The if/else Statement

The `if` statement is a fundamental control statement that allows Java to make decisions or, more precisely, to execute statements conditionally. The `if` statement has an associated expression and statement. If the expression evaluates to `true`, the interpreter executes the statement. If the expression evaluates to `false`, the interpreter skips the statement.

Java allows the expression to be of the wrapper type `Boolean` instead of the primitive type `boolean`. In this case, the wrapper object is automatically unboxed.

Here is an example `if` statement:

```
if (username == null)          // If username is null,
   username = "John Doe";      // use a default value
```

Although they look extraneous, the parentheses around the expression are a required part of the syntax for the `if` statement. As we already saw, a block of statements enclosed in curly braces is itself a statement, so we can write `if` statements that look like this as well:

```
if ((address == null) || (address.equals(""))) {
   address = "[undefined]";
   System.out.println("WARNING: no address specified.");
}
```

An `if` statement can include an optional `else` keyword that is followed by a second statement. In this form of the statement, the expression is evaluated, and, if it is `true`, the first statement is executed. Otherwise, the second statement is executed. For example:

```
if (username != null)
   System.out.println("Hello " + username);
else {
   username = askQuestion("What is your name?");
   System.out.println("Hello " + username + ". Welcome!");
}
```

When you use nested `if/else` statements, some caution is required to ensure that the `else` clause goes with the appropriate `if` statement. Consider the following lines:

```
if (i == j)
   if (j == k)
     System.out.println("i equals k");
else
   System.out.println("i doesn't equal j");    // WRONG!!
```

In this example, the inner `if` statement forms the single statement allowed by the syntax of the outer `if` statement. Unfortunately, it is not clear (except from the hint given by the indentation) which `if` the `else` goes with. And in this example, the indentation hint is wrong. The rule is that an `else` clause like this is associated with the nearest `if` statement. Properly indented, this code looks like this:

```
if (i == j)
   if (j == k)
     System.out.println("i equals k");
   else
     System.out.println("i doesn't equal j");    // WRONG!!
```

This is legal code, but it is clearly not what the programmer had in mind. When working with nested `if` statements, you should use curly braces to make your code easier to read. Here is a better way to write the code:

```
if (i == j) {
   if (j == k)
     System.out.println("i equals k");
}
else {
   System.out.println("i doesn't equal j");
}
```

The else if clause

The `if/else` statement is useful for testing a condition and choosing between two statements or blocks of code to execute. But what about when you need to choose between several blocks of code? This is typically done with an `else if` clause, which is not really new syntax, but a common idiomatic usage of the standard `if/else` statement. It looks like this:

```
if (n == 1) {
    // Execute code block #1
}
else if (n == 2) {
    // Execute code block #2
}
else if (n == 3) {
    // Execute code block #3
}
else {
    // If all else fails, execute block #4
}
```

There is nothing special about this code. It is just a series of `if` statements, where each `if` is part of the `else` clause of the previous statement. Using the `else if` idiom is preferable to, and more legible than, writing these statements out in their fully nested form:

```
if (n == 1) {
   // Execute code block #1
```

```
}
else {
   if (n == 2) {
     // Execute code block #2
   }
   else {
     if (n == 3) {
       // Execute code block #3
     }
     else {
       // If all else fails, execute block #4
     }
   }
}
```

The switch Statement

An `if` statement causes a branch in the flow of a program's execution. You can use multiple `if` statements, as shown in the previous section, to perform a multiway branch. This is not always the best solution, however, especially when all of the branches depend on the value of a single variable. In this case, it is inefficient to repeatedly check the value of the same variable in multiple `if` statements.

A better solution is to use a `switch` statement, which is inherited from the C programming language. Although the syntax of this statement is not nearly as elegant as other parts of Java, the brute practicality of the construct makes it worthwhile.

A `switch` statement starts with an expression whose type is an `int`, `short`, `char`, `byte` (or their wrapper type), `String`, or an enum (see Chapter 4 for more on enumerated types).

