5

Control Statements

Aprogramming language uses *control* statements to cause the flow of execution to

advance and branch based on changes to the state of a program. Java’s program control statements can be put into the following categories: selection, iteration, and jump. *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion. All of Java’s control statements are examined here.

Java’s Selection Statements

Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program’s execution based upon conditions known only during run time. You will be pleasantly surprised by the power and flexibility contained in these two statements.

if

The **if** statement was introduced in Chapter 2. It is examined in detail here. The **if** statement is Java’s conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement:

if (*condition*) *statement1*;

else *statement2*;

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.

The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

int a, b;

// ...

if(a < b) a = 0;

else b = 0;

7 7

78 Part I: The Java Language

Here, if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero.

Most often, the expression used to control the **if** will involve the relational operators. However, this is not technically necessary. It is possible to control the **if** using a single **boolean** variable, as shown in this code fragment:

boolean dataAvailable;

// ...

if (dataAvailable)

ProcessData();

else

waitForMoreData();

Remember, only one statement can appear directly after the **if** or the **else**. If you want to include more statements, you’ll need to create a block, as in this fragment:

int bytesAvailable;

// ...

if (bytesAvailable > 0) {

ProcessData();

bytesAvailable -= n;

} else

waitForMoreData();

Here, both statements within the **if** block will execute if **bytesAvailable** is greater than zero. Some programmers find it convenient to include the curly braces when using the **if**, even when there is only one statement in each clause. This makes it easy to add another statement at a later date, and you don’t have to worry about forgetting the braces. In fact, forgetting to define a block when one is needed is a common cause of errors. For example, consider the following code fragment:

int bytesAvailable;

// ...

if (bytesAvailable > 0) {

ProcessData();

bytesAvailable -= n;

} else

waitForMoreData();

bytesAvailable = n;

It seems clear that the statement **bytesAvailable = n;** was intended to be executed inside the **else** clause, because of the indentation level. However, as you recall, whitespace is insignificant to Java, and there is no way for the compiler to know what was intended. This code will compile without complaint, but it will behave incorrectly when run. The preceding example is fixed in the code that follows:

int bytesAvailable;

// ...

Chapter 5: Control Statements 79

if (bytesAvailable > 0) {

ProcessData();

bytesAvailable -= n;

} else {

waitForMoreData();

bytesAvailable = n;

}

Nested ifs

A*nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. When you nest **if**s, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**. Here is an example:

if(i == 10) {

if(j < 20) a = b;

if(k > 100) c = d; // this if is

else a = c; // associated with this else

}

else a = d; // this else refers to if(i == 10)

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)** because it is the closest **if** within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s is the *if-else-if ladder*. It looks like this:

if(*condition*)

*statement*;

else if(*condition*)

*statement*;

else if(*condition*)

*statement*;

.

.

.

else

*statement*;

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the

80 Part I: The Java Language

last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

Here is a program that uses an **if-else-if** ladder to determine which season a particular month is in.

// Demonstrate if-else-if statements.

class IfElse {

public static void main(String args[]) {

int month = 4; // April

String season;

if(month == 12 || month == 1 || month == 2)

season = "Winter";

else if(month == 3 || month == 4 || month == 5)

season = "Spring";

else if(month == 6 || month == 7 || month == 8)

season = "Summer";

else if(month == 9 || month == 10 || month == 11)

season = "Autumn";

else

season = "Bogus Month";

System.out.println("April is in the " + season + ".");

}

}

Here is the output produced by the program:

April is in the Spring.

You might want to experiment with this program before moving on. As you will find, no matter what value you give **month**, one and only one assignment statement within the ladder will be executed.

switch

The **switch** statement is Java’s multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

switch (*expression*) {

case *value1*:

// statement sequence

break;

case *value2*:

// statement sequence

break;

.

.

.

case *valueN*:

Chapter 5: Control Statements 81

// statement sequence

break;

default:

// default statement sequence

}

The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression. (An enumeration value can also be used to control a **switch** statement. Enumerations are described in Chapter 12.) Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case**

values are not allowed.

The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken.

The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of “jumping out” of the **switch**. Here is a simple example that uses a **switch** statement:

// A simple example of the switch.

class SampleSwitch {

public static void main(String args[]) {

for(int i=0; i<6; i++)

switch(i) {

case 0:

System.out.println("i is zero.");

break;

case 1:

System.out.println("i is one.");

break;

case 2:

System.out.println("i is two.");

break;

case 3:

System.out.println("i is three.");

break;

default:

System.out.println("i is greater than 3.");

}

}

}

The output produced by this program is shown here:

i is zero.

i is one.

i is two.

i is three.

i is greater than 3.

i is greater than 3.

82 Part I: The Java Language

As you can see, each time through the loop, the statements associated with the **case** constant that matches **i** are executed. All others are bypassed. After **i** is greater than 3, no **case** statements match, so the **default** statement is executed.

