Operators and Statements

OCA EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ **Using Operators and Decision Constructs**

- Use Java operators; including parentheses to override operator precedence
- Create if and if/else and ternary constructs
- Use a switch statement

✓ **Using Loop Constructs**

- Create and use while loops
- Create and use for loops including the enhanced for loop
- Create and use do/while loops
- Compare loop constructs
- Use break and continue

Like many programming languages, Java is composed primarily of variables, operators, and statements put together in some logical order. In the previous chapter, we discussed variables and gave some examples; in this chapter we'll discuss the various operators and statements available to you within the language. This knowledge will allow you to build complex functions and class structures that you'll see in later chapters.

Understanding Java Operators

A Java *operator* is a special symbol that can be applied to a set of variables, values, or literals—referred to as operands—and that returns a result. Three flavors of operators are available in Java: unary, binary, and ternary. These types of operators can be applied to one, two, or three operands, respectively.

Java operators are not necessarily evaluated from left-to-right order. For example, the following Java expression is actually evaluated from right-to-left given the specific operators involved:

```
int y = 4;
double x = 3 + 2 * --y;
```

In this example, you would first decrement `y` to `3`, and then multiply the resulting value by `2`, and finally add `3`. The value would then be automatically upcast from `9` to `9.0` and assigned to `x`. The final values of `x` and `y` would be `9.0` and `3`, respectively. If you didn't follow that evaluation, don't worry. By the end of this chapter, solving problems like this should be second nature.

Unless overridden with parentheses, Java operators follow *order of operation*, listed in Table 2.1, by decreasing order of *operator precedence*. If two operators have the same level of precedence, then Java guarantees left-to-right evaluation. You need to know only those operators in bold for the OCA exam.

Operator	Symbols and examples
Post-unary operators	*expression*`++`, *expression*`--`
Pre-unary operators	`++`*expression*, `--`*expression*

Operator	Symbols and examples
Other unary operators	`+`, `-`, `!`
Multiplication/Division/Modulus	`*`, `/`, `%`
Addition/Subtraction	`+`, `-`
Shift operators	`<<`, `>>`, `>>>`
Relational operators	`<`, `>`, `<=`, `>=`, `instanceof`
Equal to/not equal to	`==`, `!=`
Logical operators	`&`, `^`, `|`
Short-circuit logical operators	`&&`, `||`
Ternary operators	`boolean expression ? expression1 : expression2`
Assignment operators	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `!=`, `<<=`, `>>=`, `>>>=`

We'll spend the first half of this chapter discussing many of the operators in this list as well as how operator precedence determines which operators should be applied first. Note that you won't be tested on some operators, although we recommend that you be aware of their existence.

Working with Binary Arithmetic Operators

We'll begin our discussion with *binary operators*, by far the most common operators in the Java language. They can be used to perform mathematical operations on variables, create logical expressions, as well as perform basic variable assignments. Binary operators are commonly combined in complex expressions with more than two variables; therefore, operator precedence is very important in evaluating expressions.

Arithmetic Operators

Arithmetic operators are often encountered in early mathematics and include addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulus (`%`). They also include the unary operators, `++` and `--`, although we cover them later in this chapter. As you may have

noticed in Table 2.1, the *multiplicative* operators (`*`, `/`, `%`) have a higher order of precedence than the *additive* operators (`+`, `-`). That means when you see an expression such as this:

```
int x = 2 * 5 + 3 * 4 - 8;
```

you first evaluate the `2 * 5` and `3 * 4`, which reduces the expression to the following:

```
int x = 10 + 12 - 8;
```

Then, you evaluate the remaining terms in left-to-right order, resulting in a value of `x` of *14*. Make sure you understand why the result is *24* as you'll likely see this kind of operator precedence question on the exam.

Notice that we said "Unless overridden with parentheses…" prior to Table 2.1. That's because you can change the order of operation explicitly by wrapping parentheses around the sections you want evaluated first. Compare the previous example with the following one containing the same values and operators, in the same order, but with two sets of parentheses:

```
int x = 2 * ((5 + 3) * 4 - 8);
```

This time you would evaluate the addition operator `10 + 3`, which reduces the expression to the following:

```
int x = 2 * (8 * 4 - 8);
```

You can further reduce this expression by multiplying the first two values within the parentheses:

```
int x = 2 * (32 - 8);
```

Next, you subtract the values within the parentheses before applying terms outside the parentheses:

```
int x = 2 * 24;
```

Finally, you would multiply the result by `2`, resulting in a value of `48` for `x`.

All of the arithmetic operators may be applied to any Java primitives, except `boolean` and `String`. Furthermore, only the addition operators `+` and `+=` may be applied to `String` values, which results in `String` concatenation.

Although we are sure you have seen most of the arithmetic operators before, the modulus operator, `%`, may be new to you. The modulus, or remainder operator, is simply the remainder when two numbers are divided. For example, `9` divided by `3` divides evenly and has no remainder; therefore, the remainder, or `9 % 3`, is `0`. On the other hand, `11` divided by `3` does not divide evenly; therefore, the remainder, or `11 % 3`, is `2`.

Be sure to understand the difference between arithmetic division and modulus. For integer values, division results in the floor value of the nearest integer that fulfills the operation, whereas modulus is the remainder value. The following examples illustrate this distinction:

```
System.out.print(9 / 3);  // Outputs 3
System.out.print(9 % 3);  // Outputs 0
```

```
System.out.print(10 / 3);  // Outputs 3
System.out.print(10 % 3);  // Outputs 1

System.out.print(11 / 3);  // Outputs 3
System.out.print(11 % 3);  // Outputs 2

System.out.print(12 / 3);  // Outputs 4
System.out.print(12 % 3);  // Outputs 0
```

Note that the division results only increase when the value on the left-hand side goes from `9` to `12`, whereas the modulus remainder value increases by `1` each time the left-hand side is increased until it wraps around to zero. For a given divisor `y`, which is `3` in these examples, the modulus operation results in a value between `0` and `(y - 1)` for positive dividends. This means that the result of a modulus operation is always `0`, `1`, or `2`.

The modulus operation is not limited to positive integer values in Java and may also be applied to negative integers and floating-point integers. For a given divisor `y` and negative dividend, the resulting modulus value is between and `(-y + 1)` and `0`. For the OCA exam, though, you are not required to be able to take the modulus of a negative integer or a floating-point number.

Numeric Promotion

Now that you understand the basics of arithmetic operators, it is vital we talk about primitive *numeric promotion*, as Java may do things that seem unusual to you at first. If you recall in Chapter 1, "Java Building Blocks," where we listed the primitive numeric types, each primitive has a bit-length. For example, you should know that a `long` takes up more space than an `int`, which in turn takes up more space than a `short`, and so on.

You should memorize certain rules Java will follow when applying operators to data types:

Numeric Promotion Rules

1. If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.
2. If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.

3. Smaller data types, namely `byte`, `short`, and `char`, are first promoted to `int` any time they're used with a Java binary arithmetic operator, even if neither of the operands is `int`.
4. After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

The last two rules are the ones most people have trouble with, and the ones likely to trip you up on the exam. For the third rule, note that unary operators are excluded from this rule. For example, applying `++` to a `short` value results in a `short` value. We'll discuss unary operators in the next section.

Let's tackle some examples for illustrative purposes:

- What is the data type of `x * y`?

  ```
  int x = 1;
  long y = 33;
  ```

  If we follow the first rule, since one of the values is `long` and the other is `int`, and `long` is larger than `int`, then the `int` value is promoted to a `long`, and the resulting value is `long`.

- What is the data type of `x + y`?

  ```
  double x = 39.21;
  float y = 2.1;
  ```

  This is actually a trick question, as this code will not compile! As you may remember from Chapter 1, , as in `2.1f`. If the value was set properly to `2.1f`, then the promotion would be similar to the last example, with both operands being promoted to a `double`, and the result would be a `double` value.

- What is the data type of `x / y`?

  ```
  short x = 10;
  short y = 3;
  ```

  In this case, we must apply the third rule, namely that `x` and `y` will both be promoted to `int` before the operation, resulting in an output of type `int`. Pay close attention to the fact that the resulting output is not a `short`, as we'll come back to this example in the upcoming section on assignment operators.

- What is the data type of `x * y / z`?

  ```
  short x = 14;
  float y = 13;
  double z = 30;
  ```

  In this case, we must apply all of the rules. First, `x` will automatically be promoted to `int` solely because it is a `short` and it is being used in an arithmetic binary operation.

The promoted `x` value will then be automatically promoted to a `float` so that it can be multiplied with `y`. The result of `x * y` will then be automatically promoted to a `double`, so that it can be multiplied with `z`, resulting in a `double` value.

Working with Unary Operators

By definition, a *unary* operator is one that requires exactly one operand, or variable, to function. As shown in Table 2.2, they often perform simple tasks, such as increasing a numeric variable by one, or negating a `boolean` value.