This expression is followed by a block of code in curly braces that contains various entry points that correspond to possible values for the expression. For example, the following `switch` statement is equivalent to the repeated `if` and `else/if` statements shown in the previous section:

```
switch(n) {
   case 1:                          // Start here if n == 1
     // Execute code block #1
     break;                         // Stop here
   case 2:                          // Start here if n == 2
     // Execute code block #2
     break;                         // Stop here
   case 3:                          // Start here if n == 3
     // Execute code block #3
     break;                         // Stop here
   default:                         // If all else fails...
     // Execute code block #4
```

```
        break;                          // Stop here
    }
```

As you can see from the example, the various entry points into a `switch` statement are labeled either with the keyword `case`, followed by an integer value and a colon, or with the special `default` keyword, followed by a colon. When a `switch` statement executes, the interpreter computes the value of the expression in parentheses and then looks for a `case` label that matches that value. If it finds one, the interpreter starts executing the block of code at the first statement following the `case` label. If it does not find a `case` label with a matching value, the interpreter starts execution at the first statement following a special-case `default`: label. Or, if there is no `default`: label, the interpreter skips the body of the `switch` statement altogether.

Note the use of the `break` keyword at the end of each `case` in the previous code. The `break` statement is described later in this chapter, but, in this case, it causes the interpreter to exit the body of the `switch` statement. The `case` clauses in a `switch` statement specify only the starting point of the desired code. The individual cases are not independent blocks of code, and they do not have any implicit ending point. Therefore, you must explicitly specify the end of each case with a `break` or related statement. In the absence of `break` statements, a `switch` statement begins executing code at the first statement after the matching `case` label and continues executing statements until it reaches the end of the block. On rare occasions, it is useful to write code like this that falls through from one `case` label to the next, but 99% of the time you should be careful to end every `case` and `default` section with a statement that causes the `switch` statement to stop executing. Normally you use a `break` statement, but `return` and `throw` also work.

A `switch` statement can have more than one `case` clause labeling the same statement. Consider the `switch` statement in the following method:

```
boolean parseYesOrNoResponse(char response) {
   switch(response) {
     case 'y':
     case 'Y': return true;
     case 'n':
     case 'N': return false;
     default:
       throw new IllegalArgumentException("Response must be Y or N");
   }
}
```

The `switch` statement and its `case` labels have some important restrictions. First, the expression associated with a `switch` statement must have an appropriate type—either `byte`, `char`, `short`, `int` (or their wrappers), or an enum type or a `String`. The floating-point and `boolean` types are not supported, and neither is `long`, even though `long` is an integer type. Second, the value associated with each `case` label must be a constant value or a constant expression the compiler can evaluate. A `case` label cannot contain a runtime expression involving variables or method calls, for example. Third, the `case` label values must be within the range of the data type used

for the `switch` expression. And finally, it is not legal to have two or more `case` labels with the same value or more than one `default` label.

The while Statement

The `while` statement is a basic statement that allows Java to perform repetitive actions—or, to put it another way, it is one of Java's primary *looping constructs*. It has the following syntax:

```
while (expression)
   statement
```

The `while` statement works by first evaluating the *`expression`*, which must result in a `boolean` or `Boolean` value. If the value is `false`, the interpreter skips the *`statement`* associated with the loop and moves to the next statement in the program. If it is `true`, however, the *`statement`* that forms the body of the loop is executed, and the *`expression`* is reevaluated. Again, if the value of *`expression`* is `false`, the interpreter moves on to the next statement in the program; otherwise, it executes the *`statement`* again. This cycle continues while the *`expression`* remains `true` (i.e., until it evaluates to `false`), at which point the `while` statement ends, and the interpreter moves on to the next statement. You can create an infinite loop with the syntax `while(true)`.