The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**. It is sometimes desirable to have multiple **case**s without **break** statements between them. For example, consider the following program:

// In a switch, break statements are optional.

class MissingBreak {

public static void main(String args[]) {

for(int i=0; i<12; i++)

switch(i) {

case 0:

case 1:

case 2:

case 3:

case 4:

System.out.println("i is less than 5");

break;

case 5:

case 6:

case 7:

case 8:

case 9:

System.out.println("i is less than 10");

break;

default:

System.out.println("i is 10 or more");

}

}

}

This program generates the following output:

i is less than 5

i is less than 5

i is less than 5

i is less than 5

i is less than 5

i is less than 10

i is less than 10

i is less than 10

i is less than 10

i is less than 10

i is 10 or more

i is 10 or more

As you can see, execution falls through each **case** until a **break** statement (or the end of the **switch**) is reached.

Chapter 5: Control Statements 83

While the preceding example is, of course, contrived for the sake of illustration, omitting the **break** statement has many practical applications in real programs. To sample its more realistic usage, consider the following rewrite of the season example shown earlier. This version uses a **switch** to provide a more efficient implementation.

// An improved version of the season program.

class Switch {

public static void main(String args[]) {

int month = 4;

String season;

switch (month) {

case 12:

case 1:

case 2:

season = "Winter";

break;

case 3:

case 4:

case 5:

season = "Spring";

break;

case 6:

case 7:

case 8:

season = "Summer";

break;

case 9:

case 10:

case 11:

season = "Autumn";

break;

default:

season = "Bogus Month";

}

System.out.println("April is in the " + season + ".");

}

}

Nested switch Statements

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested* **switch**. Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**. For example, the following fragment is perfectly valid:

switch(count) {

case 1:

switch(target) { // nested switch

case 0:

System.out.println("target is zero");

break;

84 Part I: The Java Language

case 1: // no conflicts with outer switch

System.out.println("target is one");

break;

}

break;

case 2: // ...

Here, the **case 1:** statement in the inner switch does not conflict with the **case 1:** statement in the outer switch. The **count** variable is only compared with the list of cases at the outer level. If **count** is 1, then **target** is compared with the inner list cases.

In summary, there are three important features of the **switch** statement to note:

• The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.

• No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common. • A **switch** statement is usually more efficient than a set of nested **if**s.

The last point is particularly interesting because it gives insight into how the Java compiler works. When it compiles a **switch** statement, the Java compiler will inspect each of the **case** constants and create a “jump table” that it will use for selecting the path of execution depending on the value of the expression. Therefore, if you need to select among a large group of values, a **switch** statement will run much faster than the equivalent logic coded using a sequence of **if-else**s. The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression. The compiler has no such knowledge of a long list of **if** expressions.

Iteration Statements

Java’s iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops.* As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

while

The **while** loop is Java’s most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

while(*condition*) {

// body of loop

}

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Chapter 5: Control Statements 85

Here is a **while** loop that counts down from 10, printing exactly ten lines of “tick”:

// Demonstrate the while loop.

class While {

public static void main(String args[]) {

int n = 10;

while(n > 0) {

System.out.println("tick " + n);

n--;

}

}

}

When you run this program, it will “tick” ten times:

tick 10

tick 9

tick 8

tick 7

tick 6

tick 5

tick 4

tick 3

tick 2

tick 1

Since the **while** loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with. For example, in the following fragment, the call to **println( )** is never executed:

int a = 10, b = 20;

while(a > b)

System.out.println("This will not be displayed");

The body of the **while** (or any other of Java’s loops) can be empty. This is because a *null statement* (one that consists only of a semicolon) is syntactically valid in Java. For example, consider the following program:

// The target of a loop can be empty.

class NoBody {

public static void main(String args[]) {

int i, j;

i = 100;

j = 200;

// find midpoint between i and j

while(++i < --j) ; // no body in this loop

86 Part I: The Java Language

System.out.println("Midpoint is " + i);

}

}

This program finds the midpoint between **i** and **j**. It generates the following output: Midpoint is 150

Here is how this **while** loop works. The value of **i** is incremented, and the value of **j** is decremented. These values are then compared with one another. If the new value of **i** is still less than the new value of **j**, then the loop repeats. If **i** is equal to or greater than **j**, the loop stops. Upon exit from the loop, **i** will hold a value that is midway between the original values of **i** and **j**. (Of course, this procedure only works when **i** is less than **j** to begin with.) As you can see, there is no need for a loop body; all of the action occurs within the conditional expression, itself. In professionally written Java code, short loops are frequently coded without bodies when the controlling expression can handle all of the details itself.

do-while

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

do {

// body of loop

} while (*condition*);

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java’s loops, *condition* must be a Boolean expression.

Here is a reworked version of the “tick” program that demonstrates the **do**-**while** loop. It generates the same output as before.

// Demonstrate the do-while loop.

class DoWhile {

public static void main(String args[]) {

int n = 10;

do {

System.out.println("tick " + n);

n--;

} while(n > 0);

}

}

The loop in the preceding program, while technically correct, can be written more efficiently as follows:

do {

System.out.println("tick " + n); } while(--n > 0);

Chapter 5: Control Statements 87

In this example, the expression **(– –n > 0)** combines the decrement of **n** and the test for zero into one expression. Here is how it works. First, the **– –**n statement executes, decrementing **n** and returning the new value of **n**. This value is then compared with zero. If it is greater than zero, the loop continues; otherwise it terminates.