TABLE 2.2 Java unary operators

Unary operator	Description
`+`	Indicates a number is positive, although numbers are assumed to be positive in Java unless accompanied by a negative unary operator
`-`	Indicates a literal number is negative or negates an expression
`++`	Increments a value by 1
`--`	Decrements a value by 1
`!`	Inverts a Boolean's logical value

Logical Complement and Negation Operators

The *logical complement operator,* `!`, flips the value of a `boolean` expression. For example, if the value is `true`, it will be converted to `false`, and vice versa. To illustrate this, compare the outputs of the following statements:

```
boolean x = false;
System.out.println(x);  // false
x = !x;
System.out.println(x);  // true
```

Likewise, the *negation operator*, `-`, reverses the sign of a numeric expression, as shown in these statements:

```
double x = 1.21;
```

```
System.out.println(x);  // 1.21
x = -x;
System.out.println(x);  // -1.21
x = -x;
System.out.println(x);  // 1.21
```

Based on the description, it might be obvious that some operators require the variable or expression they're acting upon to be of a specific type. For example, you cannot apply a negation operator, `-`, to a `boolean` expression, nor can you apply a logical complement operator, `!`, to a numeric expression. Be wary of questions on the exam that try to do this, as they'll cause the code to fail to compile. For example, none of the following lines of code will compile:

```
int x = !5;  // DOES NOT COMPILE
boolean y = -true;  // DOES NOT COMPILE
boolean z = !0;  // DOES NOT COMPILE
```

The first statement will not compile due the fact that in Java you cannot perform a logical inversion of a numeric value. The second statement does not compile because you cannot numerically negate a `boolean` value; you need to use the logical inverse operator. Finally, the last statement does not compile because you cannot take the logical complement of a numeric value, nor can you assign an integer to a `boolean` variable.

Keep an eye out for questions on the exam that use the logical complement operator or numeric values with `boolean` expressions or variables. Unlike some other programming languages, in Java `1` and `true` are not related in any way, just as `0` and `false` are not related.

Increment and Decrement Operators

Increment and decrement operators, `++` and `--`, respectively, can be applied to numeric operands and have the higher order or precedence, as compared to binary operators. In other words, they often get applied first to an expression.

Increment and decrement operators require special care because the order they are applied to their associated operand can make a difference in how an expression is processed. If the operator is placed before the operand, referred to as the *pre-increment operator* and the *pre-decrement operator*, then the operator is applied first and the value return is the *new value* of the expression. Alternatively, if the operator is placed after the operand, referred to as the *post-increment operator* and the *post-decrement operator*, then the *original value* of the expression is returned, with operator applied after the value is returned.

The following code snippet illustrates this distinction:

```
int counter = 0;
System.out.println(counter);  // Outputs 0
System.out.println(++counter);  // Outputs 1
System.out.println(counter); // Outputs 1
System.out.println(counter--);  // Outputs 1
System.out.println(counter);  // Outputs 0
```

The first pre-increment operator updates the value for `counter` and outputs the new value of `1`. The next post-decrement operator also updates the value of `counter` but outputs the value before the decrement occurs.

One common practice in a certification exam, albeit less common in the real world, is to apply multiple increment or decrement operators to a single variable on the same line:

```
int x = 3;
int y = ++x * 5 / x-- + --x;
System.out.println("x is " + x);
System.out.println("y is " + y);
```

This one is more complicated than the previous example because `x` is modified three times on the same line. Each time it is modified, as the expression moves from left to right, the value of `x` changes, with different values being assigned to the variable. As you'll recall from our discussion on operator precedence, order of operation plays an important part in evaluating this example.

So how do you read this code? First, the `x` is incremented and returned to the expression, which is multiplied by `5`. We can simplify this:

```
int y = 4 * 5 / x-- + --x;  // x assigned value of 4
```

Next, `x` is decremented, but the original value of `4` is used in the expression, leading to this:

```
int y = 4 * 5 / 4 + --x;  // x assigned value of 3
```

The final assignment of `x` reduces the value to `2`, and since this is a pre-increment operator, that value is returned to the expression:

```
int y = 4 * 5 / 4 + 2;  // x assigned value of 2
```

Finally, we evaluate the multiple and division from left-to-right, and finish with the addition. The result is then printed:

```
x is 2
y is 7
```

Using Additional Binary Operators

We'll now expand our discussion of binary operators to include all other binary operators that you'll need to know for the exam. This includes operators that perform assignments, those that compare arithmetic values and return `boolean` results, and those that compare `boolean` and object values and return `boolean` results.

Assignment Operators

An *assignment operator* is a binary operator that modifies, or assigns, the variable on the left-hand side of the operator, with the result of the value on the right-hand side of the equation. The simplest assignment operator is the `=` assignment, which you have seen already:

```
int x = 1;
```

This statement assigns `x` the value of `1`.

Java will automatically promote from smaller to larger data types, as we saw in the previous section on arithmetic operators, but it will throw a compiler exception if it detects you are trying to convert from larger to smaller data types.

Let's return to some examples similar to what you saw in Chapter 1 in order to show how casting can resolve these issues:

```
int x = 1.0;  // DOES NOT COMPILE
short y = 1921222;  // DOES NOT COMPILE
int z = 9f;  // DOES NOT COMPILE
long t = 192301398193810323;  // DOES NOT COMPILE
```

The first statement does not compile because you are trying to assign a double `1.0` to an integer value. Even though the value is a mathematic integer, by adding `.0`, you're instructing the compiler to treat it as a `double`. The second statement does not compile because the literal value `1921222` is outside the range of `short` and the compiler detects this. The third statement does not compile because of the `f` added to the end of the number that instructs the compiler to treat the number as floating-point value. Finally, the last statement does not compile because Java interprets the literal as an `int` and notices that the value is larger than `int` allows. The literal would need a postfix `L` to be considered a `long`.

Casting Primitive Values

We can fix the examples in the previous section by casting the results to a smaller data type. Casting primitives is required any time you are going from a larger numerical data type to a smaller numerical data type, or converting from a floating-point number to an integral value.

```
int x = (int)1.0;
short y = (short)1921222;  // Stored as 20678
```

```
int z = (int)9l;
long t = 1923013981938103231;
```

Overflow and Underflow

The expressions in the previous example now compile, although there's a cost. The second value, `1,921,222`, is too large to be stored as a `short`, so numeric overflow occurs and it becomes `20,678`. *Overflow* is when a number is so large that it will no longer fit within the data type, so the system "wraps around" to the next lowest value and counts up from there. There's also an analogous *underflow*, when the number is too low to fit in the data type.

This is beyond the scope of the exam, but something to be careful of in your own code. For example, the following statement outputs a negative number:

```
System.out.print(2147483647+1);  // -2147483648
```

Since `2147483647` is the maximum `int` value, adding any strictly positive value to it will cause it to wrap to the next negative number.

Let's return to one of our earlier examples for a moment:

```
short x = 10;
short y = 3;
short z = x * y;  // DOES NOT COMPILE
```

Based on everything you have learned up until now, can you understand why the last line of this statement will not compile? If you remember, `short` values are automatically promoted to `int` when applying any arithmetic operator, with the resulting value being of type `int`. Trying to set a `short` variable to an `int` results in a compiler error, as Java thinks you are trying to implicitly convert from a larger data type to a smaller one.

There are times that you may want to override the default behavior of the compiler. For example, in the preceding example, we know the result of `10 * 3` is `30`, which can easily fit into a `short` variable. If you need the result to be a `short`, though, you can override this behavior by casting the result of the multiplication:

```
short x = 10;
short y = 3;
short z = (short)(x * y);
```

By performing this explicit cast of a larger value into a smaller data type, you are instructing the compiler to ignore its default behavior. In other words, you are telling the compiler that you have taken additional steps to prevent overflow or underflow. It is also possible that in your particular application and scenario, overflow or underflow would result in acceptable values.

Compound Assignment Operators

Besides the simple assignment operator, `=`, there are also numerous *compound assignment operators*. Only two of the compound operators listed in Table 2.1 are required for the exam, `+=` and `-=`. Complex operators are really just glorified forms of the simple assignment operator, with a built-in arithmetic or logical operation that applies the left- and right-hand sides of the statement and stores the resulting value in a variable in the left-hand side of the statement. For example, the following two statements after the declaration of `x` and `z` are equivalent:

```
int x = 2, z = 3;
x = x * z;  // Simple assignment operator
x *= z;  // Compound assignment operator
```

The left-hand side of the compound operator can only be applied to a variable that is already defined and cannot be used to declare a new variable. In the previous example, if `x` was not already defined, then the expression `x *= z` would not compile.

Compound operators are useful for more than just shorthand—they can also save us from having to explicitly cast a value. For example, consider the following example, in which the last line will not compile due to the result being promoted to a `long` and assigned to an `int` variable:

```
long x = 10;
int y = 5;
y = y * x;  // DOES NOT COMPILE
```

Based on the last two sections, you should be able to spot the problem in the last line. This last line could be fixed with an explicit cast to `(int)`, but there's a better way using the compound assignment operator:

```
long x = 10;
int y = 5;
y *= x;
```

The compound operator will first cast `x` to a `long`, apply the multiplication of two `long` values, and then cast the result to an `int`. Unlike the previous example, in which the compiler threw an exception, in this example we see that the compiler will automatically cast the resulting value to the data type of the value on the left-hand side of the compound operator.

