Chapter 4
Making Decisions

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Using Operators and Decision Constructs**
  - Use Java control statements including if, if/else, switch
  - Create and use do/while, while, for and for each loops, including nested loops, use break and continue statements

Like many programming languages, Java is composed primarily of variables, operators, and statements put together in some logical order. Previously, we covered how to create and manipulate variables. Writing software is about more than managing variables, though; it is about creating applications that can make intelligent decisions. In this chapter, we present the various decision-making statements available to you within the language. This knowledge will allow you to build complex functions and class structures that you'll see throughout this book.

Creating Decision-Making 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. Imagine you want a method to be executed only under certain conditions that cannot be evaluated until runtime. For example, on rainy days, a zoo should remind patrons to bring an umbrella, or on a snowy day, the zoo might need to close. The software doesn't change, but the behavior of the software should, depending on the inputs supplied in the moment. In this

section, we will discuss decision-making statements including `if`, `else`, and `switch` statements.

Statements and Blocks

As you may recall from Chapter 2, "Java Building Blocks," 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 Chapter 2, 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. For example, the following two snippets are equivalent, with the first being a single expression and the second being a block of statements:

```
// Single statement
patrons++;

// Statement inside a block
{
   patrons++;
}
```

A statement or block often functions as the target of a decision-making statement. For example, we can prepend the decision-making `if` statement to these two examples:

```
// Single statement
if(ticketsTaken > 1)
   patrons++;

// Statement inside a block
```

```
if(ticketsTaken > 1)
{
   patrons++;
}
```

Again, both of these code snippets are equivalent. Just remember that the target of a decision-making statement can be a single statement or block of statements. For the rest of the chapter, we will use both forms to better prepare you for what you will see on the exam.

While both of the previous examples are equivalent, stylistically the second form is often preferred, even if the block has only one statement. The second form has the advantage that you can quickly insert new lines of code into the block, without modifying the surrounding structure. While either of these forms is correct, it might explain why you often see developers who always use blocks with all decision-making statements.

The *if* Statement

Oftentimes, we want to execute a block of code only under certain circumstances. The *if* statement, as shown in Figure 4.1, 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 4.1 The structure of an `if` 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` 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` evaluation. Notice that the first statement didn't contain a block around the print section, but it easily could have. As discussed in the previous section, it is often considered good coding practice to put blocks around the execution component of `if` statements, as well as many other control flow statements, although it is certainly not required.

WATCH INDENTATION AND BRACES

One area where the exam writers will try to trip you up is on `if` 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 `morningGreetingCount` 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, ignoring any indentation you may come across.

The *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 `else` statement, as shown in Figure 4.2.

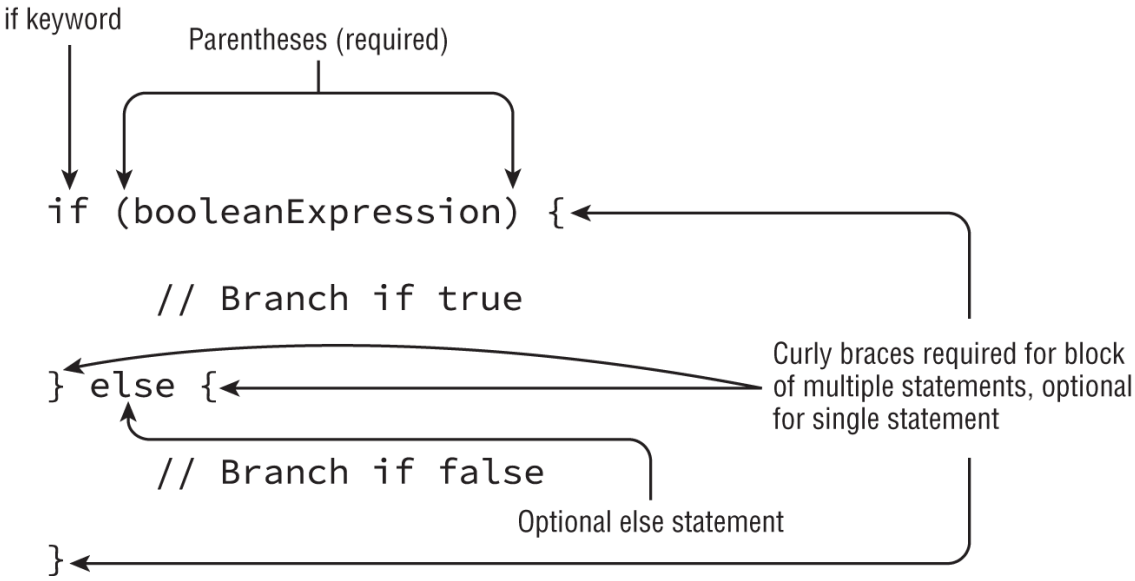

FIGURE 4.2 The structure of an `else` statement

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 statements, in the same manner as the `if` statement does. Similarly, we can append additional `if` 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` statement that evaluates to `true` . If neither of the first two expressions is `true` , it will execute the final code of the `else` block. One thing to keep in mind in creating complex `if` and `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");  // COMPILES BUT IS UNREACHABLE
} 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. Do you see 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), then if the first branch is executed, the second cannot be executed. Therefore, there is no way the second branch will ever be executed, and the code is deemed unreachable.

VERIFYING THAT 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` statement is not actually a `boolean` expression. For example, take a look at the following lines of code:

```
int hourOfDay = 1;
if(hourOfDay) {  // 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, like you saw in Chapter 3, "Operators," be wary of assignment operators being used as if they were equals (`==`) operators in `if` statements:

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

The *switch* Statement

What if we have a lot of possible branches for a single value? For example, we might want to print a different message based on the day of the week. We could certainly accomplish this with a combination of seven `if` or `else` statements, but that tends to create code that is long, difficult to read, and often not fun to maintain. For example, the following code prints a different value based on the day of the week using various different styles for each decision statement:

```
int dayOfWeek = 5;

if(dayOfWeek == 0) System.out.print("Sunday");
else if(dayOfWeek == 1)
{
   System.out.print("Monday");
}
else if(dayOfWeek == 2) {
   System.out.print("Tuesday");
} else if(dayOfWeek == 3)
   System.out.print("Wednesday");
...
```

Luckily, Java, along with many other languages, provides a cleaner approach. A *switch* statement, as shown in Figure 4.3, 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.

FIGURE 4.3 The structure of a `switch` statement

Proper Switch Syntax

Because `switch` statements can be longer than most decision-making statements, the exam may present invalid `switch` syntax to see whether you are paying attention. See if you can figure out why each of the following `switch` statements does not compile:

```
int month = 5;

switch month {  // DOES NOT COMPILE
   case 1: System.out.print("January");
}

switch (month)  // DOES NOT COMPILE
   case 1: System.out.print("January");

switch (month) {
   case 1: 2: System.out.print("January"); // DOES NOT COMPILE
}

switch (month) {
   case 1 || 2: System.out.print("January"); // DOES NOT COMPILE
}
```

The first `switch` statement does not compile because it is missing parentheses around the `switch` variable. The second statement does not compile because it is missing braces around the `switch` body. The third statement does not compile because the `case` keyword is missing before the `2:` label. Each `case` statement requires the keyword `case`, followed by a value and a colon (`:`).

Finally, the last `switch` statement does not compile because `1 || 2` uses the short-circuit `boolean` operator, which cannot be applied to numeric values. A single bitwise operator (`|`) would have allowed the code to compile, although the interpretation of this might not be what you expect. It would then only match a value of `month` that is the bitwise result of `1 | 2`, which is `3`, and would *not* match `month` having a value `1` or `2`.

You don't need to know bitwise arithmetic for the exam, but you do need to know proper syntax for `case` statements.

Notice that these last two statements both try to combine `case` statements in ways that are not valid. One last note you should be aware of for the exam: a `switch` statement is not required to contain any `case` statements. For example, this statement is perfectly valid:

```
switch (month) {}
```

For the exam, make sure you memorize the syntax used in Figure 4.3. As you will see in the next section, while some aspects of `switch` statements have changed over the years, many things have not changed.

Switch Data Types

As shown in Figure 4.3, 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`, which we refer to as *primitive numeric types*.

The `switch` statement also supports any of the wrapper class versions of these primitive numeric types, such as `Byte`, `Short`, `Character`, or `Integer`. Don't worry if you haven't seen numeric wrapper classes—we'll be covering them in Chapter 5, "Core Java APIs." For now, you just need to know that they are objects that can store primitive values.

Notice that `boolean`, `long`, `float`, and `double` are excluded from `switch` statements, as are their associated `Boolean`, `Long`, `Float`, and `Double` classes. The reasons are varied, such as `boolean` having too small a range of values and floating-point numbers having quite a wide range of values. For the exam, though, you just need to know that they are not permitted in `switch` statements.

When enumeration, denoted `enum`, was added in Java 5.0, support was added to `switch` statements to support `enum` values. An enumeration is a fixed set of constant values, which can also include methods and class variables, similar to a class definition. For the exam, you do not need to know how to create enums, but you should be aware they can be used as the target of `switch` statements.

In Java 7, `switch` statements were further updated to allow matching on `String` values. In Java 10, if the type a `var` resolves to is one of the types supported by a `switch` statement, then `var` can be used in a `switch` statement too.

SWITCH HISTORY AND CHANGES

As you can see, `switch` statements have been modified in numerous versions of Java. You don't have to worry about remembering the history—just know what types are now allowed. The history lesson is for experienced Java developers who have been using an older version of Java and may not be aware of the numerous changes to `switch` statements over the years.

But wait, there's more. Java 12 launched with a Preview release of a powerful new feature called Switch Expressions, a construct that combines `switch` statements with lambda expressions and allows `switch` statements to return a value. You won't need to know Switch Expressions for the exam, but it's just a sign that the writers of Java are far from done making enhancements to `switch` statements.

The following is a list of all data types supported by `switch` statements:

- `int` and `Integer`
- `byte` and `Byte`
- `short` and `Short`
- `char` and `Character`
- `String`
- `enum` values
- `var` (if the type resolves to one of the preceding types)

For the exam, we recommend you memorize this list. Remember, `boolean`, `long`, `float`, `double`, and each of their associated wrapper classes are not supported by `switch` statements.

Switch Control Flow

Let's look at a simple `switch` 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 the following:

```
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 more detail when we discuss branching, but for now all you need to know is that they terminate the `switch` statement and return flow control to the enclosing statement. As you'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` statement 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:

```
var 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. Notice that we used a `var` for the `switch` variable, which is allowed because it resolves to an `int` by the compiler. Next, 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 `switch` statement:

```
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 as follows:

```
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 branched to only 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` statement 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.

Acceptable Case Values

We conclude our discussion of `switch` statements by talking about acceptable values for `case` statements, given a particular `switch` variable. Not just any variable or value can be used in a `case` statement!

First off, the values in each `case` statement must be compile-time constant values of the same data type as the `switch` value. This means you can use only literals, `enum` constants, or `final` constant variables of the same data type. 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. For example, you can't have a `case` statement value that requires executing a method at runtime, even

if that method always returns the same value. For these reasons, only the first and last `case` statements in the following example compiles:

```
final int getCookies() { return 4; }
void feedAnimals() {
   final int bananas = 1;
   int apples = 2;
   int numberOfAnimals = 3;
   final int cookies = getCookies();
   switch (numberOfAnimals) {
      case bananas:
      case apples:  // DOES NOT COMPILES
      case getCookies():  // DOES NOT COMPILE
      case cookies :  // DOES NOT COMPILE
      case 3 * 5 :
   }
}
```

The `bananas` variable is marked `final`, and its value is known at compile-time, so it is valid. The `apples` variable is not marked `final`, even though its value is known, so it is not permitted. The next two `case` statements, with values `getCookies()` and `cookies`, do not compile because methods are not evaluated until runtime, so they cannot be used as the value of a `case` statement, even if one of the values is stored in a `final` variable. The last `case` statement, with value `3 * 5`, does compile, as expressions are allowed as `case` values, provided the value can be resolved at compile-time. They also must be able to fit in the `switch` data type without an explicit cast. We'll go into that in more detail shortly.

Next, the data type for `case` statements must all match the data type of the `switch` variable. For example, you can't have a `case` statement of type `String`, if the `switch` statement variable is of type `int`, since the types are incomparable.

A More Complex Example

We now present a large `switch` statement, not unlike what you could see on the exam, with numerous broken `case` statements. See if you can figure out why certain `case` statements compile and others do 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, `"Test"`, compiles without issue since it is 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 constant value, despite having a known value at this particular line of ex-

ecution. If a `final` modifier was added to the declaration of `middleName`, this case statement would have compiled. 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 do not compile because none of them has a matching type of `String`, the last one being an `enum` value.

Numeric Promotion and Casting

Last but not least, `switch` statements support numeric promotion that does not require an explicit cast. For example, see if you can understand why only two of these `case` statements compile:

```
short size = 4;
final int small = 15;
final int big = 1_000_000;
switch(size) {
   case small:
   case 1+2 :
   case big:  // DOES NOT COMPILE
}
```

As you may recall from our discussion of numeric promotion and casting in Chapter 3, the compiler can easily cast `small` from `int` to `short` at compile-time because the value `15` is small enough to fit inside a `short`. This would not be permitted if `small` was not a compile-time constant. Likewise, it can convert the expression `1+2` from `int` to `short` at compile-time. On the other hand, `1_000_000` is too large to fit inside of `short` without an explicit cast, so the last `case` statement does not compile.

Writing *while* Loops

A common practice when writing software is the need to do the same task some number of times. You could use the decision structures we have presented so far to accomplish this, but that's going to be a pretty long chain of `if` or `else` statements, especially if you have to execute the same thing 100 times or more.

Enter loops! A *loop* is a repetitive control structure that can execute a statement of code multiple times in succession. By making use of variables being able to be assigned new values, each repetition of the statement may be different. In the following example, the loop increments a `counter` variable that causes the value of `price` to increase by `10` on each execution of the loop. The loop continues for a total of 10 times.