The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once. Consider the following program, which implements a very simple help system for Java’s selection and iteration statements:

// Using a do-while to process a menu selection

class Menu {

public static void main(String args[])

throws java.io.IOException {

char choice;

do {

System.out.println("Help on:");

System.out.println(" 1. if");

System.out.println(" 2. switch");

System.out.println(" 3. while");

System.out.println(" 4. do-while");

System.out.println(" 5. for\n");

System.out.println("Choose one:");

choice = (char) System.in.read();

} while( choice < '1' || choice > '5');

System.out.println("\n");

switch(choice) {

case '1':

System.out.println("The if:\n");

System.out.println("if(condition) statement;");

System.out.println("else statement;");

break;

case '2':

System.out.println("The switch:\n");

System.out.println("switch(expression) {");

System.out.println(" case constant:");

System.out.println(" statement sequence");

System.out.println(" break;");

System.out.println(" // ...");

System.out.println("}");

break;

case '3':

System.out.println("The while:\n");

System.out.println("while(condition) statement;");

break;

case '4':

88 Part I: The Java Language

System.out.println("The do-while:\n");

System.out.println("do {");

System.out.println(" statement;");

System.out.println("} while (condition);");

break;

case '5':

System.out.println("The for:\n");

System.out.print("for(init; condition; iteration)");

System.out.println(" statement;");

break;

}

}

}

Here is a sample run produced by this program:

Help on:

1. if

2. switch

3. while

4. do-while

5. for

Choose one:

4

The do-while:

do {

statement;

} while (condition);

In the program, the **do-while** loop is used to verify that the user has entered a valid choice. If not, then the user is reprompted. Since the menu must be displayed at least once, the **do while** is the perfect loop to accomplish this.

A few other points about this example: Notice that characters are read from the keyboard by calling **System.in.read( )**. This is one of Java’s console input functions. Although Java’s console I/O methods won’t be discussed in detail until Chapter 13, **System.in.read( )** is used here to obtain the user’s choice. It reads characters from standard input (returned as integers, which is why the return value was cast to **char**). By default, standard input is line buffered, so you must press ENTER before any characters that you type will be sent to your program.

Java’s console input can be a bit awkward to work with. Further, most real-world Java programs will be graphical and window-based. For these reasons, not much use of console input has been made in this book. However, it is useful in this context. One other point to consider: Because **System.in.read( )** is being used, the program must specify the **throws java.io.IOException** clause. This line is necessary to handle input errors. It is part of Java’s exception handling features, which are discussed in Chapter 10.

for

You were introduced to a simple form of the **for** loop in Chapter 2. As you will see, it is a powerful and versatile construct.

Chapter 5: Control Statements 89

Beginning with JDK 5, there are two forms of the **for** loop. The first is the traditional form that has been in use since the original version of Java. The second is the new “for-each” form. Both types of **for** loops are discussed here, beginning with the traditional form. Here is the general form of the traditional **for** statement:

for(*initialization*; *condition*; *iteration*) {

// body

}

If only one statement is being repeated, there is no need for the curly braces. The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false. Here is a version of the “tick” program that uses a **for** loop:

// Demonstrate the for loop.

class ForTick {

public static void main(String args[]) {

int n;

for(n=10; n>0; n--)

System.out.println("tick " + n);

}

}

Declaring Loop Control Variables Inside the for Loop

Often the variable that controls a **for** loop is only needed for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the **for**. For example, here is the preceding program recoded so that the loop control variable **n** is declared as an **int** inside the **for**:

// Declare a loop control variable inside the for.

class ForTick {

public static void main(String args[]) {

// here, n is declared inside of the for loop

for(int n=10; n>0; n--)

System.out.println("tick " + n);

}

}

When you declare a variable inside a **for** loop, there is one important point to remember: the scope of that variable ends when the **for** statement does. (That is, the scope of the variable is limited to the **for** loop.) Outside the **for** loop, the variable will cease to exist. If you need

90 Part I: The Java Language

to use the loop control variable elsewhere in your program, you will not be able to declare it inside the **for** loop.

When the loop control variable will not be needed elsewhere, most Java programmers declare it inside the **for**. For example, here is a simple program that tests for prime numbers. Notice that the loop control variable, **i**, is declared inside the **for** since it is not needed elsewhere.

// Test for primes.

class FindPrime {

public static void main(String args[]) {

int num;

boolean isPrime = true;

num = 14;

for(int i=2; i <= num/i; i++) {

if((num % i) == 0) {

isPrime = false;

break;

}

}

if(isPrime) System.out.println("Prime");

else System.out.println("Not Prime");

}

}

Using the Comma

There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop. For example, consider the loop in the following program:

class Sample {

public static void main(String args[]) {

int a, b;

b = 4;

for(a=1; a<b; a++) {

System.out.println("a = " + a);

System.out.println("b = " + b);

b--;

}

}

}

As you can see, the loop is controlled by the interaction of two variables. Since the loop is governed by two variables, it would be useful if both could be included in the **for** statement, itself, instead of **b** being handled manually. Fortunately, Java provides a way to accomplish this. To allow two or more variables to control a **for** loop, Java permits you to include multiple statements in both the initialization and iteration portions of the **for**. Each statement is separated from the next by a comma.