One final thing to know about the assignment operator is that the result of the assignment is an expression in and of itself, equal to the value of the assignment. For example, the following snippet of code is perfectly valid, if not a little odd looking:

```
long x = 5;
long y = (x=3);
System.out.println(x); // Outputs 3
System.out.println(y); // Also, outputs 3
```

The key here is that `(x=3)` does two things. First, it sets the value of the variable `x` to be `3`. Second, it returns a value of the assignment, which is also `3`. The exam creators are fond of inserting the assignment operator `=` in the middle of an expression and using the value of the assignment as part of a more complex expression.

Relational Operators

We now move on to *relational operators*, which compare two expressions and return a `boolean` value. The first four relational operators (see Table 2.3) are applied to numeric primitive data types only. If the two numeric operands are not of the same data type, the smaller one is promoted in the manner as previously discussed.

TABLE 2.3 Relational operators

`<`	Strictly less than
`<=`	Less than or equal to
`>`	Strictly greater than
`>=`	Greater than or equal to

Let's look at examples of these operators in action:

```
int x = 10, y = 20, z = 10;
System.out.println(x < y);  // Outputs true
System.out.println(x <= y);  // Outputs true
System.out.println(x >= z);  // Outputs true
System.out.println(x > z);  // Outputs false
```

Notice that the last example outputs `false`, because although `x` and `z` are the same value, `x` is not strictly greater than `z`.

The fifth relational operator (Table 2.4) is applied to object references and classes or interfaces.

TABLE 2.4

`a instanceof b`	

The `instanceof` operator, while useful for determining whether an arbitrary object is a member of a particular class or interface, is out of scope for the OCA exam.

Logical Operators

If you have studied computer science, you may have already come across logical operators before. If not, no need to panic—we'll be covering them in detail in this section.

The *logical operators*, (&), (|), and (^), may be applied to both numeric and `boolean` data types. When they're applied to `boolean` data types, they're referred to as *logical operators*. Alternatively, when they're applied to numeric data types, they're referred to as *bitwise operators*, as they perform bitwise comparisons of the bits that compose the number. For the exam, though, you don't need to know anything about numeric bitwise comparisons, so we'll leave that educational aspect to other books.

You should familiarize with the truth tables in Figure 2.1, where `x` and `y` are assumed to be `boolean` data types.

FIGURE 2.1 The logical true tables for &, |, and ^

x & y (AND)

	y = true	y = false
x = true	true	false
x = false	false	false

x | y (INCLUSIVE OR)

	y = true	y = false
x = true	true	true
x = false	true	false

x ^ y (EXCLUSIVE OR)

	y = true	y = false
x = true	false	true
x = false	true	false

Finally, we present the conditional operators, && and ||, which are often referred to as short-circuit operators. The *short-circuit operators* are nearly identical to the logical operators, & and |, respectively, except that the right-hand side of the expression may never be evaluated if the final result can be determined by the left-hand side of the expression. For example, consider the following statement:

```
boolean x = true || (y < 4);
```

Referring to the truth tables, the value `x` can only be false if both sides of the expression are false. Since we know the left-hand side is true, there's no need to evaluate the right-hand side, since no value of `y` will ever make the value of `x` anything other than true. It may help you to illustrate this concept by executing the previous line of code for various values of `y`.

A more common example of where short-circuit operators are used is checking for `null` objects before performing an operation, such as this:

```
if(x != null && x.getValue() < 5) {
  // Do something
}
```

In this example, if `x` was *null*, then the short-circuit prevents a `NullPointerException` from ever being thrown, since the evaluation of `x.getValue() < 5` is never reached. Alternatively, if we used a logical `&`, then both sides would always be evaluated and when `x` was `null` this would throw an exception:

```
if(x != null & x.getValue() < 5) { // Throws an exception if x is null
  // Do something
}
```

Be wary of short-circuit behavior on the exam, as questions are known to alter a variable on the right-hand side of the expression that may never be reached. For example, what is the output of the following code?

```
int x = 6;
boolean y = (x >= 6) || (++x <= 7);
System.out.println(x);
```

Because `x >= 6` is `true`, the increment operator on the right-hand side of the expression is never evaluated, so the output is `6`.

Equality Operators

Determining equality in Java can be a nontrivial endeavor as there's a semantic difference between "two objects are the same" and "two objects are equivalent." It is further complicated by the fact that for numeric and `boolean` primitives, there is no such distinction.

Let's start with the basics, the *equals* operator `==` and *not equals* operator `!=`. Like the relational operators, they compare two operands and return a `boolean` value about whether the expressions or values are equal, or not equal, respectively.

The equality operators are used in one of three scenarios:

1. Comparing two numeric primitive types. If the numeric values are of different data types, the values are automatically promoted as previously described. For example, `5 == 5.00` returns `true` since the left side is promoted to a `double`.
2. Comparing two `boolean` values.
3. Comparing two objects, including `null` and `String` values.

The comparisons for equality are limited to these three cases, so you cannot mix and match types. For example, each of the following would result in a compiler error:

```
boolean x = true == 3;  // DOES NOT COMPILE
boolean y = false != "Giraffe";  // DOES NOT COMPILE
boolean z = 3 == "Kangaroo";  // DOES NOT COMPILE
```

Pay close attention to the data types when you see an equality operator on the exam. The exam creators also have a habit of mixing assignment operators and equality operators, as in the following snippet:

```
boolean y = false;
boolean x = (y = true);
System.out.println(x);  // Outputs true
```

At first glance, you might think the output should be `false`, and if the expression was `(y == true)`, then you would be correct. In this example, though, the expression is assigning the value of `true` to `y`, and as you saw in the section on assignment operators, the assignment itself has the value of the assignment. Therefore, the output would be `true`.

For object comparison, the equality operator is applied to the references to the objects, not the objects they point to. Two references are equal if and only if they point to the same object, or both point to `null`. Let's take a look at some examples:

```
File x = new File("myFile.txt");
File y = new File("myFile.txt");
File z = x;
System.out.println(x == y);  // Outputs false
System.out.println(x == z);  // Outputs true
```

Even though all of the variables point to the same file information, only two, `x` and `z`, are equal in terms of `==`. In this example, as well as during the OCA exam, you may be presented with classnames that are unfamiliar, such as `File`. Many times you can answer questions about these classes without knowing the specific details of these classes. In particular, you should be able to answer questions that indicate `x` and `y` are two separate and distinct objects, even if you do not know the data types of these objects.

In Chapter 3, "Core Java APIs," we'll continue the discussion of object equality by introducing what it means for two different objects to be equivalent. We'll also cover `String` equality and show how this can be a nontrivial topic.

Understanding Java Statements

Java operators allow you to create a lot of complex expressions, but they're limited in the manner in which they can control program flow. For example, imagine you want a section of code to only be executed under certain conditions that cannot be evaluated until

runtime. Or suppose you want a particular segment of code to repeat once for every item in some list.

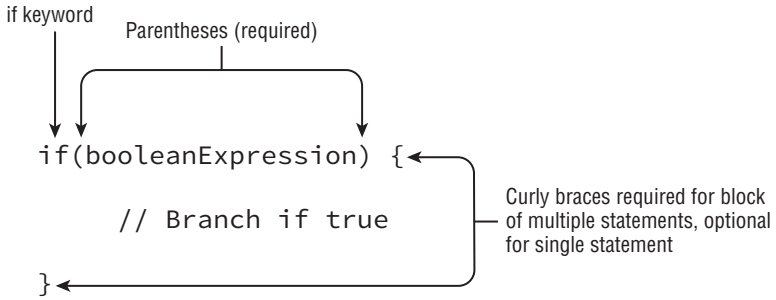
As you may recall from Chapter 1, a Java *statement* is a complete unit of execution in Java, terminated with a semicolon (;). For the remainder of the chapter, we'll be introducing you to various Java control flow statements. *Control flow statements* break up the flow of execution by using decision making, looping, and branching, allowing the application to selectively execute particular segments of code.

These statements can be applied to single expressions as well as a block of Java code. As described in the previous chapter, a *block* of code in Java is a group of zero or more statements between balanced braces, ({}), and can be used anywhere a single statement is allowed.

The *if-then* Statement

Often, we only want to execute a block of code under certain circumstances. The *if-then* statement, as shown in Figure 2.2, accomplishes this by allowing our application to execute a particular block of code if and only if a `boolean` expression evaluates to `true` at runtime.