```
int counter = 0;
while (counter < 10) {
   double price = counter * 10;
   System.out.println(price);
   counter++;
}
```

If you don't follow this code, don't panic—we'll be covering it shortly. In this section, we're going to discuss the `while` loop and its two forms. In the next section, we'll move onto `for` loops, which have their roots in `while` loops.

The *while* Statement

The simplest repetitive control structure in Java is the `while` statement, described in Figure 4.4. 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 4.4 The structure of a `while` statement

As shown in Figure 4.4, a `while` loop is similar to an `if` statement in that it is composed of a `boolean` expression and a statement, or a block of statements. During execution, the `boolean` expression is evaluated before each iteration of the loop and exits if the evaluation returns `false`.

Let's return to our mouse example from Chapter 2 and show how 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.

One thing to remember is that a `while` loop may terminate after its first evaluation of the `boolean` expression. For example, how many times is `Not full!` printed in the following example?

```
int full = 5;
while(full < 5) {
   System.out.println("Not full!");
   full++;
}
```

The answer? Zero! On the first iteration of the loop, the condition is reached, and the loop exits. This is why `while` loops are often used in places where you expect zero or more executions of the loop. Simply put, the body of the loop may not execute at all or may execute many times.

The *do/while* Statement

The second form a `while` loop can take is called a *do/while* loop, which like a `while` loop is a repetition control structure with a termination condition and statement, or a block of statements, as shown in Figure 4.5. Unlike a `while` loop, though, a `do`/`while` loop guarantees that the statement or block will be executed at least once. Whereas a `while` loop is executed zero or more times, a `do`/`while` loop is executed one or more times.

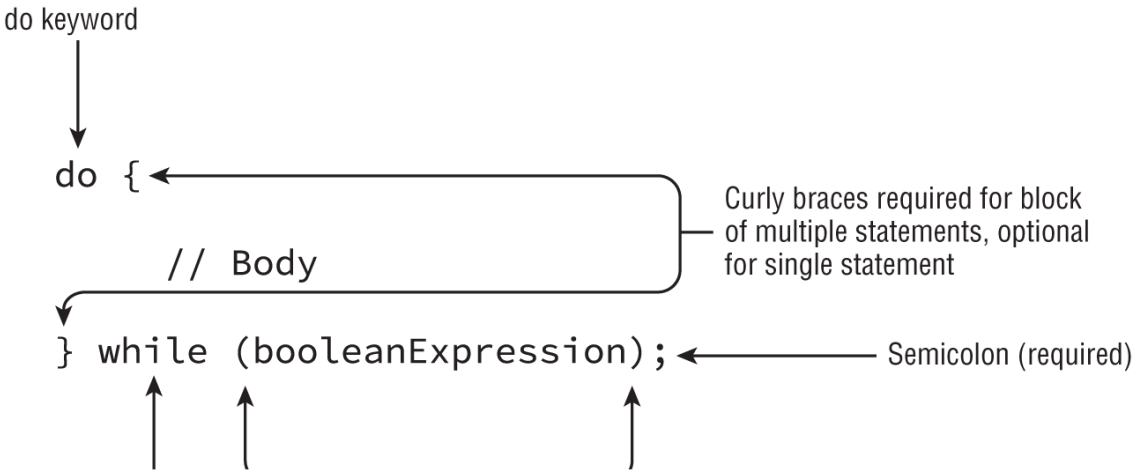

FIGURE 4.5 The structure of a `do`/`while` statement

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 body before the conditional expression so that the body will be executed at least once. For example, what is the output of the following statements?

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

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

Comparing *while* and *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(llama > 10) {
   System.out.println("Llama!");
   llama--;
}
```

and this `do`/`while` loop:

```
if(llama > 10) {
   do {
      System.out.println("Llama!");
      llama--;
```

```
    } while(llama > 10);
}
```

Although one of the loops is certainly easier to read, they are functionally equivalent. Think about it. If `llama` is less than or equal to `10` at the start, then both code snippets will exit without printing anything. If `llama` is greater than `10`, say `15`, then both loops will print `Llama!` exactly five times.

We recommend you use a `while` loop when the code will execute zero or more times and a `do`/`while` loop when the code will execute one or more times. To put it another way, you should use a `do`/`while` loop when you want your loop to execute at least once.

That said, determining whether you should use a `while` loop or a `do`/`while` loop in practice is sometimes about personal preference and about code readability.

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

```
if(llama > 10) {
    do {
        System.out.println("Llama!");
        llama--;
    } while(llama > 10);
} else {
    llama++;
}
```

For fun, try taking a `do`/`while` loop you've written in the past and convert it to a `while` loop, or vice versa.

Infinite Loops

The single most important thing you should be aware of when you are using any repetition control structure is to make sure they always terminate! Failure to terminate a loop can lead to numerous problems in practice including overflow exceptions, memory leaks, slow performance, and even bad data. Let's take a look at an example:

```
int pen = 2;
int pigs = 5;
while(pen < 10)
   pigs++;
```

You may notice one glaring problem with this statement: it will never end. The variable `pen` is never modified, so the expression `(pen < 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. An *infinite loop* is a loop whose termination condition is never reached during runtime.

Anytime you write a loop, you should examine it to determine whether the termination condition is always eventually met under some condition. For example, a loop in which no variables are changing between two executions suggests that the termination condition may not be met. The loop variables should always be moving in a particular direction.

In other words, make sure the loop condition, or the variables the condition is dependent on, are changing between executions. Then, ensure that the termination condition will be eventually reached in all circumstances. As you'll see in the last section of this chapter, a loop may also exit under other conditions, such as a `break` statement.

Real World Scenario

A PRACTICAL USE OF AN INFINITE LOOP

In practice, infinite loops can be used to monitor processes that exist for the life of the program—for example, a process that wakes up every 30 seconds to look for work to be done and then goes back to sleep afterward.

When creating an infinite loop like this, you need to make sure there are only a fixed number of them created by the application, or you could run out of memory. You also have to make sure that there is a way to stop them, often as part of the application shutting down. Finally, there are modern alternatives to creating infinite loops, such as using a scheduled thread executor, that are well beyond the scope of the exam.