Using the comma, the preceding **for** loop can be more efficiently coded as shown here:

// Using the comma.

class Comma {

public static void main(String args[]) { int a, b;

for(a=1**,** b=4; a<b; a++**,** b--) {

System.out.println("a = " + a);

System.out.println("b = " + b);

}

}

}

Chapter 5: Control Statements 91

In this example, the initialization portion sets the values of both **a** and **b**. The two comma separated statements in the iteration portion are executed each time the loop repeats. The program generates the following output:

a = 1

b = 4

a = 2

b = 3

***NOTE*** *If you are familiar with C/C++, then you know that in those languages the comma is an operator that can be used in any valid expression. However, this is not the case with Java. In Java, the comma is a separator.*

Some for Loop Variations

The **for** loop supports a number of variations that increase its power and applicability. The reason it is so flexible is that its three parts—the initialization, the conditional test, and the iteration—do not need to be used for only those purposes. In fact, the three sections of the

**for** can be used for any purpose you desire. Let’s look at some examples. One of the most common variations involves the conditional expression. Specifically, this expression does not need to test the loop control variable against some target value. In fact, the condition controlling the **for** can be any Boolean expression. For example, consider the following fragment:

boolean done = false;

for(int i=1; !done; i++) {

// ...

if(interrupted()) done = true;

}

In this example, the **for** loop continues to run until the **boolean** variable **done** is set to **true**. It does not test the value of **i**.

Here is another interesting **for** loop variation. Either the initialization or the iteration expression or both may be absent, as in this next program:

// Parts of the for loop can be empty.

class ForVar {

public static void main(String args[]) {

int i;

92 Part I: The Java Language

boolean done = false;

i = 0;

for( ; !done; ) {

System.out.println("i is " + i);

if(i == 10) done = true;

i++;

}

}

}

Here, the initialization and iteration expressions have been moved out of the **for**. Thus, parts of the **for** are empty. While this is of no value in this simple example—indeed, it would be considered quite poor style—there can be times when this type of approach makes sense. For example, if the initial condition is set through a complex expression elsewhere in the program or if the loop control variable changes in a nonsequential manner determined by actions that occur within the body of the loop, it may be appropriate to leave these parts of the **for** empty.

Here is one more **for** loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for** empty. For example:

for( ; ; ) {

// ...

}

This loop will run forever because there is no condition under which it will terminate. Although there are some programs, such as operating system command processors, that require an infinite loop, most “infinite loops” are really just loops with special termination requirements. As you will soon see, there is a way to terminate a loop— even an infinite loop like the one shown—that does not make use of the normal loop conditional expression.

The For-Each Version of the for Loop

Beginning with JDK 5, a second form of **for** was defined that implements a “for-each” style loop. As you may know, contemporary language theory has embraced the for-each concept, and it is quickly becoming a standard feature that programmers have come to expect. A for each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. Unlike some languages, such as C#, that implement a for-each loop by using the keyword **foreach**, Java adds the for-each capability by enhancing the **for** statement. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. The for-each style of **for** is also referred to as the *enhanced* **for** loop. The general form of the for-each version of the **for** is shown here:

for(*type itr-var : collection*) *statement-block*

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by *collection*. There are various types of collections that can be used with the **for**, but the only type used in this chapter is the array. (Other types of collections that can be used with the **for**, such as those defined by the Collections Framework,

Chapter 5: Control Statements 93

are discussed later in this book.) With each iteration of the loop, the next element in the collection is retrieved and stored in *itr-var*. The loop repeats until all elements in the collection have been obtained.

Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the base type of the array.

To understand the motivation behind a for-each style loop, consider the type of **for** loop that it is designed to replace. The following fragment uses a traditional **for** loop to compute the sum of the values in an array:

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];

To compute the sum, each element in **nums** is read, in order, from start to finish. Thus, the entire array is read in strictly sequential order. This is accomplished by manually indexing the **nums** array by **i**, the loop control variable.

The for-each style **for** automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array. Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end. For example, here is the preceding fragment rewritten using a for-each version of the **for**:

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int sum = 0;

for(int x: nums) sum += x;

With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on. Not only is the syntax streamlined, but it also prevents boundary errors.

Here is an entire program that demonstrates the for-each version of the **for** just described:

// Use a for-each style for loop.

class ForEach {

public static void main(String args[]) {

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int sum = 0;

// use for-each style for to display and sum the values

for(int x : nums) {

System.out.println("Value is: " + x);

sum += x;

}

System.out.println("Summation: " + sum);

}

}

94 Part I: The Java Language

The output from the program is shown here.