FIGURE 2.2 The structure of an `if-then` statement



For example, imagine we had a function that used the hour of day, an integer value from `0` to `23`, to display a message to the user:

```
if(hourOfDay < 11)
      System.out.println("Good Morning");
```

If the hour of the day is less than 11, then the message will be displayed. Now let's say we also wanted to increment some value, `morningGreetingCount`, every time the greeting is printed. We could write the `if-then` statement twice, but luckily Java offers us a more natural approach using a block:

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
  morningGreetingCount++;
}
```

The block allows multiple statements to be executed based on the `if-then` evaluation. Notice that the first statement didn't contain a block around the print section, but it easily could have. For readability, it is considered good coding practice to put blocks around the execution component of `if-then` statements, as well as many other control flow statements, although it is not required.

Watch Indentation and Braces

One area that the exam writers will try to trip you up is on `if-then` statements without braces (`{}`). For example, take a look at this slightly modified form of our example:

```
if(hourOfDay < 11)
  System.out.println("Good Morning");
  morningGreetingCount++;
```

Based on the indentation, you might be inclined to think the variable `morningGreeting-Count` is only going to be incremented if the `hourOfDay` is less than `11`, but that's not what this code does. It will execute the print statement only if the condition is met, but it will always execute the increment operation.

Remember that in Java, unlike some other programming languages, tabs are just whitespace and are not evaluated as part of the execution. When you see a control flow statement in a question, be sure to trace the open and close braces of the block and ignore any indentation you may come across.

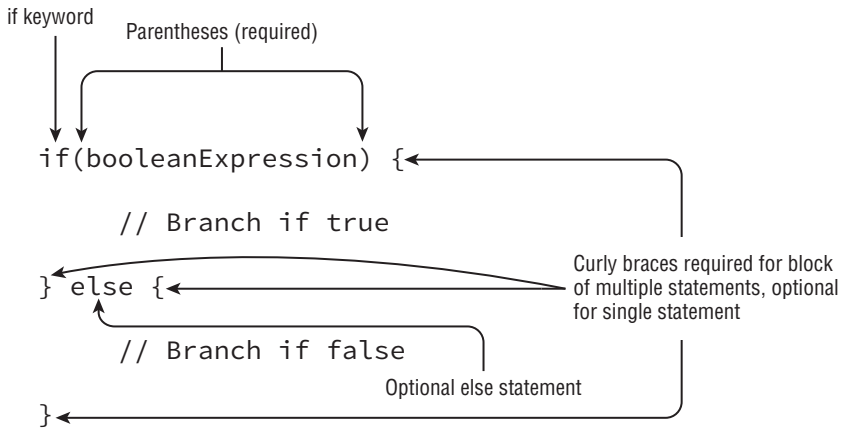
The *if-then-else* Statement

Let's expand our example a little. What if we want to display a different message if it is 11 a.m. or later? Could we do it using only the tools we have? Of course we can!

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
}
if(hourOfDay >= 11) {
  System.out.println("Good Afternoon");
}
```

This seems a bit redundant, though, since we're performing an evaluation on `hourOfDay` twice. It's also wasteful because in some circumstances the cost of the `boolean` expression we're evaluating could be computationally expensive. Luckily, Java offers us a more useful approach in the form of an *if-then-else* statement, as shown in Figure 2.3.

FIGURE 2.3 The structure of an `if-then-else` statement



Let's return to this example:

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
} else {
  System.out.println("Good Afternoon");
}
```

Now our code is truly branching between one of the two possible options, with the `boolean` evaluation happening only once. The `else` operator takes a statement or block of statement, in the same manner as the `if` statement does. In this manner, we can append additional `if-then` statements to an `else` block to arrive at a more refined example:

```
if(hourOfDay < 11) {
  System.out.println("Good Morning");
} else if(hourOfDay < 15) {
  System.out.println("Good Afternoon");
} else {
  System.out.println("Good Evening");
}
```

In this example, the Java process will continue execution until it encounters an `if-then` statement that evaluates to `true`. If neither of the first two expressions are `true`, it will execute the final code of the `else` block. One thing to keep in mind in creating complex

`if-then-else` statements is that order is important. For example, see what happens if we reorder the previous snippet of code as follows:

```
if(hourOfDay < 15) {
  System.out.println("Good Afternoon");
} else if(hourOfDay < 11) {
  System.out.println("Good Morning");  // UNREACHABLE CODE
} else {
  System.out.println("Good Evening");
}
```

For hours of the day less than 11, this code behaves very differently than the previous set of code. See if you can determine why the second block can never be executed regardless of the value of `hourOfDay`.

If a value is less than `11`, then it must be also less than `15` by definition. Therefore, if the second branch in the example can be reached, the first branch can also be reached. Since execution of each branch is mutually exclusive in this example—that is, only one branch can be executed—if the first branch is executed, then the second cannot be executed. Therefore, there is no way the second branch will ever be executed, and the code is deemed unreachable.

Verifying the *if* Statement Evaluates to a Boolean Expression

Another common place the exam may try to lead you astray is by providing code where the `boolean` expression inside the `if-then` statement is not actually a `boolean` expression. For example, take a look at the following lines of code:

```
int x = 1;
if(x) {  // DOES NOT COMPILE
  ...
}
```

This statement may be valid in some other programming and scripting languages, but not in Java, where `0` and `1` are not considered `boolean` values. Also, be wary of assignment operators being used as if they were equals `==` operators in `if-then` statements:

```
int x = 1;
if(x = 5) {  // DOES NOT COMPILE
  ...
}
```

Ternary Operator

Now that we have discussed `if-then-else` statements, we can briefly return to our discussion of operators and present the final operator that you need to learn for the exam. The conditional operator, `? :`, otherwise known as the *ternary operator*, is the only operator that takes three operands and is of the form:

```
booleanExpression ? expression1 : expression2
```

The first operand must be a `boolean` expression, and the second and third can be any expression that returns a value. The ternary operation is really a condensed form of an `if-then-else` statement that returns a value. For example, the following two snippets of code are equivalent:

```
int y = 10;
final int x;
if(y > 5) {
  x = 2 * y;
} else {
  x = 3 * y;
}
```

Compare the previous code snippet with the following equivalent ternary operator code snippet:

```
int y = 10;
int x = (y > 5) ? (2 * y) : (3 * y);
```

Note that it is often helpful for readability to add parentheses around the expressions in ternary operations, although it is certainly not required.

There is no requirement that second and third expressions in ternary operations have the same data types, although it may come into play when combined with the assignment operator. Compare the following two statements:

```
System.out.println((y > 5) ? 21 : "Zebra");
int animal = (y < 91) ? 9 : "Horse";  // DOES NOT COMPILE
```

Both expressions evaluate similar `boolean` values and return an `int` and a `String`, although only the first line will compile. The `System.out.println()` does not care that the statements are completely different types, because it can convert both to `String`. On the other hand, the compiler does know that `"Horse"` is of the wrong data type and cannot be assigned to an `int`; therefore, it will not allow the code to be compiled.

Ternary Expression Evaluation

As of Java 7, only one of the right-hand expressions of the ternary operator will be evaluated at runtime. In a manner similar to the short-circuit operators, if one of the two right-hand expressions in a ternary operator performs a side effect, then it may not be applied at runtime. Let's illustrate this principle with the following example:

```
int y = 1;
int z = 1;
final int x = y<10 ? y++ : z++;
System.out.println(y+","+z); // Outputs 2,1
```

Notice that since the left-hand `boolean` expression was `true`, only `y` was incremented. Contrast the preceding example with the following modification:

```
int y = 1;
int z = 1;
final int x = y>=10 ? y++ : z++;
System.out.println(y+","+z); // Outputs 1,2
```

Now that the left-hand `boolean` expression evaluates to `false`, only `z` was incremented. In this manner, we see how the expressions in a ternary operator may not be applied if the particular expression is not used.

For the exam, be wary of any question that includes a ternary expression in which a variable is modified in one of the right-hand side expressions.

The *switch* Statement

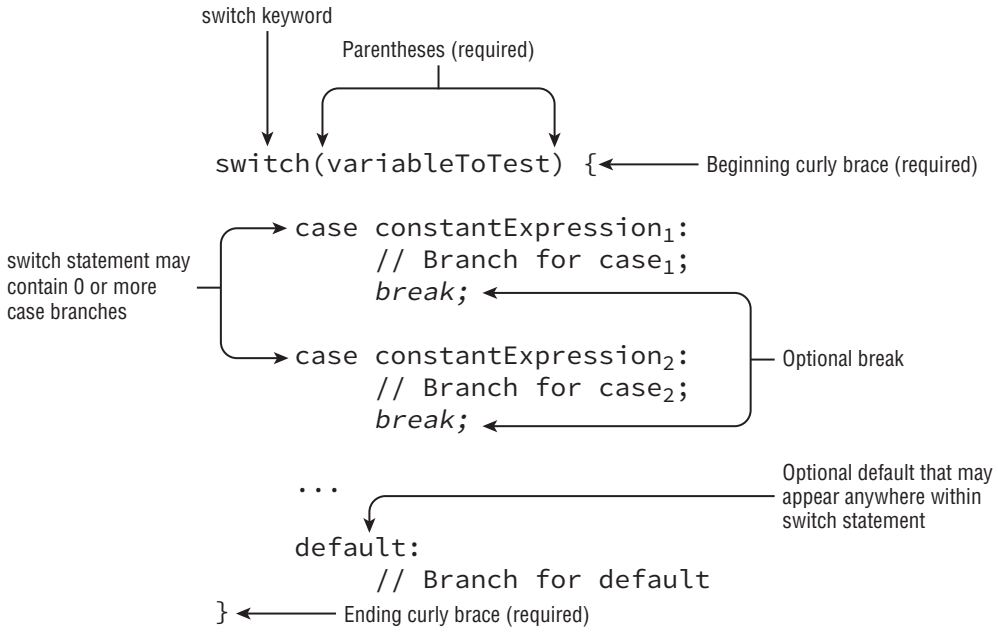
We now expand on our discussion of `if-then-else` statements by discussing a `switch` statement. A *switch* statement, as shown in Figure 2.4, is a complex decision-making structure in which a single value is evaluated and flow is redirected to the first matching branch, known as a *case* statement. If no such `case` statement is found that matches the value, an optional *default* statement will be called. If no such `default` option is available, the entire `switch` statement will be skipped.

Supported Data Types

As shown in Figure 2.4, a `switch` statement has a target variable that is not evaluated until runtime. Prior to Java 5.0, this variable could only be `int` values or those values that could be promoted to `int`, specifically `byte`, `short`, `char`, or `int`. When `enum` was added in Java 5.0, support was added to `switch` statements to support `enum` values. In Java 7, `switch`

statements were further updated to allow matching on `String` values. Finally, the `switch` statement also supports any of the primitive numeric wrapper classes, such as `Byte`, `Short`, `Character`, or `Integer`.

FIGURE 2.4 The structure of a `switch` statement



For the exam, we recommend you memorize this list. Note that `boolean` and `long`, and their associated wrapper classes, are not supported by `switch` statements.

By final constant, we mean that the variable must be marked with the final modifier and initialized with a literal value in the same expression in which it is declared.

Let's look at a simple example using the day of the week, with 0 for Sunday, 1 for Monday, and so on:

```
int dayOfWeek = 5;
switch(dayOfWeek) {
  default:
    System.out.println("Weekday");
    break;
  case 0:
    System.out.println("Sunday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
}
```

With a value of dayOfWeek of 5, this code will output:

```
Weekday
```

The first thing you may notice is that there is a break statement at the end of each case and default section. We'll discuss break statements in detail when we discuss loops, but for now all you need to know is that they terminate the switch statement and return flow control to the enclosing statement. As we'll soon see, if you leave out the break statement, flow will continue to the next proceeding case or default block automatically.

Another thing you might notice is that the default block is not at the end of the switch statement. There is no requirement that the case or default statements be in a particular order, unless you are going to have pathways that reach multiple sections of the switch block in a single execution.

To illustrate both of the preceding points, consider the following variation:

```
int dayOfWeek = 5;
switch(dayOfWeek) {
  case 0:
    System.out.println("Sunday");
  default:
    System.out.println("Weekday");
  case 6:
    System.out.println("Saturday");
    break;
}
```

This code looks a lot like the previous example except two of the `break` statements have been removed and the order has been changed. This means that for the given value of `dayOfWeek`, `5`, the code will jump to the `default` block and then execute all of the proceeding `case` statements in order until it finds a `break` statement or finishes the structure:

```
Weekday
Saturday
```

The order of the `case` and `default` statements is now important since placing the `default` statement at the end of the `switch` statement would cause only one word to be output.

What if the value of `dayOfWeek` was `6` in this example? Would the `default` block still be executed? The output of this example with `dayOfWeek` set to `6` would be:

```
Saturday
```

Even though the `default` block was before the `case` block, only the `case` block was executed. If you recall the definition of the `default` block, it is only branched to if there is no matching `case` value for the `switch` statement, regardless of its position within the `switch` statement.

Finally, if the value of `dayOfWeek` was `0`, all three statements would be output:

```
Sunday
Weekday
Saturday
```

Notice that in this last example, the `default` is executed since there was no `break` statement at the end of the preceding `case` block. While the code will not branch to the `default` statement if there is a matching `case` value within the `switch` statement, it will execute the `default` statement if it encounters it after a `case` statement for which there is no terminating `break` statement.

The exam creators are fond of `switch` *examples that are missing* `break` *statements!* When evaluating `switch` statements on the exam, always consider that multiple branches may be visited in a single execution.

We conclude our discussion on `switch` statements by acknowledging that the data type for `case` statements must all match the data type of the `switch` variable. As already discussed, the `case` statement value must also be a literal, `enum` constant, or `final` constant variable. For example, given the following `switch` statement, notice which `case` statements will compile and which will not:

```
private int getSortOrder(String firstName, final String lastName) {
  String middleName = "Patricia";
  final String suffix = "JR";
  int id = 0;
  switch(firstName) {
    case "Test":
      return 52;
```

```
    case middleName:  // DOES NOT COMPILE
      id = 5;
      break;
    case suffix:
      id = 0;
      break;
    case lastName:  // DOES NOT COMPILE
      id = 8;
      break;
    case 5:  // DOES NOT COMPILE
      id = 7;
      break;
    case 'J':  // DOES NOT COMPILE
      id = 10;
      break;
    case java.time.DayOfWeek.SUNDAY:  // DOES NOT COMPILE
      id=15;
      break;
  }
  return id;
}
```

The first `case` statement compiles without issue using a `String` literal and is a good example of how a `return` statement, like a `break` statement, can be used to exit the `switch` statement early. The second `case` statement does not compile because `middleName` is not a `final` variable, despite having a known value at this particular line of execution. The third `case` statement compiles without issue because `suffix` is a `final` constant variable.

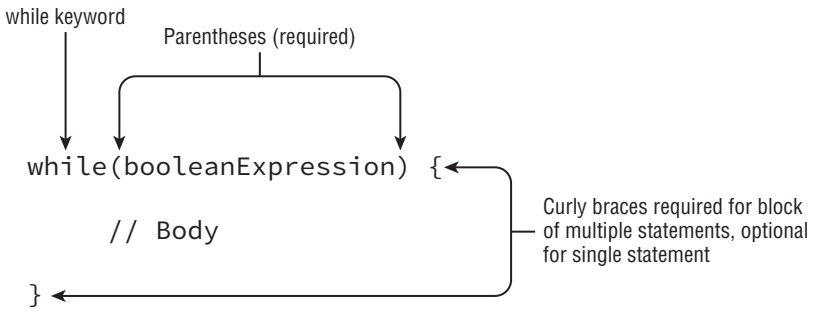
In the fourth `case` statement, despite `lastName` being `final`, it is not constant as it is passed to the function; therefore, this line does not compile as well. Finally, the last three `case` statements don't compile because none of them have a matching type of `String`; the last one is an `enum` value.

The *while* Statement

A repetition control structure, which we refer to as a *loop*, executes a statement of code multiple times in succession. By using nonconstant variables, each repetition of the statement may be different. For example, a statement that iterates over a list of unique names and outputs them would encounter a new name on every execution of the loop.

The simplest such repetition control structure in Java is the *while* statement, described in Figure 2.5. Like all repetition control structures, it has a termination condition, implemented as a `boolean` expression, that will continue as long as the expression evaluates to `true`.

FIGURE 2.5 The structure of a `while` statement



As shown in Figure 2.5, a `while` loop is similar to an `if-then` statement in that it is composed of a `boolean` expression and a statement, or block of statements. During execution, the `boolean` expression is evaluated before each iteration of the loop and exits if the evaluation returns `false`. It is important to note that a `while` loop may terminate after its first evaluation of the `boolean` expression. In this manner, the statement block may never be executed.

Let's return to our mouse example from Chapter 1 and show a loop can be used to model a mouse eating a meal:

```
int roomInBelly = 5;

public void eatCheese(int bitesOfCheese) {
  while (bitesOfCheese > 0 && roomInBelly > 0) {
    bitesOfCheese--;
    roomInBelly--;
  }
  System.out.println(bitesOfCheese+" pieces of cheese left");
}
```

This method takes an amount of food, in this case cheese, and continues until the mouse has no room in its belly or there is no food left to eat. With each iteration of the loop, the mouse "eats" one bite of food and loses one spot in its belly. By using a compound `boolean` statement, you ensure that the `while` loop can end for either of the conditions.

Real World Scenario

Infinite Loops

Consider the following segment of code:

```
int x = 2;
int y = 5;
while(x < 10)
  y++;
```

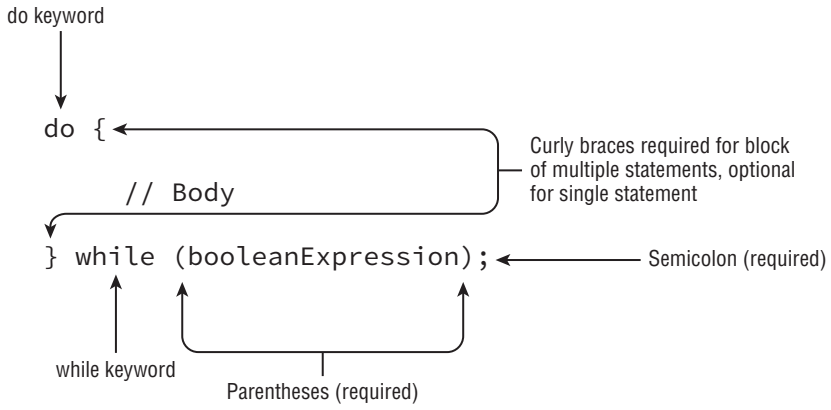
You may notice one glaring problem with this statement: it will never end! The `boolean` expression that is evaluated prior to each loop iteration is never modified, so the expression `(x < 10)` will always evaluate to `true`. The result is that the loop will never end, creating what is commonly referred to as an *infinite loop*.

Infinite loops are something you should be aware of any time you create a loop in your application. You should be absolutely certain that the loop will eventually terminate under some condition. First, make sure the loop variable is modified. Then, ensure that the termination condition will be eventually reached in all circumstances. As you'll see in the upcoming section "Understanding Advanced Flow Control," a loop may also exit under other conditions such as a `break` statement.

The *do-while* Statement

Java also allows for the creation of a *do-while* loop, which like a `while` loop, is a repetition control structure with a termination condition and statement, or block of statements, as shown in Figure 2.6. Unlike a `while` loop, though, a `do-while` loop guarantees that the statement or block will be executed at least once.

FIGURE 2.6 The structure of a `do-while` statement



The primary difference between the syntactic structure of a `do-while` loop and a `while` loop is that a `do-while` loop purposely orders the statement or block of statements before the conditional expression, in order to reinforce that the statement will be executed before the expression is ever evaluated. For example, take a look at the output of the following statements:

```
int x = 0;
do {
  x++;
} while(false);
System.out.println(x);  // Outputs 1
```

Java will execute the statement block first, and then check the loop condition. Even though the loop exits right away, the statement block was still executed once and the program outputs a `1`.

When to Use *while* vs. *do-while* Loops

In practice, it might be difficult to determine when you should use a `while` loop and when you should use a `do-while` loop. The short answer is that it does not actually matter. Any `while` loop can be converted to a `do-while` loop, and vice versa. For example, compare this `while` loop:

```
while(x > 10) {
  x--;
}
```

and this `do-while` loop:

```
if(x > 10) {
  do {
    x--;
  } while(x > 10);
}
```

Though one of the loops is certainly easier to read, they are functionally equivalent. Java recommends you use a `while` loop when a loop might not be executed at all and a `do-while` loop when the loop is executed at least once. But determining whether you should use a `while` loop or a `do-while` loop in practice is sometimes about personal preference and code readability.

For example, although the first statement is shorter, the second has the advantage that you could leverage the existing `if-then` statement and perform some other operation in a new `else` branch, as shown in the following example:

continues

continued

```
if(x > 10) {
  do {
    x--;
  } while(x > 10);
} else {
  x++;
}
```

The *for* Statement

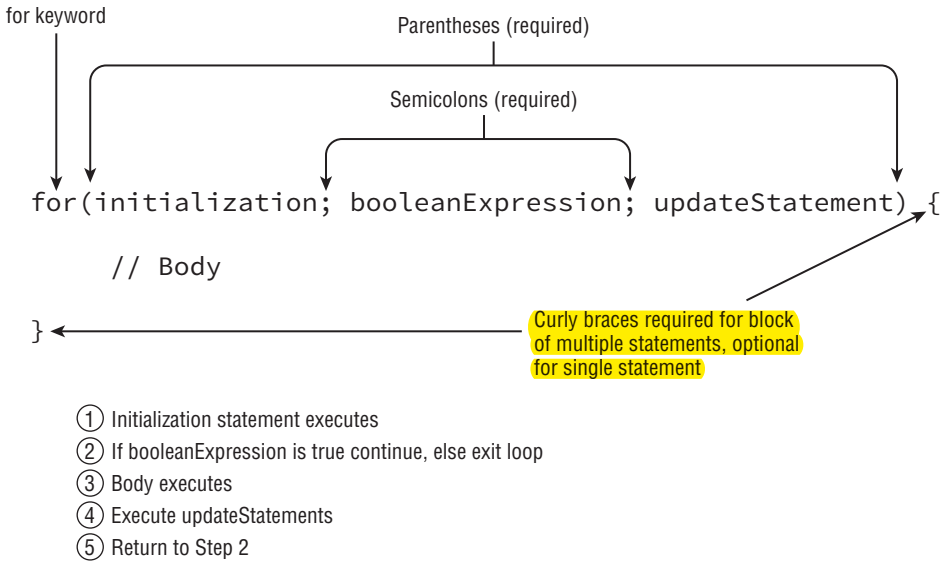
Now that you can build applications with simple `while` and `do-while` statements, we expand our discussion of loops to a more complex repetition control structure called a `for` loop.

Starting in Java 5.0, there are now two types of `for` statements. The first is referred to as the basic `for` loop, and the second is often called the enhanced `for` loop. For clarity, we'll refer to the enhanced `for` loop as the `for-each` statement throughout the book.

The Basic *for* Statement

A basic *for loop* has the same conditional `boolean` expression and statement, or block of statements, as the other loops you have seen, as well as two new sections: an *initialization block* and an *update* statement. Figure 2.7 shows how these components are laid out.

FIGURE 2.7



Although Figure 2.7 might seem a little confusing and almost arbitrary at first, the organization of the components and flow allow us to create extremely powerful statements in a very small amount of space that otherwise would take multiple lines with a standard `while` loop. Note that each section is separated by a semicolon. The initialization and update sections may contain multiple statements, separated by commas.

Variables declared in the initialization block of a `for` loop have limited scope and are only accessible within the `for` loop. Be wary of any exam questions in which a variable declared within the initialization block of a `for` loop is available outside the loop. Alternatively, variables declared before the `for` loop and assigned a value in the initialization block may be used outside the `for` loop because their scope precedes the `for` loop creation.

Let's take a look at an example that prints the numbers `0` to `9`:

```
for(int i = 0; i < 10; i++) {
  System.out.print(i + " ");
}
```

The local variable `i` is initialized first to `0`. The variable `i` is only in scope for the duration of the loop and is not available outside the loop once the loop has completed. Like a `while` loop, the `boolean` condition is evaluated on every iteration of the loop *before* the loop executes. Since it returns `true`, the loop executes and outputs the `0` followed by a space. Next, the loop executes the update section, which in this case increases the value of `i` to `1`. The loop then evaluates the `boolean` expression a second time, and the process repeats multiple times, printing:

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

On the 10th iteration of the loop, the value of `i` reaches `9` and is incremented by `1` to reach `10`. On the 11th iteration of the loop, the `boolean` expression is evaluated and since `(10 < 10)` returns `false`, the loop terminates without executing the statement loop body.

Although most `for` loops you are likely to encounter in practice will be well defined and similar to the previous example, there are a number of variations and edge cases you could see on the exam. You should familiarize yourself with the following five examples: variations of these are likely to be seen on the exam.

Let's tackle some examples for illustrative purposes:

1. Creating an Infinite Loop

```
for( ; ; ) {
  System.out.println("Hello World");
}
```

Although this `for` loop may look like it will throw a compiler error, it will in fact compile and run without issue. It is actually an infinite loop that will print the same statement repeatedly. This example reinforces the fact that the components of the `for` loop are each optional. Note that the semicolons separating the three sections are required, as `for( ; )` and `for( )` will not compile.

2. Adding Multiple Terms to the `for` Statement

```
int x = 0;
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
  System.out.print(y + " ");
}
System.out.print(x);
```

This code demonstrates three variations of the `for` loop you may not have seen. First, you can declare a variable, such as `x` in this example, before the loop begins and use it after it completes. Second, your initialization block, `boolean` expression, and update statements can include extra variables that may not reference each other. For example, `z` is defined in the initialization block and is never used. Finally, the `update` statement can modify multiple variables. This code will print the following when executed:

```
0 1 2 3 4
```

Keep this example in mind when we look at the next three examples, none of which compile.

3. Redeclaring a Variable in the Initialization Block

```
int x = 0;
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {   // DOES NOT COMPILE
  System.out.print(x + " ");
}
```

This example looks similar to the previous one, but it does not compile because of the initialization block. The difference is that `x` is repeated in the initialization block after already being declared before the loop, resulting in the compiler stopping because of a duplicate variable declaration. We can fix this loop by changing the declaration of `x` and `y` as follows:

```
int x = 0;
long y = 10;
for(y = 0, x = 4; x < 5 && y < 10; x++, y++) {
  System.out.print(x + " ");
}
```

Note that this variation will now compile because the initialization block simply assigns a value to `x` and does not declare it.

4. Using Incompatible Data Types in the Initialization Block