If you're not familiar with how to create and execute multiple processes at once, don't worry, you don't need to be for this exam. When you continue on to exam 1Z0-816, you will study these topics as part of concurrency.

Constructing *for* Loops

Even though `while` and `do`/`while` statements are quite powerful, some tasks are so common in writing software that special types of loops were created—for example, iterating over a statement exactly 10 times or iterating over a list of names. You could easily accomplish these tasks with various `while` loops that you've seen so far, but they usually require writing multiple lines of code and managing variables manually. Wouldn't it be great if there was a looping structure that could do the same thing in a single line of code?

With that, we present the most convenient repetition control structure, `for` loops. There are two types of `for` loops, although both use the same

`for` keyword. 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 them as the `for` loop and the for-each loop, respectively, throughout the book.

The *for* Loop

A basic *for loop* has the same conditional `boolean` expression and statement, or block of statements, as the `while` loops, as well as two new sections: an *initialization block* and an *update* statement. Figure 4.6 shows how these components are laid out.

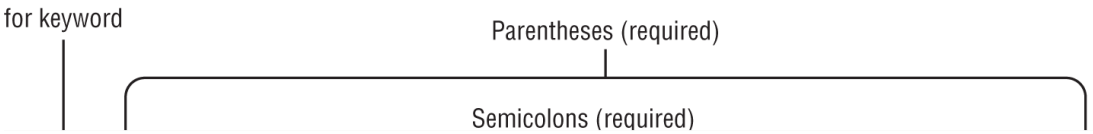

FIGURE 4.6 The structure of a basic `for` loop

Although Figure 4.6 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 single line that otherwise would take multiple lines with a `while` loop. Note that each section is separated by a semicolon. Also, 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 accessible only within the `for` loop. Be wary of any exam questions in which a variable is declared within the initialization block of

a `for` loop and then read outside the loop. For example, this code does not compile because the loop variable `i` is referenced outside the loop:

```
for(int i=0; i < 10; i++)
   System.out.print("Value is: "+i);
System.out.println(i);  // DOES NOT COMPILE
```

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 creation of the `for` loop.

Let's take a look at an example that prints the first five numbers, starting with zero:

```
for(int i = 0; i < 5; 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 the following:

```
0 1 2 3 4
```

On the fifth iteration of the loop, the value of `i` reaches `4` and is incremented by `1` to reach `5`. On the sixth iteration of the loop, the `boolean` expression is evaluated, and since `(5 < 5)` returns `false`, the loop terminates without executing the statement loop body.

Real World Scenario

WHY *I* IN *FOR* LOOPS?

You may notice it is common practice to name a `for` loop variable `i`. Long before Java existed, programmers started using `i` as short for increment variable, and the practice exists today, even though many of those programming languages no longer do!

For double or triple loops, where `i` is already used, the next letters in the alphabet, `j` and `k`, are often used, respectively. One advantage of using a single-letter variable name in a `for` loop is that it doesn't take up a lot of space, allowing the `for` loop declaration to fit on a single line.

For the exam, and for your own coding experience, you should know that using a single-letter variable name is not required. That said, you are likely to encounter `i` in `for` loops throughout your professional software development experience.

Printing Elements in Reverse

Let's say you wanted to print the same first five numbers from zero as we did in the previous section, but this time in reverse order. The goal then is to print `4 3 2 1 0`.

How would you do that? Starting with Java 10, you may now see `var` used in a `for` loop, so let's use that for this example. An initial implementation might look like the following:

```
for (var counter = 5; counter > 0; counter--) {
   System.out.print(counter + " ");
}
```

First, how is `var` interpreted? Since it is assigned a value of `5`, the compiler treats it as having a type of `int`. Next, what does the example output? While this snippet does output five distinct values and it resembles our first `for` loop example, it does not output the same five values. Instead, this is the output:

```
5 4 3 2 1
```

Wait, that's not what we wanted! We wanted `4 3 2 1 0`. It starts with `5`, because that is the first value assigned to it. Let's fix that by starting with `4` instead:

```
for (var counter = 4; counter > 0; counter--) {
   System.out.print(counter + " ");
}
```

What does this print now? This prints the following:

```
4 3 2 1
```

So close! The problem is it ends with `1`, not `0`, because we told it to exit as soon as the value was not strictly greater than `0`. If we want to print the same `0` through `4` as our first example, we need to update the termination condition, like this:

```
for (var counter = 4; counter >= 0; counter--) {
   System.out.print(counter + " ");
}
```

Finally! We have code that now prints `4 3 2 1 0` and matches the reverse of our `for` loop example in the previous section. We could have instead used `counter > -1` as the loop termination condition in this example, although `counter >= 0` tends to be more readable.

For the exam, you are going to have to know how to read forward and backward `for` loops. When you see a `for` loop on the exam, pay close attention to the loop variable and operations if the decrement operator, `--`, is used. While incrementing from `0` in a `for` loop is often straightforward, decrementing tends to be less intuitive. In fact, if you do see a `for` loop with a decrement operator on the exam, you should assume they are trying to test your knowledge of loop operations.

Working with *for* Loops

Although most `for` loops you are likely to encounter in your professional development experience will be well defined and similar to the previous examples, 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 does not compile, 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( )` without any semicolons 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 or 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 5
```

3. Redeclaring a Variable in the Initialization Block

```
int x = 0;
for(int x = 4; x < 5; x++) {   // 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 removing the declaration of `x` from the `for` loop as follows:

```
int x = 0;
for(x = 0; x < 5; x++) {
```

```
   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

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

Like the third example, this code 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 multiple terms 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
```

We covered this already at the start of this section, but this is so important for passing the exam that we discuss it again here. 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 cause a compiler error.

Modifying Loop Variables

What happens if you modify a variable in a `for` loop, or any other loop for that matter? Does Java even let you modify these variables? Take a look at the following three examples, and see whether you can determine what will happen if they are each run independently:

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

for(int j=1; j<10; j++)
   j--;

for(int k=0; k<10; )
   k++;
```

All three of these examples compile, as Java does let you modify loop variables, whether they be in `for`, `while`, or `do`/`while` loops. The first two examples create infinite loops, as loop conditions `i<10` and `j<10` are never reached, independently. In the first example, `i` is reset during every loop to `0`, then incremented to `1`, then reset to `0`, and so on. In the second example, `j` is decremented to `0`, then incremented to `1`, then decremented to `0`, and so on. The last example executes the loop exactly 10 times, so it is valid, albeit a little unusual.

Java does allow modification of loop variables, but you should be wary if you see questions on the exam that do this. While it is normally straightforward to look at a `for` loop and get an idea of how many times the loop will execute, once we start modifying loop variables, the behavior can be extremely erratic. This is especially true when nested loops are involved, which we cover later in this chapter.