Value is: 1

Value is: 2

Value is: 3

Value is: 4

Value is: 5

Value is: 6

Value is: 7

Value is: 8

Value is: 9

Value is: 10

Summation: 55

As this output shows, the for-each style **for** automatically cycles through an array in sequence from the lowest index to the highest.

Although the for-each **for** loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement. For example, this program sums only the first five elements of **nums**:

// Use break with a for-each style for.

class ForEach2 {

public static void main(String args[]) {

int sum = 0;

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// use for to display and sum the values

for(int x : nums) {

System.out.println("Value is: " + x);

sum += x;

if(x == 5) break; // stop the loop when 5 is obtained

}

System.out.println("Summation of first 5 elements: " + sum);

}

}

This is the output produced:

Value is: 1

Value is: 2

Value is: 3

Value is: 4

Value is: 5

Summation of first 5 elements: 15

As is evident, the **for** loop stops after the fifth element has been obtained. The **break** statement can also be used with Java’s other loops, and it is discussed in detail later in this chapter. There is one important point to understand about the for-each style loop. Its iteration variable is “read-only” as it relates to the underlying array. An assignment to the iteration variable has no effect on the underlying array. In other words, you can’t change

Chapter 5: Control Statements 95

the contents of the array by assigning the iteration variable a new value. For example, consider this program:

// The for-each loop is essentially read-only.

class NoChange {

public static void main(String args[]) {

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

for(int x : nums) {

System.out.print(x + " ");

x = x \* 10; // no effect on nums

}

System.out.println();

for(int x : nums)

System.out.print(x + " ");

System.out.println();

}

}

The first **for** loop increases the value of the iteration variable by a factor of 10. However, this assignment has no effect on the underlying array **nums**, as the second **for** loop illustrates. The output, shown here, proves this point:

1 2 3 4 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

Iterating Over Multidimensional Arrays

The enhanced version of the **for** also works on multidimensional arrays. Remember, however, that in Java, multidimensional arrays consist of *arrays of arrays*. (For example, a two-dimensional array is an array of one-dimensional arrays.) This is important when iterating over a multidimensional array, because each iteration obtains the *next array*, not an individual element. Furthermore, the iteration variable in the **for** loop must be compatible with the type of array being obtained. For example, in the case of a two-dimensional array, the iteration variable must be a reference to a one-dimensional array. In general, when using the for-each **for** to iterate over an array of *N* dimensions, the objects obtained will be arrays of *N*–1 dimensions. To understand the implications of this, consider the following program. It uses nested **for** loops to obtain the elements of a two-dimensional array in row order, from first to last.

// Use for-each style for on a two-dimensional array.

class ForEach3 {

public static void main(String args[]) {

int sum = 0;

int nums[][] = new int[3][5];

// give nums some values

for(int i = 0; i < 3; i++)

96 Part I: The Java Language

for(int j=0; j < 5; j++)

nums[i][j] = (i+1)\*(j+1);

// use for-each for to display and sum the values

for(int x[] : nums) {

for(int y : x) {

System.out.println("Value is: " + y);

sum += y;

}

}

System.out.println("Summation: " + sum);

}

}

The output from this program is shown here:

Value is: 1

Value is: 2

Value is: 3

Value is: 4

Value is: 5

Value is: 2

Value is: 4

Value is: 6

Value is: 8

Value is: 10

Value is: 3

Value is: 6

Value is: 9

Value is: 12

Value is: 15

Summation: 90

In the program, pay special attention to this line:

for(int x[] : nums) {

Notice how **x** is declared. It is a reference to a one-dimensional array of integers. This is necessary because each iteration of the **for** obtains the next *array* in **nums**, beginning with the array specified by **nums[0]**. The inner **for** loop then cycles through each of these arrays, displaying the values of each element.

Applying the Enhanced for

Since the for-each style **for** can only cycle through an array sequentially, from start to finish, you might think that its use is limited, but this is not true. A large number of algorithms require exactly this mechanism. One of the most common is searching. For example, the following program uses a **for** loop to search an unsorted array for a value. It stops if the value is found.

Chapter 5: Control Statements 97

// Search an array using for-each style for.

class Search {

public static void main(String args[]) {

int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };

int val = 5;

boolean found = false;

// use for-each style for to search nums for val

for(int x : nums) {

if(x == val) {

found = true;

break;

}

}

if(found)

System.out.println("Value found!");

}

}

The for-each style **for** is an excellent choice in this application because searching an unsorted array involves examining each element in sequence. (Of course, if the array were sorted, a binary search could be used, which would require a different style loop.) Other types of applications that benefit from for-each style loops include computing an average, finding the minimum or maximum of a set, looking for duplicates, and so on.

Although we have been using arrays in the examples in this chapter, the for-each style **for** is especially useful when operating on collections defined by the Collections Framework, which is described in Part II. More generally, the **for** can cycle through the elements of any collection of objects, as long as that collection satisfies a certain set of constraints, which are described in Chapter 17.

