution of the program. For example, you might have a method that should always return a positive integer value. You might test this by asserting that the return value is greater than zero using an **assert** statement. At run time, if the condition is true, no other action takes place. However, if the condition is false, then an **AssertionError** is thrown. Assertions are often used during testing to verify that some expected condition is actually met. They are not usually used for released code.

The **assert** keyword has two forms. The first is shown here:

```
assert condition;
```

Here, *condition* is an expression that must evaluate to a Boolean result. If the result is true, then the assertion is true and no other action takes place. If the condition is false, then the assertion fails and a default **AssertionError** object is thrown.

The second form of **assert** is shown here:

```
assert condition: expr ;
```

In this version, *expr* is a value that is passed to the **AssertionError** constructor. This value is converted to its string format and displayed if an assertion fails. Typically, you will specify a string for *expr*, but any non-**void** expression is allowed as long as it defines a reasonable string conversion.

Here is an example that uses **assert**. It verifies that the return value of **getnum()** is positive.

```
// Demonstrate assert.
class AssertDemo {
    static int val = 3;

    // Return an integer.
    static int getnum() {
        return val--;
    }

    public static void main(String[] args)
    {
        int n;

        for(int i=0; i < 10; i++) {
            n = getnum();

            assert n > 0; // will fail when n is 0

            System.out.println("n is " + n);
        }
    }
}
```

To enable assertion checking at run time, you must specify the **-ea** option. For example, to enable assertions for **AssertDemo**, execute it using this line:

```
java -ea AssertDemo
```

After compiling and running as just described, the program creates the following output:

```
n is 3
n is 2
n is 1
Exception in thread "main" java.lang.AssertionError
        at AssertDemo.main(AssertDemo.java:17)
```

In **main()**, repeated calls are made to the method **getnum()**, which returns an integer value. The return value of **getnum()** is assigned to **n** and then

tested using this **assert** statement:

```
assert n > 0; // will fail when n is 0
```

This statement will fail when **n** equals 0, which it will after the fourth call. When this happens, an exception is thrown.

As explained, you can specify the message displayed when an assertion fails. For example, if you substitute

```
assert n > 0 : "n is not positive!";
```

for the assertion in the preceding program, then the following output will be generated:

```
n is 3  
n is 2  
  
n is 1  
Exception in thread "main" java.lang.AssertionError: n is not  
positive!  
at AssertDemo.main(AssertDemo.java:17)
```

One important point to understand about assertions is that you must not rely on them to perform any action actually required by the program. The reason is that normally, released code will be run with assertions disabled. For example, consider this variation of the preceding program:

```
// A poor way to use assert!!!
class AssertDemo {
    // get a random number generator
    static int val = 3;

    // Return an integer.
    static int getnum() {
        return val--;
    }

    public static void main(String[] args)
    {
        int n = 0;

        for(int i=0; i < 10; i++) {

            assert (n = getnum()) > 0; // This is not a good idea!

            System.out.println("n is " + n);
        }
    }
}
```

In this version of the program, the call to **getnum()** is moved inside the **assert** statement. Although this works fine if assertions are enabled, it will cause a malfunction when assertions are disabled because the call to **getnum()** will never be executed! In fact, **n** must now be initialized because the compiler will recognize that it might not be assigned a value by the **assert** statement.

Assertions can be quite useful because they streamline the type of error checking that is common during development. For example, prior to **assert**, if you wanted to verify that **n** was positive in the preceding program, you had to use a sequence of code similar to this:

```
if(n < 0) {
    System.out.println("n is negative!");
    return; // or throw an exception
}
```

With **assert**, you need only one line of code. Furthermore, you don't have to remove the **assert** statements from your released code.

## Assertion Enabling and Disabling Options

When executing code, you can disable all assertions by using the **-da** option. You can enable or disable a specific package (and all of its subpackages) by specifying its name followed by three periods after the **-ea** or **-da** option. For example, to enable assertions in a package called **MyPack**, use

```
-ea:MyPack...
```

To disable assertions in **MyPack**, use

```
-da:MyPack...
```

You can also specify a class with the **-ea** or **-da** option. For example, this enables **AssertDemo** individually:

```
-ea:AssertDemo
```

## Static Import

Java includes a feature called *static import* that expands the capabilities of the **import** keyword. By following **import** with the keyword **static**, an **import** statement can be used to import the static members of a class or interface. When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class. This simplifies and shortens the syntax required to use a static member.

To understand the usefulness of static import, let's begin with an example that does *not* use it. The following program computes the hypotenuse of a right triangle. It uses two static methods from Java's built-in math class **Math**, which is part of **java.lang**. The first is **Math.pow()**, which returns a value raised to a specified power. The second is **Math.sqrt()**, which returns the square root of its argument.