There are also some special considerations when modifying a `Collection` object within a loop. For example, if you delete an element from a `List` while iterating over it, you could run into a `ConcurrentModificationException`. This topic is out of scope for the exam, though. You'll revisit this when studying for the 1Z0-816 exam.

As a general rule, it is considered a poor coding practice to modify loop variables due to the unpredictability of the result. It also tends to make code difficult for other people to read. If you need to exit a loop early or change the flow, you can use `break`, `continue`, or `return`, which we'll discuss later in this chapter.

The for-each Loop

Let's say you want to iterate over a set of values, such as a list of names, and print each of them. Using a `for` loop, this can be accomplished with a `counter` variable:

```
public void printNames(String[] names) {
   for(int counter=0; counter<names.length; counter++)
      System.out.println(names[counter]);
}
```

This works, although it's a bit verbose. We're creating a `counter` variable, but we really don't care about its value—just that it loops through the array in order.

After almost 20 years of programming `for` loops like this, the writers of Java took a page from some other programming languages and added the enhanced `for` loop, or for-each loop as we like to call it. The *for-each* loop is a specialized structure designed to iterate over arrays and various Collection Framework classes, as presented in Figure 4.7.

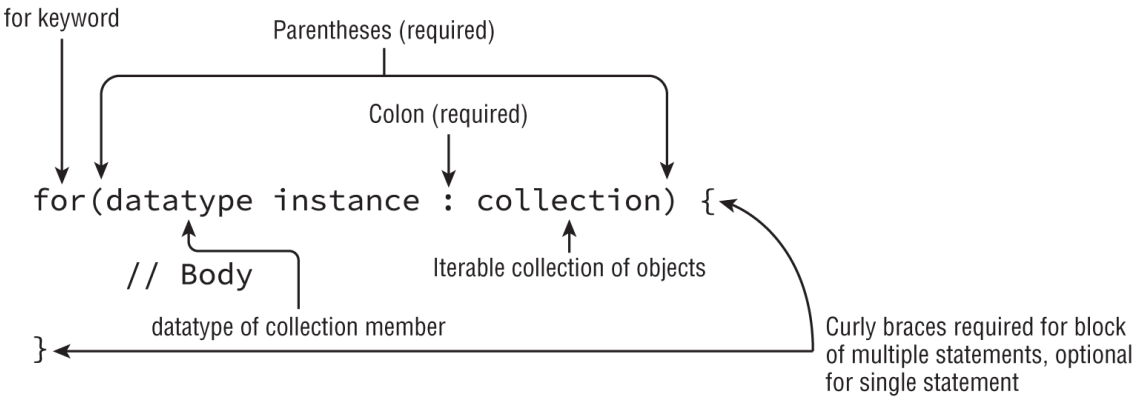

FIGURE 4.7 The structure of an enhanced for-each loop

The for-each loop declaration is composed of an initialization section and an object to be iterated over. The right side of the for-each loop must be one of the following:

- A built-in Java array
- An object whose type implements `java.lang.Iterable`

We'll cover what implements means in Chapter 9, "Advanced Class Design," but for now you just need to know the right side must be an array or collection of items, such as a `List` or a `Set`. For the exam, you should know that this does not include all of the Collections Framework classes or interfaces, but only those that implement or extend that `Collection` interface. For example, `Map` is not supported in a for-each loop, although `Map` does include methods that return `Collection` instances.

In Chapter 5, we'll go into detail about how to create `List` objects and how they differ from traditional Java arrays. Likewise, `String` and `StringBuilder`, which you will also see in the next chapter, do not implement `Iterable` and cannot be used as the right side of a for-each statement.

The left side of the for-each loop must include a declaration for an instance of a variable whose type is compatible with the type of the array or collection on the right side of the statement. A `var` may also be used for the variable type declaration, with the specific type determined by the

right side of the for-each statement. On each iteration of the loop, the named variable on the left side of the statement is assigned a new value from the array or collection on the right side of the statement.

Let's return to our previous example and see how we can apply a for-each loop to it.

```
public void printNames(String[] names) {
   for(String name : names)
      System.out.println(name);
}
```

A lot shorter, isn't it? We no longer have a `counter` loop variable that we need to create, increment, and monitor. Like using a `for` loop in place of a `while` loop, for-each loops are meant to make code easier to read/write, freeing you to focus on the parts of your code that really matter.

Tackling the for-each Statement

Let's work with 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 is a simple one, with no tricks. The code will compile and print the following:

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

- What will this code output?

```
List<String> values = new ArrayList<String>();
values.add("Lisa");
values.add("Kevin");
values.add("Roger");

for(var value : values) {
   System.out.print(value + ", ");
}
```

- This code will compile and print the same values:

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

- Like the regular `for` loop, the for-each loop also accepts `var` for the loop variable, with the type implied by the data type being iterated over.
- When you see a for-each loop on the exam, make sure the right side is an array or `Iterable` object and the left side has a matching type.
- Why does this 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 define a list of items, so the compiler will throw an exception since it does not know how to iterate over the `String` . As a developer, you could iterate over each character of a `String` , but this would require using the

`charAt()` method, which is not compatible with a for-each loop. The `charAt()` method, along with other `String` methods, will be covered in Chapter 5.

- Why does this 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 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.

Switching Between *for* and for-each Loops

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:

```
List<String> names = new 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
```

This is not as short as our for-each example, but it does create the output we wanted, without the extra comma.

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:

```
int[] values = new int[3];
values[0] = 1;
values[1] = Integer.valueOf(3);
values[2] = 6;

for(int i=1; i<values.length; i++) {
   System.out.print((values[i]-values[i-1]) + ", ");
}
```

This sample code would output the following:

```
2, 3,
```

Notice that we skip the first index of the array, since `value[-1]` is not defined and would throw an `IndexOutOfBoundsException` error if called with `i=0`. When comparing `n` elements of a list with each other, our loop should be executed `n-1` times.

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

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 `Iterable`, there is a different, but similar, conversion. For example, assuming `values` is an instance of `List<Integer>`, the following two loops are equivalent:

```
for(int value : values) {
   System.out.print(value + ", ");
}
for(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 in the `for` loop as `next()` both retrieves the next available value and

moves the iterator forward.

Controlling Flow with Branching

The final type of control flow structures we will cover in this chapter are branching statements. Up to now, we have been dealing with single loops that ended only 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 the previous examples.

Nested Loops

Before we move into branching statements, we need to introduce the concept of nested loops. A *nested loop* is a loop that contains another loop including `while`, `do`/`while`, `for`, and for-each loops. For example, consider the following code that iterates over a two-dimensional array, which is an array that contains other arrays as its members. We'll cover multidimensional arrays in detail in Chapter 5, 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 loop 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 whether you can determine what this code will output:

```
int hungryHippopotamus = 8;
while(hungryHippopotamus>0) {
   do {
      hungryHippopotamus -= 2;
   } while (hungryHippopotamus>5);
   hungryHippopotamus--;
   System.out.print(hungryHippopotamus+", ");
}
```

The first time this loop executes, the inner loop repeats until the value of `hungryHippopotamus` 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 `hungryHippopotamus` 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,
```

The examples in the rest of this section will include many nested loops. You will also encounter nested loops on the exam, so the more practice you have with them, the more prepared you will be.