Nested Loops

Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests **for** loops:

// Loops may be nested.

class Nested {

public static void main(String args[]) {

int i, j;

for(i=0; i<10; i++) {

for(j=i; j<10; j++)

System.out.print(".");

System.out.println();

}

}

}

98 Part I: The Java Language

The output produced by this program is shown here:

..........

.........

........

.......

......

.....

....

...

..

.

Jump Statements

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program. Each is examined here.

***NOTE*** *In addition to the jump statements discussed here, Java supports one other way that you can change your program’s flow of execution: through exception handling. Exception handling provides a structured method by which run-time errors can be trapped and handled by your program. It is supported by the keywords* ***try****,* ***catch****,* ***throw****,* ***throws****, and* ***finally****. In essence, the exception handling mechanism allows your program to perform a nonlocal branch. Since exception handling is a large topic, it is discussed in its own chapter, Chapter 10.*

Using break

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a “civilized” form of goto. The last two uses are explained here.

Using break to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

// Using break to exit a loop.

class BreakLoop {

public static void main(String args[]) {

for(int i=0; i<100; i++) {

if(i == 10) break; // terminate loop if i is 10

System.out.println("i: " + i);

}

System.out.println("Loop complete.");

}

}

This program generates the following output:

i: 0

i: 1

i: 2

i: 3

i: 4

i: 5

i: 6

i: 7

i: 8

i: 9

Loop complete.

Chapter 5: Control Statements 99

As you can see, although the **for** loop is designed to run from 0 to 99, the **break** statement causes it to terminate early, when **i** equals 10.

The **break** statement can be used with any of Java’s loops, including intentionally infinite loops. For example, here is the preceding program coded by use of a **while** loop. The output from this program is the same as just shown.

// Using break to exit a while loop.

class BreakLoop2 {

public static void main(String args[]) {

int i = 0;

while(i < 100) {

if(i == 10) break; // terminate loop if i is 10

System.out.println("i: " + i);

i++;

}

System.out.println("Loop complete.");

}

}

When used inside a set of nested loops, the **break** statement will only break out of the innermost loop. For example:

// Using break with nested loops.

class BreakLoop3 {

public static void main(String args[]) {

for(int i=0; i<3; i++) {

System.out.print("Pass " + i + ": ");

for(int j=0; j<100; j++) {

if(j == 10) break; // terminate loop if j is 10

System.out.print(j + " ");

}

System.out.println();

}

System.out.println("Loops complete.");

}

}

100 Part I: The Java Language

This program generates the following output:

Pass 0: 0 1 2 3 4 5 6 7 8 9

Pass 1: 0 1 2 3 4 5 6 7 8 9

Pass 2: 0 1 2 3 4 5 6 7 8 9

Loops complete.

As you can see, the **break** statement in the inner loop only causes termination of that loop. The outer loop is unaffected.

Here are two other points to remember about **break**. First, more than one **break** statement may appear in a loop. However, be careful. Too many **break** statements have the tendency to destructure your code. Second, the **break** that terminates a **switch** statement affects only that **switch** statement and not any enclosing loops.

***REMEMBER break*** *was not designed to provide the normal means by which a loop is terminated. The loop’s conditional expression serves this purpose. The* ***break*** *statement should be used to cancel a loop only when some sort of special situation occurs.*

Using break as a Form of Goto

In addition to its uses with the **switch** statement and loops, the **break** statement can also be employed by itself to provide a “civilized” form of the goto statement. Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also prohibits certain compiler optimizations. There are, however, a few places where the goto is a valuable

and legitimate construct for flow control. For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By using this form of **break**, you can, for example, break out of one or more blocks of code. These blocks need not be part of a loop or a **switch**. They can be

any block. Further, you can specify precisely where execution will resume, because this form of **break** works with a label. As you will see, **break** gives you the benefits of a goto without its problems.

The general form of the labeled **break** statement is shown here:

break *label*;

Most often, *label* is the name of a label that identifies a block of code. This can be a stand-alone block of code but it can also be a block that is the target of another statement. When this form of **break** executes, control is transferred out of the named block. The labeled block must enclose the **break** statement, but it does not need to be the immediately enclosing block. This means, for example, that you can use a labeled **break** statement to exit from a set of nested blocks. But you cannot use **break** to transfer control out of a block that does not enclose the **break** statement.

To name a block, put a label at the start of it. A *label* is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement. Doing so causes execution to resume at the *end* of the labeled block. For example, the following program shows three nested blocks, each with its own label. The **break** statement causes execution to jump forward, past the end of the block labeled **second**, skipping the two **println( )** statements.

Chapter 5: Control Statements 101

// Using break as a civilized form of goto.

class Break {

public static void main(String args[]) {

boolean t = true;

first: {

second: {

third: {

System.out.println("Before the break.");

if(t) break second; // break out of second block

System.out.println("This won't execute");

}

System.out.println("This won't execute");

}

System.out.println("This is after second block.");

}

}

}

Running this program generates the following output:

Before the break.

This is after second block.

