

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 used. 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' 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. The 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, it is the methods that determine how a class' 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, applets don't require a `main()` method at all.

Note

C++ programmers will notice that the class declaration and the implementation of the methods are stored in the same place and not defined separately. This sometimes makes for very large .java files, since any class must be entirely defined in a single source file. This design feature was built into Java because it was felt that in the long run, having specification, declaration, and implementation all in one place makes for code that is easier to maintain.

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

Java™ 2: The Complete Reference

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 be an instance of **Box**. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement.

Again, 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.

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, more or less, 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.

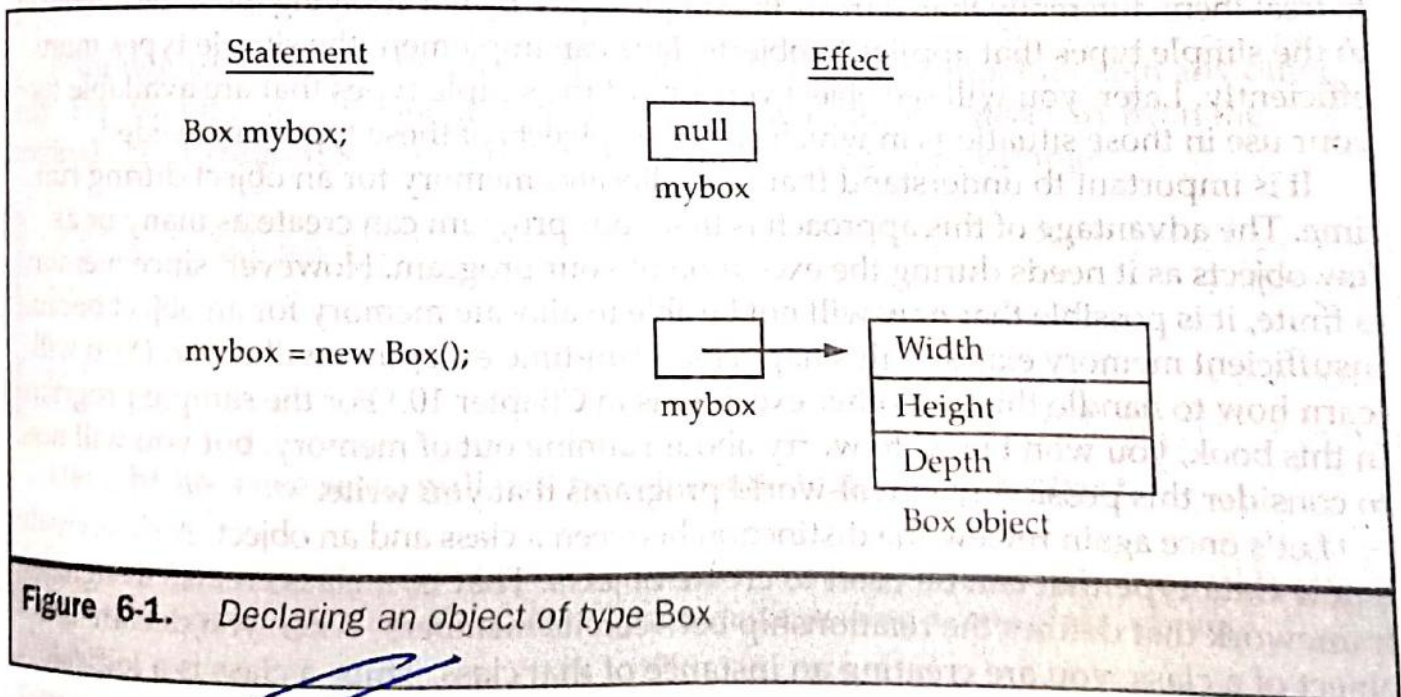
In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

```
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**. After this line executes, **mybox** contains the value **null**, which indicates that it does not yet point to an actual object. Any attempt to use **mybox** at this point will result in a compile-time error. The next line allocates an actual 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 the memory address of the actual **Box** object. The effect of these two lines of code is depicted in Figure 6-1.



ote

Those readers familiar with C/C++ have probably noticed that object references appear to be similar to pointers. This suspicion is, essentially, correct. An object reference is similar to a memory pointer. The main difference—and the key to Java's safety—is that you cannot manipulate references as you can actual pointers. Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

Closer Look at new

As just explained, the new operator dynamically allocates memory for an object. It has this general form:

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

Specifies the constructor for the class.

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 simple 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 simple types. By not applying the same overhead to the simple types that applies to objects, Java can implement the simple types more efficiently. Later, you will see object versions of the simple 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 this and other 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.

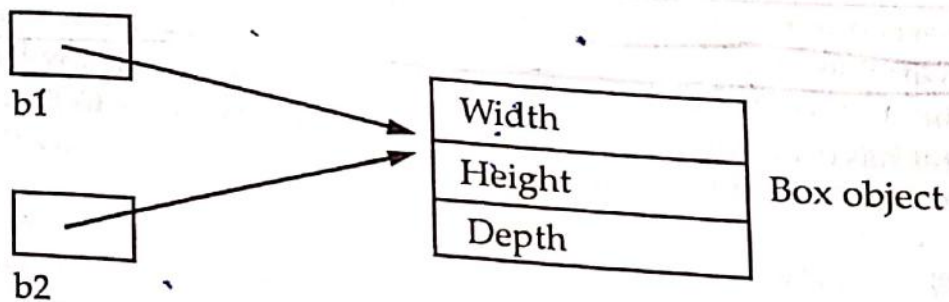
Let's once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance 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 **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);
    }
}
```

method

```
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:

Returning a Value

```
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, it 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 3,000 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 in 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 dimension 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.