Here is an example `while` loop that prints the numbers 0 to 9:

```
int count = 0;
while (count < 10) {
   System.out.println(count);
   count++;
}
```

As you can see, the variable `count` starts off at 0 in this example and is incremented each time the body of the loop runs. Once the loop has executed 10 times, the expression becomes `false` (i.e., `count` is no longer less than 10), the `while` statement finishes, and the Java interpreter can move to the next statement in the program. Most loops have a counter variable like `count`. The variable names `i`, `j`, and `k` are commonly used as loop counters, although you should use more descriptive names if it makes your code easier to understand.

The do Statement

A `do` loop is much like a `while` loop, except that the loop expression is tested at the bottom of the loop rather than at the top. This means that the body of the loop is always executed at least once. The syntax is:

```
do
   statement
while (expression);
```

Notice a couple of differences between the `do` loop and the more ordinary `while` loop. First, the `do` loop requires both the `do` keyword to mark the beginning of the

loop and the `while` keyword to mark the end and introduce the loop condition. Also, unlike the `while` loop, the `do` loop is terminated with a semicolon. This is because the `do` loop ends with the loop condition rather than simply ending with a curly brace that marks the end of the loop body. The following `do` loop prints the same output as the `while` loop just discussed:

```
int count = 0;
do {
   System.out.println(count);
   count++;
} while(count < 10);
```

The `do` loop is much less commonly used than its `while` cousin because, in practice, it is unusual to encounter a situation where you are sure you always want a loop to execute at least once.

The for Statement

The `for` statement provides a looping construct that is often more convenient than the `while` and `do` loops. The `for` statement takes advantage of a common looping pattern. Most loops have a counter, or state variable of some kind, that is initialized before the loop starts, tested to determine whether to execute the loop body, and then incremented or updated somehow at the end of the loop body before the test expression is evaluated again. The initialization, test, and update steps are the three crucial manipulations of a loop variable, and the `for` statement makes these three steps an explicit part of the loop syntax:

```
for(initialize; test; update) {
    statement
}
```

This `for` loop is basically equivalent to the following `while` loop:

```
initialize;
while (test) {
   statement;
   update;
}
```

Placing the *`initialize`*, *`test`*, and *`update`* expressions at the top of a `for` loop makes it especially easy to understand what the loop is doing, and it prevents mistakes such as forgetting to initialize or update the loop variable. The interpreter discards the values of the *`initialize`* and *`update`* expressions, so to be useful, these expressions must have side effects. *`initialize`* is typically an assignment expression while *`update`* is usually an increment, decrement, or some other assignment.

The following `for` loop prints the numbers 0 to 9, just as the previous `while` and `do` loops have done:

```
int count;
for(count = 0 ; count < 10 ; count++)
   System.out.println(count);
```

Notice how this syntax places all the important information about the loop variable on a single line, making it very clear how the loop executes. Placing the update expression in the `for` statement itself also simplifies the body of the loop to a single statement; we don't even need to use curly braces to produce a statement block.

The `for` loop supports some additional syntax that makes it even more convenient to use. Because many loops use their loop variables only within the loop, the `for` loop allows the *`initialize`* expression to be a full variable declaration, so that the variable is scoped to the body of the loop and is not visible outside of it. For example:

```
for(int count = 0 ; count < 10 ; count++)
   System.out.println(count);
```

Furthermore, the `for` loop syntax does not restrict you to writing loops that use only a single variable. Both the *`initialize`* and *`update`* expressions of a `for` loop can use a comma to separate multiple initializations and update expressions. For example:

```
for(int i = 0, j = 10 ; i < 10 ; i++, j--)
     sum += i * j;
```

Even though all the examples so far have counted numbers, `for` loops are not restricted to loops that count numbers. For example, you might use a `for` loop to iterate through the elements of a linked list:

```
for(Node n = listHead; n != null; n = n.nextNode())
   process(n);
```

The *`initialize`*, *`test`*, and *`update`* expressions of a `for` loop are all optional; only the semicolons that separate the expressions are required. If the *`test`* expression is omitted, it is assumed to be `true`. Thus, you can write an infinite loop as `for(;;)`.