One of the most common uses for a labeled **break** statement is to exit from nested loops. For example, in the following program, the outer loop executes only once:

// Using break to exit from nested loops

class BreakLoop4 {

public static void main(String args[]) {

outer: for(int i=0; i<3; i++) {

System.out.print("Pass " + i + ": ");

for(int j=0; j<100; j++) {

if(j == 10) break outer; // exit both loops

System.out.print(j + " ");

}

System.out.println("This will not print");

}

System.out.println("Loops complete.");

}

}

This program generates the following output:

Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.

As you can see, when the inner loop breaks to the outer loop, both loops have been terminated. Notice that this example labels the **for** statement, which has a block of code as its target. Keep in mind that you cannot break to any label which is not defined for an enclosing block. For example, the following program is invalid and will not compile:

// This program contains an error.

class BreakErr {

102 Part I: The Java Language

public static void main(String args[]) {

one: for(int i=0; i<3; i++) {

System.out.print("Pass " + i + ": ");

}

for(int j=0; j<100; j++) {

if(j == 10) break one; // WRONG

System.out.print(j + " ");

}

}

}

Since the loop labeled **one** does not enclose the **break** statement, it is not possible to transfer control out of that block.

Using continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop’s end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Here is an example program that uses **continue** to cause two numbers to be printed on each line:

// Demonstrate continue.

class Continue {

public static void main(String args[]) {

for(int i=0; i<10; i++) {

System.out.print(i + " ");

if (i%2 == 0) continue;

System.out.println("");

}

}

}

This code uses the **%** operator to check if **i** is even. If it is, the loop continues without printing a newline. Here is the output from this program:

0 1

2 3

4 5

6 7

8 9

As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue. Here is an example program that uses **continue** to print a triangular multiplication table for 0 through 9.

Chapter 5: Control Statements 103

// Using continue with a label.

class ContinueLabel {

public static void main(String args[]) {

outer: for (int i=0; i<10; i++) {

for(int j=0; j<10; j++) {

if(j > i) {

System.out.println();

continue outer;

}

System.out.print(" " + (i \* j));

}

}

System.out.println();

}

}

The **continue** statement in this example terminates the loop counting **j** and continues with the next iteration of the loop counting **i**. Here is the output of this program:

0

0 1

0 2 4

0 3 6 9

0 4 8 12 16

0 5 10 15 20 25

0 6 12 18 24 30 36

0 7 14 21 28 35 42 49

0 8 16 24 32 40 48 56 64

0 9 18 27 36 45 54 63 72 81

Good uses of **continue** are rare. One reason is that Java provides a rich set of loop statements which fit most applications. However, for those special circumstances in which early iteration is needed, the **continue** statement provides a structured way to accomplish it.

return

The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. Although a full discussion of **return** must

wait until methods are discussed in Chapter 6, a brief look at **return** is presented here. At any time in a method the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed. The following example illustrates this point. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main( )**.

// Demonstrate return.

class Return {

public static void main(String args[]) {

boolean t = true;

104 Part I: The Java Language

System.out.println("Before the return.");

if(t) return; // return to caller

System.out.println("This won't execute.");

}

}

The output from this program is shown here:

Before the return.

As you can see, the final **println( )** statement is not executed. As soon as **return** is executed, control passes back to the caller.

One last point: In the preceding program, the **if(t)** statement is necessary. Without it, the Java compiler would flag an “unreachable code” error because the compiler would know that the last **println( )** statement would never be executed. To prevent this error, the **if** statement is used here to trick the compiler for the sake of this demonstration.

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 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. A simplified general form of a **class** definition is shown here:

class *classname* {

*type instance-variable1*;

*type instance-variable2*;

105

106 Part I: The Java Language

// ...

*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’ 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;

}

Chapter 6: Introducing Classes 107

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 be 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.

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);

}

}

108 Part I: The Java Language

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);

}

}

Chapter 6: Introducing Classes 109

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.

***NOTE*** *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.*

A 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*( );

110 Part I: The Java Language

FIGURE 6-1

Declaring an object

of type Box

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 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.

Chapter 6: Introducing Classes 111

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.

112 Part I: The Java Language

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);

}

}

Chapter 6: Introducing Classes 113

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

114 Part I: The Java Language

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();

Chapter 6: Introducing Classes 115

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;

}

116 Part I: The Java Language

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;

Chapter 6: Introducing Classes 117

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 functions 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 immediately after 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’ 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.

118 Part I: The Java Language

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 simply 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

Chapter 6: Introducing Classes 119

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. The default constructor automatically initializes all instance variables to zero. 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 them 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;

}

}

120 Part I: The Java Language

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’ 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.

Chapter 6: Introducing Classes 121

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, which overlap with the names of the class’ 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** 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 name space 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 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 explicit need to destroy objects as in C++. 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.

The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle

122 Part I: The Java Language

such situations, Java provides a mechanism called *finalization.* By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.