Some of the most time-consuming questions you may see on the exam could involve nested loops with lots of branching. We recommend you try to answer the question right away, but if you start to think it is going to take too long, you should mark it and come back to it later. Remember, all questions on the exam are weighted evenly!

Adding Optional Labels

One thing we intentionally skipped when we presented `if` 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 identifier 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();
}
```

Labels follow the same rules for formatting as identifiers. For readability, they are commonly expressed using uppercase letters, with underscores between words, to distinguish them from regular variables. When dealing with only one loop, labels do not add any value, but as we'll see in the next section, they are extremely useful in nested structures.

While this topic is not on the exam, it is possible to add optional labels to control and block statements. For example, the following is permitted by the compiler, albeit extremely uncommon:

```
int frog = 15;
BAD_IDEA: if(frog>10)
EVEN_WORSE_IDEA: {
   frog++;
}
```

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 a `break` statement that appears inside of a `while`, `do`/`while`, or `for` loop, as it will end the loop early, as shown in Figure 4.8.

FIGURE 4.8 The structure of a `break` statement

Notice in Figure 4.8 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 FindInMatrix {
   public static void main(String[] args) {
      int[][] list = {{1,13},{5,2},{2,2}};
```

```
        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 the following:

```
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: (2,1)
```

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 extend 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 4.9.

FIGURE 4.9 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 identical 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 an example. Imagine we have a zookeeper who is supposed to clean the first leopard in each of four stables but skip stable `b` entirely.

```
1: public class CleaningSchedule {
2:    public static void main(String[] args) {
3:       CLEANING: for(char stables = 'a'; stables<='d'; stables++) {
4:          for(int leopard = 1; leopard<4; leopard++) {
5:             if(stables=='b' || leopard==2) {
6:                continue CLEANING;
7:             }
8:             System.out.println("Cleaning: "+stables+","+leopard);
9: } } } }
```

With the structure as defined, the loop will return control to the parent loop any time the first value is `b` or the second value is `2`. On the first, third, and fourth executions of the outer loop, the inner loop prints a statement exactly once and then exits on the next inner loop when `leopard` is `2`. On the second execution of the outer loop, the inner loop immediately exits without printing anything since `b` is encountered right away. The following is printed:

```
Cleaning: a,1
Cleaning: c,1
Cleaning: d,1
```

Now, imagine we removed the `CLEANING` label in the `continue` statement so that control is returned to the inner loop instead of the outer. Line 6 becomes the following:

```
6:                  continue;
```

This corresponds to the zookeeper skipping all leopards except those labeled `2` or in stable `b`. The output would then be the following:

```
Cleaning: a,1
Cleaning: a,3
Cleaning: c,1
Cleaning: c,3
Cleaning: d,1
Cleaning: d,3
```

Finally, if we remove the `continue` statement and associated `if` statement altogether by removing lines 5–7, we arrive at a structure that outputs all the values, such as this:

```
Cleaning: a,1
Cleaning: a,2
Cleaning: a,3
Cleaning: b,1
Cleaning: b,2
Cleaning: b,3
Cleaning: c,1
Cleaning: c,2
Cleaning: c,3
Cleaning: d,1
Cleaning: d,2
Cleaning: d,3
```

The *return* Statement

Given that this book shouldn't be your first foray into programming, we hope you've come across methods that contain `return` statements. Regardless, we'll be covering how to design and create methods that use them in detail in Chapter 7, "Methods and Encapsulation."

For now, though, you should be familiar with the idea that creating methods and using `return` statements can be used as an alternative to using labels and `break` statements. For example, take a look at this rewrite of our earlier `FindInMatrix` class:

```
public class FindInMatrixUsingReturn {
   private static int[] searchForValue(int[][] list, int v) {
      for (int i = 0; i < list.length; i++) {
         for (int j = 0; j < list[i].length; j++) {
            if (list[i][j] == v) {
               return new int[] {i,j};
            }
         }
      }
      return null;
   }

   public static void main(String[] args) {
      int[][] list = { { 1, 13 }, { 5, 2 }, { 2, 2 } };
      int searchValue = 2;
      int[] results = searchForValue(list,searchValue);

      if (results == null) {
         System.out.println("Value " + searchValue + " not found");
      } else {
         System.out.println("Value " + searchValue + " found at: " +
            "(" + results[0] + "," + results[1] + ")");
      }
   }
}
```

This class is functionally the same as the first FindInMatrix class we saw earlier using `break` . If you need finer-grained control of the loop with

multiple `break` and `continue` statements, the first class is probably better. That said, we find code without labels and `break` statements a lot easier to read and debug. Also, making the search logic an independent function makes the code more reusable and the calling `main()` method a lot easier to read.

For the exam, you will need to know both forms. Just remember that `return` statements can be used to exit loops quickly and can lead to more readable code in practice, especially when used with nested loops.

Unreachable Code

One facet of `break`, `continue`, and `return` that you should be aware of is that any code placed immediately after them in the same block is considered unreachable and will not compile. For example, the following code snippet does not compile:

```
int checkDate = 0;
while(checkDate<10) {
   checkDate++;
   if(checkDate>100) {
      break;
      checkDate++;  // DOES NOT COMPILE
   }
}
```

Even though it is not logically possible for the `if` statement to evaluate to `true` in this code sample, the compiler notices that you have statements immediately following the `break` and will fail to compile with "unreachable code" as the reason. The same is true for `continue` and `return` statements too, as shown in the following two examples:

```
int minute = 1;
WATCH: while(minute>2) {
   if(minute++>2) {
      continue WATCH;
```