The foreach Statement

Java's `for` loop works well for primitive types, but it is needlessly clunky for handling collections of objects. Instead, an alternative syntax known as a *foreach* loop is used for handling collections of objects that need to be looped over.

The foreach loop uses the keyword `for` followed by an opening parenthesis, a variable declaration (without initializer), a colon, an expression, a closing parenthesis, and finally the statement (or block) that forms the body of the loop:

```
for( declaration : expression )
     statement
```

Despite its name, the foreach loop does not have a keyword `foreach`—instead, it is common to read the colon as "in"—as in "foreach name in studentNames."

For the `while`, `do`, and `for` loops, we've shown an example that prints 10 numbers. The foreach loop can do this too, but it needs a collection to iterate over. In order to loop 10 times (to print out 10 numbers), we need an array or other collection with 10 elements. Here's code we can use:

```
// These are the numbers we want to print
int[] primes = new int[] { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
// This is the loop that prints them
for(int n : primes)
     System.out.println(n);
```


What foreach cannot do

Foreach is different from the `while`, `for`, or `do` loops, because it hides the loop counter or `Iterator` from you. This is a very powerful idea, as we'll see when we discuss lambda expressions, but there are some algorithms that cannot be expressed very naturally with a foreach loop.

For example, suppose you want to print the elements of an array as a comma-separated list. To do this, you need to print a comma after every element of the array except the last, or equivalently, before every element of the array except the first. With a traditional `for` loop, the code might look like this:

```
for(int i = 0; i < words.length; i++) {
     if (i > 0) System.out.print(", ");
     System.out.print(words[i]);
}
```

This is a very straightforward task, but you simply cannot do it with foreach. The problem is that the foreach loop doesn't give you a loop counter or any other way to tell if you're on the first iteration, the last iteration, or somewhere in between.

A similar issue exists when using foreach to iterate through the elements of a collection. Just as a foreach loop over an array has no way to obtain the array index of the current element, a foreach loop over a collection has no way to obtain the `Iterator` object that is being used to itemize the elements of the collection.

Here are some other things you cannot do with a foreach style loop:

- Iterate backward through the elements of an array or `List`.
- Use a single loop counter to access the same-numbered elements of two distinct arrays.
- Iterate through the elements of a `List` using calls to its `get()` method rather than calls to its iterator.

The break Statement

A `break` statement causes the Java interpreter to skip immediately to the end of a containing statement. We have already seen the `break` statement used with the `switch` statement. The `break` statement is most often written as simply the keyword `break` followed by a semicolon:

```
break;
```

When used in this form, it causes the Java interpreter to immediately exit the innermost containing `while`, `do`, `for`, or `switch` statement. For example:

```
for(int i = 0; i < data.length; i++) {
    if (data[i] == target) {  // When we find what we're looking for,
        index = i;            // remember where we found it
        break;                // and stop looking!
    }
}   // The Java interpreter goes here after executing break
```

The `break` statement can also be followed by the name of a containing labeled statement. When used in this form, `break` causes the Java interpreter to immediately exit the named block, which can be any kind of statement, not just a loop or `switch`. For example:

```
TESTFORNULL: if (data != null) {
   for(int row = 0; row < numrows; row++) {
     for(int col = 0; col < numcols; col++) {
       if (data[row][col] == null)
         break TESTFORNULL;          // treat the array as undefined.
     }
   }
}  // Java interpreter goes here after executing break TESTFORNULL
```

The continue Statement

While a `break` statement exits a loop, a `continue` statement quits the current iteration of a loop and starts the next one. `continue`, in both its unlabeled and labeled forms, can be used only within a `while`, `do`, or `for` loop. When used without a label, `continue` causes the innermost loop to start a new iteration. When used with a label that is the name of a containing loop, it causes the named loop to start a new iteration. For example:

```
for(int i = 0; i < data.length; i++) {  // Loop through data.
   if (data[i] == -1)                   // If a data value is missing,
     continue;                          // skip to the next iteration.
   process(data[i]);                    // Process the data value.
}
```