The **finalize( )** method has this general form:

protected void finalize( )

{

// finalization code here

}

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class. This and the other access specifiers are explained in Chapter 7. It is important to understand that **finalize( )** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( )** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

***NOTE*** *If you are familiar with C++, then you know that C++ allows you to define a destructor for a class, which is called when an object goes out-of-scope. Java does not support this idea or provide for destructors. The* ***finalize( )*** *method only approximates the function of a destructor. As you get more experienced with Java, you will see that the need for destructor functions is minimal because of Java’s garbage collection subsystem.*

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’ 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.

Chapter 6: Introducing Classes 123

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 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 and three methods. 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();

124 Part I: The Java Language

// 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.

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);

}

125

126 Part I: The Java Language

// 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");

}

Chapter 7: A Closer Look at Methods and Classes 127

// 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

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 to have 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

128 Part I: The Java Language

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();

Chapter 7: A Closer Look at Methods and Classes 129

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;

130 Part I: The Java Language

// 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 mycube is " + vol);

}

}

The output produced by this program is shown here:

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 parameters 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 equals(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.equals(ob2));

Chapter 7: A Closer Look at Methods and Classes 131

System.out.println("ob1 == ob3: " + ob1.equals(ob3));

}

}

This program generates the following output:

ob1 == ob2: true

ob1 == ob3: false

As you can see, the **equals( )** 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 **equals( )** 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;

}

132 Part I: The Java Language

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

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, Java uses both approaches, depending upon what is passed.

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

Chapter 7: A Closer Look at Methods and Classes 133

// 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 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 by reference.

class Test {

int a, b;

Test(int i, int j) {

a = i;

b = j;

}

// pass an object

void meth(Test o) {

o.a \*= 2;

134 Part I: The Java Language

o.b /= 2;

}

}

class CallByRef {

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.

As a point of interest, 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.

***REMEMBER*** *When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.*

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;

}

Chapter 7: A Closer Look at Methods and Classes 135

}

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 ⋅ 2 ⋅ 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;

136 Part I: The Java Language

}

}

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 function calls. Many 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, you probably will not have to worry about this 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.

Chapter 7: A Closer Look at Methods and Classes 137

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

138 Part I: The Java Language

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” which 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 specifier* that modifies its declaration. Java supplies a rich set of access specifiers. 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 later. 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.

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

Let’s begin by defining **public** and **private**. When a member of a class is modified by the **public** specifier, 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** specifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access specifier 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 the following chapter.)

In the classes developed so far, all members of a class have used the default access mode, which is essentially public. 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 specifier 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:

Chapter 7: A Closer Look at Methods and Classes 139

/\* 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

140 Part I: The Java Language

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:");

Chapter 7: A Closer Look at Methods and Classes 141

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 call other **static** methods.

• They must only access **static** data.

• 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) {

142 Part I: The Java Language

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 local variable **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);

}

Chapter 7: A Closer Look at Methods and Classes 143

}

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 variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared. For 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** variables. Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant.

The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This second usage of **final** is described 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};

144 Part I: The Java Language

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[]) {

Chapter 7: A Closer Look at Methods and Classes 145

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());

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

for(int i=0; i<8; i++)

System.out.println(mystack2.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.

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 *non-static*. A static nested class is one that has the **static** modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

The most important 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

146 Part I: The Java Language

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 within the scope of class **Outer**. The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner**. (In general, an inner class instance must be created by an enclosing scope.) You can, however, create an instance of **Inner** outside of **Outer** by qualifying its name with **Outer**, as in **Outer.Inner**.

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!

Chapter 7: A Closer Look at Methods and Classes 147

}

}

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 defined by a method or even within the body of a **for** loop, as this next program shows.

// 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();

}

}

The output from this version of the program is shown here.

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

148 Part I: The Java Language

display: outer\_x = 100

display: outer\_x = 100

While nested classes are not applicable to all stiuations, they are particularly helpful when handling events. We will return to the topic of nested classes in Chapter 22. 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 final point: 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 example 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** constant.

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 a peer class of **String**, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java. (**StringBuffer** is 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, this statement

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

results in **myString** containing “I like Java.”

Chapter 7: A Closer Look at Methods and Classes 149

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(String *object*)

int length( )

char charAt(int *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

150 Part I: The Java Language

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:

// Demonstrate String arrays.

class StringDemo3 {

public static void main(String args[]) {

String str[] = { "one", "two", "three" };

for(int i=0; i<str.length; i++)

System.out.println("str[" + i + "]: " +

str[i]);

}

}

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:

// Display all command-line arguments.

class CommandLine {

public static void main(String args[]) {

for(int i=0; i<args.length; i++)

System.out.println("args[" + i + "]: " +

args[i]);

}

}

Chapter 7: A Closer Look at Methods and Classes 151

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 16.*

Varargs: Variable-Length Arguments

Beginning with JDK 5, Java has included 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 19, it takes a variable number of arguments, which it formats and then outputs.

Prior to JDK 5, 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 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 + " ");

152 Part I: The Java Language

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[])

{

Chapter 7: A Closer Look at Methods and Classes 153

// 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();

}

154 Part I: The Java Language

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();

}

Chapter 7: A Closer Look at Methods and Classes 155

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 a normal parameter. 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();

}

156 Part I: The Java Language

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.