```
      System.out.print(minute);  // DOES NOT COMPILE
   }
}

int hour = 2;
switch(hour) {
   case 1: return; hour++;  // DOES NOT COMPILE
   case 2:
}
```

One thing to remember is that it does not matter if loop or decision structure actually visits the line of code. For example, the loop could execute zero or infinite times at runtime. Regardless of execution, the compiler will report an error if it finds any code it deems unreachable, in this case any statements immediately following a `break`, `continue`, or `return` statement.

Reviewing Branching

We conclude this section with Table 4.1, which 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 4.1 Advanced flow control usage

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

Last but not least, all testing centers should offer some form of scrap paper or dry-erase board to use during the exam. We strongly recommend you make use of these testing aids should you encounter complex questions involving nested loops and branching statements.

Summary

This chapter presented how to make intelligent decisions in Java. We covered basic decision-making constructs such as `if`, `else`, and `switch` statements and showed how to use them to change the path of process at runtime. Remember that the `switch` statement allows a lot of data types it did not in the past, such as `String`, `enum`, and in certain cases `var`.

We then moved our discussion to repetition control structures, starting with `while` and `do`/`while` loops. We showed how to use them to create processes that looped multiple times and also showed how it is important to make sure they eventually terminate. Remember that most of these structures require the evaluation of a particular `boolean` expression to complete.

Next, we covered the extremely convenient repetition control structures, `for` and for-each loops. While their syntax is more complex than the traditional `while` or `do`/`while` loops, they are extremely useful in everyday coding and allow you to create complex expressions in a single line of code. With a for-each loop you don't need to explicitly write a `boolean` expression, since the compiler builds one for you. For clarity, we referred to an enhanced `for` loop as a for-each loop, but syntactically both are written using the `for` keyword.

We concluded this chapter by discussing advanced control options and how flow can be enhanced through nested loops, coupled with `break`, `continue`, and `return` statements. Be wary of questions on the exam that use nested loops, 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

Understand `if` and `else` decision control statements. The `if` and `else` statements come up frequently throughout the exam in questions unrelated to decision control, so make sure you fully understand these basic building blocks of Java.

Understand `switch` statements and their proper usage. You should be able to spot a poorly formed `switch` statement on the exam. The `switch` value and data type should be compatible with the `case` statements, and the values for the `case` statements must evaluate to compile-time constants. Finally, at runtime a `switch` statement branches to the first

matching `case`, or `default` if there is no match, or exits entirely if there is no match and no `default` branch. The process then continues into any proceeding `case` or `default` statements until a `break` or `return` statement is reached.

Understand `while` loops. Know the syntactical structure of all `while` and `do`/`while` loops. In particular, know when to use one versus the other.

Be able to use `for` loops. You should be familiar with `for` and for-each loops and know how to write and evaluate them. Each loop has its own special properties and structures. You should know how to use for-each loops to iterate over lists and arrays.

Understand how `break`, `continue`, and `return` can change flow control. Know how to change the flow control within a statement by applying a `break`, `continue`, or `return` statement. Also know which control statements can accept `break` statements and which can accept `continue` statements. Finally, you should understand how these statements work inside embedded loops or `switch` statements.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. Which of the following data types can be used in a `switch` statement? (Choose all that apply.)
    1. `enum`
    2. `int`
    3. `Byte`
    4. `long`
    5. `String`
    6. `char`
    7. `var`

8. `double`

2. What is the output of the following code snippet? (Choose all that apply.)

```
3: int temperature = 4;
4: long humidity = -temperature + temperature * 3;
5: if (temperature>=4)
6: if (humidity < 6) System.out.println("Too Low");
7: else System.out.println("Just Right");
8: else System.out.println("Too High");
```

1. `Too Low`
2. `Just Right`
3. `Too High`
4. A `NullPointerException` is thrown at runtime.
5. The code will not compile because of line 7.
6. The code will not compile because of line 8.

3. What is the output of the following code snippet?

```
List<Integer> myFavoriteNumbers = new ArrayList<>();
myFavoriteNumbers.add(10);
myFavoriteNumbers.add(14);
for (var a : myFavoriteNumbers) {
   System.out.print(a + ", ");
   break;
}

for (int b : myFavoriteNumbers) {
   continue;
   System.out.print(b + ", ");
}

for (Object c : myFavoriteNumbers)
   System.out.print(c + ", ");
```

1. It compiles and runs without issue but does not produce any output.
2. `10, 14,`
3. `10, 10, 14,`
4. `10, 10, 14, 10, 14,`
5. Exactly one line of code does not compile.
6. Exactly two lines of code do not compile.
7. Three or more lines of code do not compile.
8. The code contains an infinite loop and does not terminate.

4. Which statements about decision structures are true? (Choose all that apply.)
   1. A for-each loop can be executed on any Collections Framework object.
   2. The body of a `while` loop is guaranteed to be executed at least once.
   3. The conditional expression of a `for` loop is evaluated before the first execution of the loop body.
   4. A `switch` statement with no matching `case` statement requires a `default` statement.
   5. The body of a `do`/`while` loop is guaranteed to be executed at least once.
   6. An `if` statement can have multiple corresponding `else` statements.
5. Assuming `weather` is a well-formed nonempty array, which code snippet, when inserted independently into the blank in the following code, prints all of the elements of `weather`? (Choose all that apply.)

```
private void print(int[] weather) {
   for(____________________) {
      System.out.println(weather[i]);
   }
}
```

   1. `int i=weather.length; i>0; i--`
   2. `int i=0; i<=weather.length-1; ++i`

3. `var w : weather`
4. `int i=weather.length-1; i>=0; i--`
5. `int i=0, int j=3; i<weather.length; ++i`
6. `int i=0; ++i<10 && i<weather.length;`
7. None of the above

6. Which statements, when inserted independently into the following blank, will cause the code to print `2` at runtime? (Choose all that apply.)

```
int count = 0;
BUNNY: for(int row = 1; row <=3; row++)
   RABBIT: for(int col = 0; col <3 ; col++) {
      if((col + row) % 2 == 0)
        _____________________;
      count++;
   }
System.out.println(count);
```

1. `break BUNNY`
2. `break RABBIT`
3. `continue BUNNY`
4. `continue RABBIT`
5. `break`
6. `continue`
7. None of the above, as the code contains a compiler error

7. Given the following method, how many lines contain compilation errors? (Choose all that apply.)

```
private DayOfWeek getWeekDay(int day, final int thursday) {
   int otherDay = day;
   int Sunday = 0;
   switch(otherDay) {
      default:
      case 1: continue;
      case thursday: return DayOfWeek.THURSDAY;
```

```
            case 2: break;
            case Sunday: return DayOfWeek.SUNDAY;
            case DayOfWeek.MONDAY: return DayOfWeek.MONDAY;
         }
         return DayOfWeek.FRIDAY;
      }
```

1. None, the code compiles without issue.
2. 1
3. 2
4. 3
5. 4
6. 5
7. 6
8. The code compiles but may produce an error at runtime.

8. What is the result of the following code snippet?

```
3: int sing = 8, squawk = 2, notes = 0;
4: while(sing > squawk) {
5:    sing--;
6:    squawk += 2;
7:    notes += sing + squawk;
8: }
9: System.out.println(notes);
```

1. 11
2. 13
3. 23
4. 33
5. 50
6. The code will not compile because of line 7.

9. What is the output of the following code snippet?