```
for(long y = 0, int x = 4; x < 5 && y<10; x++, y++) {   // DOES NOT COMPILE
  System.out.print(x + " ");
}
```

This example also looks a lot like our second example, but like the third example will not compile, although this time for a different reason. The variables in the initialization block must all be of the same type. In the first example, `y` and `z` were both `long`, so the code compiled without issue, but in this example they have differing types, so the code will not compile.

5. Using Loop Variables Outside the Loop

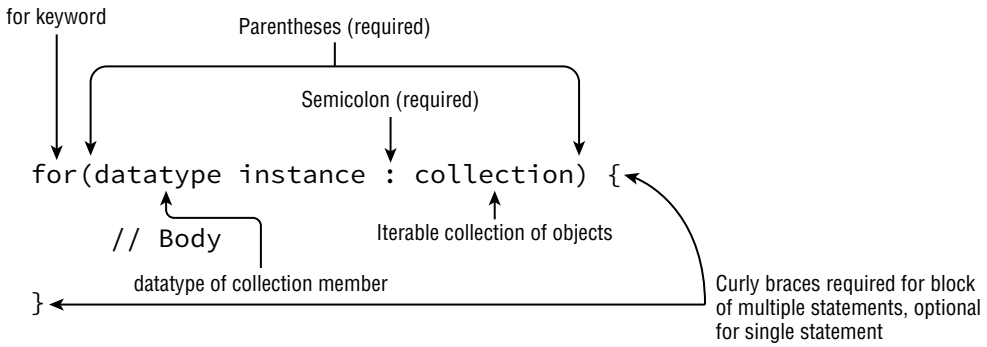
```
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {
  System.out.print(y + " ");
}
System.out.print(x);  // DOES NOT COMPILE
```

The final variation on the second example will not compile for a different reason than the previous examples. If you notice, `x` is defined in the initialization block of the loop, and then used after the loop terminates. Since `x` was only scoped for the loop, using it outside the loop will throw a compiler error.

The *for-each* Statement

Starting with Java 5.0, Java developers have had a new type of enhanced `for` loop at their disposal, one specifically designed for iterating over arrays and `Collection` objects. This enhanced `for` loop, which for clarity we'll refer to as a *for-each* loop, is shown in Figure 2.8.

FIGURE 2.8 The structure of an enhancement `for` statement



The `for-each` loop declaration is composed of an initialization section and an object to be iterated over. The right-hand side of the `for-each` loop statement must be a built-in Java array or an object whose class implements `java.lang.Iterable`, which includes most of the Java `Collections` framework. The left-hand side of the `for-each` loop must include a declaration for an instance of a variable, whose type matches the type of a member of the array or collection in the right-hand side of the statement. On each iteration of the loop, the named variable on the left-hand side of the statement is assigned a new value from the array or collection on the right-hand side of the statement.

For the OCA exam, the only members of the `Collections` framework that you need to be aware of are `List` and `ArrayList`. In this chapter, we'll show how to iterate over `List` objects, and in Chapter 3 we'll go into detail about how to create `List` objects and how they differ from traditional Java arrays.

Let's review some examples:

- What will this code output?

```
final String[] names = new String[3];
names[0] = "Lisa";
names[1] = "Kevin";
names[2] = "Roger";
for(String name : names) {
  System.out.print(name + ", ");
}
```

This code will compile and print:

```
Lisa, Kevin, Roger,
```

- What will this code output?

```
java.util.List<String> values = new java.util.ArrayList<String>();
values.add("Lisa");
values.add("Kevin");
values.add("Roger");
for(String value : values) {
  System.out.print(value + ", ");
}
```

This code will compile and print the same values:

```
Lisa, Kevin, Roger,
```

When you see a `for-each` loop on the exam, make sure the right-hand side is an array or `Iterable` object and the left-hand side has a matching type. For example, the two examples that follow will not compile.

- Why will the following fail to compile?

```
String names = "Lisa";
for(String name : names) {   // DOES NOT COMPILE
  System.out.print(name + " ");
}
```

In this example, the `String` `names` is not an array, nor does it implement `java.lang.Iterable`, so the compiler will throw an exception since it does not know how to iterate over the `String`.

- Why will the following fail to compile?

```
String[] names = new String[3];
for(int name : names) {  // DOES NOT COMPILE
  System.out.print(name + " ");
}
```

This code will fail to compile because the left-hand side of the `for-each` statement does not define an instance of `String`. Notice that in this last example, the array is initialized with three `null` pointer values. In and of itself, that will not cause the code to not compile, as a corrected loop would just output `null` three times.

Real World Scenario

Comparing *for* and *for-each* Loops

Since `for` and `for-each` both use the same keyword, you might be wondering how they are related. While this discussion is out of scope for the exam, let's take a moment to explore how `for-each` loops are converted to `for` loops by the compiler.

When `for-each` was introduced in Java 5, it was added as a compile-time enhancement. This means that Java actually converts the `for-each` loop into a standard `for` loop during compilation. For example, assuming `names` is an array of `String[]` as we saw in the first example, the following two loops are equivalent:

```
for(String name : names) {
  System.out.print(name + ", ");
}
for(int i=0; i < names.length; i++) {
  String name = names[i];
  System.out.print(name + ", ");
}
```

For objects that inherit `java.lang.Iterable`, there is a different, but similar, conversion. For example, assuming `values` is an instance of `List<Integer>`, as we saw in the second example, the following two loops are equivalent:

```
for(int value : values) {
  System.out.print(value + ", ");
}
for(java.util.Iterator<Integer> i = values.iterator(); i.hasNext(); ) {
  int value = i.next();
  System.out.print(value + ", ");
}
```

Notice that in the second version, there is no `update` statement as it is not required when using the `java.util.Iterator` class.

You may have noticed that in the previous `for-each` examples, there was an extra comma printed at the end of the list:

```
Lisa, Kevin, Roger,
```

While the `for-each` statement is convenient for working with lists in many cases, it does hide access to the loop iterator variable. If we wanted to print only the comma between names, we could convert the example into a standard `for` loop, as in the following example:

```
java.util.List<String> names = new java.util.ArrayList<String>();
names.add("Lisa");
names.add("Kevin");
names.add("Roger");
for(int i=0; i<names.size(); i++) {
  String name = names.get(i);
  if(i>0) {
    System.out.print(", ");
  }
  System.out.print(name);
}
```

This sample code would output the following:

```
Lisa, Kevin, Roger
```

It is also common to use a standard `for` loop over a `for-each` loop if comparing multiple elements in a loop within a single iteration, as in the following example. Notice that we skip the first loop's execution, since `value[-1]` is not defined and would throw an `IndexOutOfBoundsException` error.

```
int[] values = new int[3];
values[0] = 10;
values[1] = new Integer(5);
values[2] = 15;
for(int i=1; i<values.length; i++) {
  System.out.print(values[i]-values[i-1]);
}
```

This sample code would output the following:

```
-5, 10,
```

Despite these examples, enhanced `for-each` loops are quite useful in Java in a variety of circumstances. As a developer, though, you can always revert to a standard `for` loop if you need fine-grain control.

Understanding Advanced Flow Control

Up to now, we have been dealing with single loops that only ended when their `boolean` expression evaluated to `false`. We'll now show you other ways loops could end, or branch, and you'll see that the path taken during runtime may not be as straightforward as in previous examples.

Nested Loops

First off, loops can contain other loops. For example, consider the following code that iterates over a two-dimensional array, an array that contains other arrays as its members. We'll cover multidimensional arrays in detail in Chapter 3, but for now assume the following is how you would declare a two-dimensional array.

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
for(int[] mySimpleArray : myComplexArray) {
  for(int i=0; i<mySimpleArray.length; i++) {
    System.out.print(mySimpleArray[i]+"\t");
  }
  System.out.println();
}
```

Notice that we intentionally mix a `for` and `for-each` loop in this example. The outer loops will execute a total of three times. Each time the outer loop executes, the inner loop is executed four times. When we execute this code, we see the following output:

```
5       2       1       3
3       9       8       9
5       7       12       7
```

Nested loops can include `while` and `do-while`, as shown in this example. See if you can determine what this code will output.

```
int x = 20;
while(x>0) {
  do {
    x -= 2
  } while (x>5);
  x--;
  System.out.print(x+"\t");
}
```

The first time this loop executes, the inner loop repeats until the value of `x` is `4`. The value will then be decremented to `3` and that will be the output at the end of the first iteration of the outer loop. On the second iteration of the outer loop, the inner `do-while` will be executed once, even though `x` is already not greater than `5`. As you may recall, `do-while` statements always execute the body at least once. This will reduce the value to `1`, which will be further lowered by the decrement operator in the outer loop to `0`. Once the value reaches `0`, the outer loop will terminate. The result is that the code will output the following:

```
3       0
```

Adding Optional Labels

One thing we skipped when we presented `if-then` statements, `switch` statements, and loops is that they can all have optional labels. A *label* is an optional pointer to the head of a

statement that allows the application flow to jump to it or break from it. It is a single word that is proceeded by a colon (:). For example, we can add optional labels to one of the previous examples:

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
OUTER_LOOP:  for(int[] mySimpleArray : myComplexArray) {
  INNER_LOOP:  for(int i=0; i<mySimpleArray.length; i++) {
    System.out.print(mySimpleArray[i]+"\t");
  }
  System.out.println();
}
```

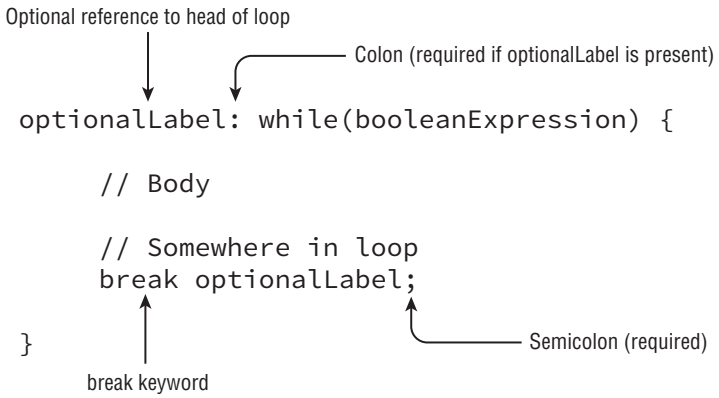
When dealing with only one loop, they add no value, but as we'll see in the next section, they are extremely useful in nested environments. Optional labels are often only used in loop structures. While this topic is not on the OCA exam, it is possible to add optional labels to control and block structures. That said, it is rarely considered good coding practice to do so.

For formatting, labels follow the same rules for identifiers. For readability, they are commonly expressed in uppercase, with underscores between words, to distinguish them from regular variables.

The *break* Statement

As you saw when working with `switch` statements, a *break* statement transfers the flow of control out to the enclosing statement. The same holds true for `break` statements that appear inside of `while`, `do-while`, and `for` loops, as it will end the loop early, as shown in Figure 2.9

FIGURE 2.9 The structure of a `break` statement



Notice in Figure 2.9 that the `break` statement can take an optional label parameter. Without a label parameter, the `break` statement will terminate the nearest inner loop it is

currently in the process of executing. The optional label parameter allows us to break out of a higher level outer loop. In the following example, we search for the first (x,y) array index position of a number within an unsorted two-dimensional array:

```
public class SearchSample {
  public static void main(String[] args) {
    int[][] list = {{1,13,5},{1,2,5},{2,7,2}};
    int searchValue = 2;
    int positionX = -1;
    int positionY = -1;
    PARENT_LOOP: for(int i=0; i<list.length; i++) {
      for(int j=0; j<list[i].length; j++) {
        if(list[i][j]==searchValue) {
          positionX = i;
          positionY = j;
          break PARENT_LOOP;
        }
      }
    }
    if(positionX==-1 || positionY==-1) {
      System.out.println("Value "+searchValue+" not found");
    } else {
      System.out.println("Value "+searchValue+" found at: " +
        "("+positionX+","+positionY+")");
    }
  }
}
```

When executed, this code will output:

```
Value 2 found at: (1,1)
```

In particular, take a look at the statement `break PARENT_LOOP`. This statement will break out of the entire loop structure as soon as the first matching value is found. Now, imagine what would happen if we replaced the body of the inner loop with the following:

```
if(list[i][j]==searchValue) {
  positionX = i;
  positionY = j;
  break;
}
```

How would this change our flow and would the output change? Instead of exiting when the first matching value is found, the program will now only exit the inner loop when the

condition is met. In other words, the structure will now find the first matching value of the last inner loop to contain the value, resulting in the following output:

```
Value 2 found at: (2,0)
```

Finally, what if we removed the `break` altogether?

```
if(list[i][j]==searchValue) {
  positionX = i;
  positionY = j;
}
```

In this case, the code will search for the last value in the entire structure that has the matching value. The output will look like this:

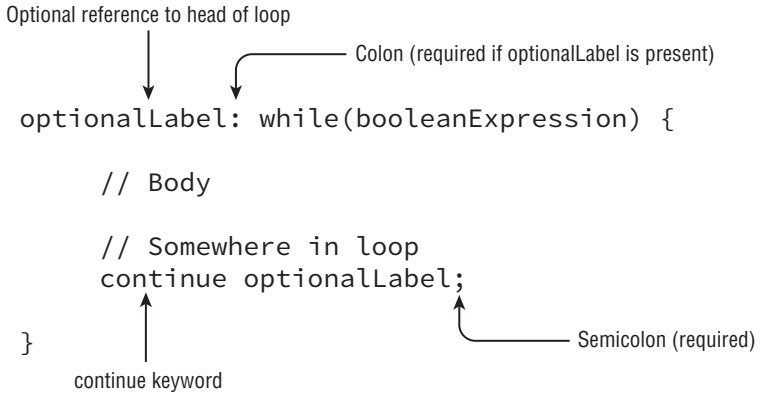
```
Value 2 found at: (2,2)
```

You can see from this example that using a label on a `break` statement in a nested loop, or not using the `break` statement at all, can cause the loop structure to behave quite differently.

The *continue* Statement

Let's now complete our discussion of advanced loop control with the *continue* statement, a statement that causes flow to finish the execution of the current loop, as shown in Figure 2.10.

FIGURE 2.10 The structure of a `continue` statement



You may notice the syntax of the `continue` statement mirrors that of the `break` statement. In fact, the statements are similar in how they are used, but with different results. While the `break` statement transfers control to the enclosing statement, the `continue` statement transfers control to the `boolean` expression that determines if the loop should continue. In other words, it ends the current iteration of the loop. Also like the `break` statement, the `continue` statement is applied to the nearest inner loop under execution using optional label statements to override this behavior. Let's take a look at the following example:

```
public class SwitchSample {
  public static void main(String[] args) {
    FIRST_CHAR_LOOP: for (int a = 1; a <= 4; a++) {
      for (char x = 'a'; x <= 'c'; x++) {
        if (a == 2 || x == 'b')
          continue FIRST_CHAR_LOOP;
        System.out.print(" " + a + x);
      }
    }
  }
}
```

With the structure as defined, the loop will return control to the parent loop any time the first value is `2` or the second value is `b`. This results in one execution of the inner loop for each of three outer loop calls. The output looks like this:

```
1a 3a 4a
```

Now, imagine we removed the `FIRST_CHAR_LOOP` label in the `continue` statement so that control is returned to the inner loop instead of the outer. See if you can understand how the output will be changed to:

```
1a 1c 3a 3c 4a 4c
```

Finally, if we remove the `continue` statement and associated `if-then` statement altogether, we arrive at a structure that outputs all the values, such as:

```
1a 1b 1c 2a 2b 2c 3a 3b 3c 4a 4b 4c
```

Table 2.5 will help remind you when labels, `break`, and `continue` statements are permitted in Java. Although for illustrative purposes our examples have included using these statements in nested loops, they can be used inside single loops as well.

TABLE 2.5 Advanced flow control usage

	Allows optional labels	Allows *break* statement	Allows *continue* statement
`if`	Yes *	No	No
`while`	Yes	Yes	Yes
`do while`	Yes	Yes	Yes
`for`	Yes	Yes	Yes
`switch`	Yes	Yes	No

* Labels are allowed for any block statement, including those that are preceded with an `if-then` statement.

Summary

This chapter covered a wide variety of topics, including dozens of Java operators, along with numerous control flow statements. Many of these operators and statements may have been new to you.

It is important that you understand how to use all of the required Java operators covered in this chapter and know how operator precedence influences the way a particular expression is interpreted. There will likely be numerous questions on the exam that appear to test one thing, such as `StringBuilder` or exception handling, when in fact the answer is related to the misuse of a particular operator that causes the application to fail to compile. When you see an operator on the exam, always check that the appropriate data types are used and that they match each other where applicable.

For statements, this chapter covered two types of control structures: decision-making controls structures, including `if-then`, `if-then-else`, and `switch` statements, as well as repetition control structures including `for`, `for-each`, `while`, and `do-while`. Remember that most of these structures require the evaluation of a particular `boolean` expression either for branching decisions or once per repetition. The `switch` statement is the only one that supports a variety of data types, including `String` variables as of Java 7.

With a `for-each` statement you don't need to explicitly write a `boolean` expression, since the compiler builds them implicitly. For clarity, we referred to an enhanced `for` loop as a `for-each` loop, but syntactically they are written as a `for` statement.

We concluded this chapter by discussing advanced control options and how flow can be enhanced through nested loops, `break` statements, and `continue` statements. Be wary of questions on the exam that use nested statements, especially ones with labels, and verify they are being used correctly.

This chapter is especially important because at least one component of this chapter will likely appear in every exam question with sample code. Many of the questions on the exam focus on proper syntactic use of the structures, as they will be a large source of questions that end in "Does not compile." You should be able to answer all of the review questions correctly or fully understand those that you answered incorrectly before moving on to later chapters.

Exam Essentials

Be able to write code that uses Java operators. This chapter covered a wide variety of operator symbols. Go back and review them several times so that you are familiar with them throughout the rest of the book.

Be able to recognize which operators are associated with which data types. Some operators may be applied only to numeric primitives, some only to `boolean` values, and some only to objects. It is important that you notice when an operator and operand(s) are mismatched, as this issue is likely to come up in a couple of exam questions.