`while`, `do`, and `for` loops differ slightly in the way that `continue` starts a new iteration:

- With a `while` loop, the Java interpreter simply returns to the top of the loop, tests the loop condition again, and, if it evaluates to `true`, executes the body of the loop again.
- With a `do` loop, the interpreter jumps to the bottom of the loop, where it tests the loop condition to decide whether to perform another iteration of the loop.
- With a `for` loop, the interpreter jumps to the top of the loop, where it first evaluates the *`update`* expression and then evaluates the *`test`* expression to decide whether to loop again. As you can see from the examples, the behavior of a `for` loop with a `continue` statement is different from the behavior of the "basically equivalent" `while` loop presented earlier; *`update`* gets evaluated in the `for` loop but not in the equivalent `while` loop.

The return Statement

A `return` statement tells the Java interpreter to stop executing the current method. If the method is declared to return a value, the `return` statement must be followed by an expression. The value of the expression becomes the return value of the method. For example, the following method computes and returns the square of a number:

```
double square(double x) {       // A method to compute x squared
   return x * x;                // Compute and return a value
}
```

Some methods are declared `void` to indicate that they do not return any value. The Java interpreter runs methods like this by executing their statements one by one until it reaches the end of the method. After executing the last statement, the interpreter returns implicitly. Sometimes, however, a `void` method has to return explicitly before reaching the last statement. In this case, it can use the `return` statement by itself, without any expression. For example, the following method prints, but does not return, the square root of its argument. If the argument is a negative number, it returns without printing anything:

```
// A method to print square root of x
void printSquareRoot(double x) {
   if (x < 0) return;               // If x is negative, return
   System.out.println(Math.sqrt(x)); // Print the square root of x
}                                   // Method end: return implicitly
```

The synchronized Statement

Java has always provided support for multithreaded programming. We cover this in some detail later on (especially in "Java's Support for Concurrency" on page 208)—but the reader should be aware that concurrency is difficult to get right, and has a number of subtleties.

In particular, when working with multiple threads, you must often take care to prevent multiple threads from modifying an object simultaneously in a way that might corrupt the object's state. Java provides the `synchronized` statement to help the programmer prevent corruption. The syntax is:

```
synchronized ( expression ) {
   statements
}
```

`expression` is an expression that must evaluate to an object or an array. *`statements`* constitute the code of the section that could cause damage and must be enclosed in curly braces.

Before executing the statement block, the Java interpreter first obtains an exclusive lock on the object or array specified by *`expression`*. It holds the lock until it is finished running the block, then releases it. While a thread holds the lock on an object, no other thread can obtain that lock.

The `synchronized` keyword is also available as a method modifier in Java, and when applied to a method, the `synchronized` keyword indicates that the entire method is locked. For a `synchronized` class method (a static method), Java obtains an exclusive lock on the class before executing the method. For a `synchronized` instance method, Java obtains an exclusive lock on the class instance. (Class and instance methods are discussed in Chapter 3.)

The throw Statement

An *exception* is a signal that indicates some sort of exceptional condition or error has occurred. To *throw* an exception is to signal an exceptional condition. To *catch* an exception is to handle it—to take whatever actions are necessary to recover from it. In Java, the `throw` statement is used to throw an exception:

```
throw expression;
```

The *`expression`* must evaluate to an exception object that describes the exception or error that has occurred. We'll talk more about types of exceptions shortly; for now, all you need to know is that an exception is represented by an object, which has a slightly specialized role. Here is some example code that throws an exception:

```
public static double factorial(int x) {
   if (x < 0)
     throw new IllegalArgumentException("x must be >= 0");
   double fact;
   for(fact=1.0; x > 1; fact *= x, x--)
     /* empty */ ;          // Note use of the empty statement
   return fact;
}
```

When the Java interpreter executes a `throw` statement, it immediately stops normal program execution and starts looking for an exception handler that can catch, or handle, the exception. Exception handlers are written with the `try/catch/finally`