```
2: boolean keepGoing = true;
3: int result = 15, meters = 10;
4: do {
```

```
5:    meters--;
6:    if(meters==8) keepGoing = false;
7:    result -= 2;
8: } while keepGoing;
9: System.out.println(result);
```

1. 7
2. 9
3. 10
4. 11
5. 15
6. The code will not compile because of line 6.
7. The code does not compile for a different reason.

10. Which statements about the following code snippet are correct? (Choose all that apply.)

```
for(var penguin : new int[2])
   System.out.println(penguin);

var ostrich = new Character[3];
for(var emu : ostrich)
   System.out.println(emu);

List parrots = new ArrayList();
for(var macaw  : parrots)
   System.out.println(macaw);
```

1. The data type of `penguin` is `Integer`.
2. The data type of `penguin` is `int`.
3. The data type of `emu` is undefined.
4. The data type of `emu` is `Character`.
5. The data type of `macaw` is undefined.
6. The data type of `macaw` is `Object`.
7. None of the above, as the code does not compile

11. What is the result of the following code snippet?

```
final char a = 'A', e = 'E';
char grade = 'B';
switch (grade) {
   default:
   case a:
   case 'B': 'C': System.out.print("great ");
   case 'D': System.out.print("good "); break;
   case e:
   case 'F': System.out.print("not good ");
}
```

1. `great`
2. `great good`
3. `good`
4. `not good`
5. The code does not compile because the data type of one or more `case` statements does not match the data type of the `switch` variable.
6. None of the above

12. Given the following array, which code snippets print the elements in reverse order from how they are declared? (Choose all that apply.)

```
char[] wolf = {'W', 'e', 'b', 'b', 'y'};
```

1.

```
int q = wolf.length;
for( ; ; ) {
   System.out.print(wolf[--q]);
   if(q==0) break;
}
```

2.

```
for(int m=wolf.length-1; m>=0; --m)
   System.out.print(wolf[m]);
```

3.

```
for(int z=0; z<wolf.length; z++)
   System.out.print(wolf[wolf.length-z]);
```

4.

```
int x = wolf.length-1;
for(int j=0; x>=0 && j==0; x--)
   System.out.print(wolf[x]);
```

5.

```
final int r = wolf.length;
for(int w = r-1; r>-1; w = r-1)
   System.out.print(wolf[w]);
```

6.

```
for(int i=wolf.length; i>0; --i)
   System.out.print(wolf[i]);
```

7. None of the above

13. What distinct numbers are printed when the following method is executed? (Choose all that apply.)

```
private void countAttendees() {
   int participants = 4, animals = 2, performers = -1;

   while((participants = participants+1) < 10) {}
   do {} while (animals++ <= 1);
```

```
        for( ; performers<2; performers+=2) {}

        System.out.println(participants);
        System.out.println(animals);
        System.out.println(performers);
    }
```

1. `6`
2. `3`
3. `4`
4. `5`
5. `10`
6. `9`
7. The code does not compile.
8. None of the above

14. What is the output of the following code snippet?

```
2: double iguana = 0;
3: do {
4:    int snake = 1;
5:    System.out.print(snake++ + " ");
6:    iguana--;
7: } while (snake <= 5);
8: System.out.println(iguana);
```

1. `1 2 3 4 -4.0`
2. `1 2 3 4 -5.0`
3. `1 2 3 4 5 -4.0`
4. `0 1 2 3 4 5 -5.0`
5. The code does not compile.
6. The code compiles but produces an infinite loop at runtime.
7. None of the above

15. Which statements, when inserted into the following blanks, allow the code to compile and run without entering an infinite loop? (Choose all that apply.)

```
4:  int height = 1;
5:  L1: while(height++ <10) {
6:     long humidity = 12;
7:     L2: do {
8:        if(humidity-- % 12 == 0) _________________;
9:        int temperature = 30;
10:       L3: for( ; ; ) {
11:          temperature++;
12:          if(temperature>50) _________________;
13:       }
14:    } while (humidity > 4);
15: }
```

1. `break L2` on line 8; `continue L2` on line 12
2. `continue` on line 8; `continue` on line 12
3. `break L3` on line 8; `break L1` on line 12
4. `continue L2` on line 8; `continue L3` on line 12
5. `continue L2` on line 8; `continue L2` on line 12
6. None of the above, as the code contains a compiler error.

16. What is the output of the following code snippet? (Choose all that apply.)

```
2: var tailFeathers = 3;
3: final var one = 1;
4: switch (tailFeathers) {
5:    case one: System.out.print(3 + " ");
6:    default: case 3: System.out.print(5 + " ");
7: }
8: while (tailFeathers > 1) {
9:    System.out.print(--tailFeathers + " "); }
```

1. `3`
2. `5 1`
3. `5 2`
4. `3 5 1`
5. `5 2 1`

6. The code will not compile because of lines 3–5.
7. The code will not compile because of line 6.

17. What is the output of the following code snippet?

```
15: int penguin = 50, turtle = 75;
16: boolean older = penguin >= turtle;
17: if (older = true) System.out.println("Success");
18: else System.out.println("Failure");
19: else if(penguin != 50) System.out.println("Other");
```

1. `Success`
2. `Failure`
3. `Other`
4. The code will not compile because of line 17.
5. The code compiles but throws an exception at runtime.
6. None of the above

18. Which of the following are possible data types for `olivia` that would allow the code to compile? (Choose all that apply.)

```
for(var sophia : olivia) {
   System.out.println(sophia);
}
```

1. `Set`
2. `Map`
3. `String`
4. `int[]`
5. `Collection`
6. `StringBuilder`
7. None of the above

19. What is the output of the following code snippet?

```
6:  String instrument = "violin";
7:  final String CELLO = "cello";
8:  String viola = "viola";
```

```
9:  int p = -1;
10: switch(instrument) {
11:    case "bass" : break;
12:    case CELLO : p++;
13:    default: p++;
14:    case "VIOLIN": p++;
15:    case "viola" : ++p; break;
16: }
17: System.out.print(p);
```

1. `-1`
2. `0`
3. `1`
4. `2`
5. `3`
6. The code does not compile.

20. What is the output of the following code snippet? (Choose all that apply.)

```
9:  int w = 0, r = 1;
10: String name = "";
11: while(w < 2) {
12:    name += "A";
13:    do {
14:       name += "B";
15:       if(name.length()>0) name += "C";
16:       else break;
17:    } while (r <=1);
18:    r++; w++; }
19: System.out.println(name);
```

1. `ABC`
2. `ABCABC`
3. `ABCABCABC`
4. Line 15 contains a compilation error.
5. Line 18 contains a compilation error.

6. The code compiles but never terminates at runtime.
7. The code compiles but throws a `NullPointerException` at runtime.