As we've just seen, the comparison operators compare their operands and yield a boolean result, which is often used in branching and looping statements. In order to make branching and looping decisions based on conditions more interesting than a single comparison, you can use the boolean (or logical) operators to combine multiple comparison expressions into a single, more complex expression. The boolean operators require their operands to be boolean values and they evaluate to boolean values. The operators are:

Conditional AND (&&)

This operator performs a boolean AND operation on its operands. It evaluates to true if and only if both its operands are true. If either or both operands are false, it evaluates to false. For example:

```
if (x < 10 && y > 3) ... // If both comparisons are true
```

This operator (and all the boolean operators except the unary ! operator) have a lower precedence than the comparison operators. Thus, it is perfectly legal to write a line of code like the one above. However, some programmers prefer to use parentheses to make the order of evaluation explicit:

```
if ((x < 10) && (y > 3)) ...
```

You should use whichever style you find easier to read.

This operator is called a conditional AND because it conditionally evaluates its second operand. If the first operand evaluates to false, the value of the expression is false, regardless of the value of the second operand. Therefore, to increase efficiency, the Java interpreter takes a shortcut and skips the second operand. Since the second operand is not guaranteed to be evaluated, you must use caution when using this operator with expressions that have side effects. On the other hand, the conditional nature of this operator allows us to write Java expressions such as the following:

```
if (data != null && i < data.length && data[i] != -1)
    ...
```

The second and third comparisons in this expression would cause errors if the first or second comparisons evaluated to false. Fortunately, we don't have to worry about this because of the conditional behavior of the && operator.

Conditional OR (||)

This operator performs a boolean OR operation on its two boolean operands. It evaluates to true if either or both of its operands are true. If both operands are false, it evaluates to false. Like the && operator, || does not always evaluate its second operand. If the first operand evaluates to true, the value of the expression is true, regardless of the value of the second operand. Thus, the operator simply skips the second operand in that case.

Boolean NOT (!)

This unary operator changes the boolean value of its operand. If applied to a true value, it evaluates to false, and if applied to a false value, it evaluates to true. It is useful in expressions like these:

```
if (!found) ...          // found is a boolean variable declared somewhere
while (!c.isEmpty()) ... // The isEmpty() method returns a boolean value
```

Because ! is a unary operator, it has a high precedence and often must be used with parentheses:

```
if (!(x > y && y > z))
```

Boolean AND (&)

When used with boolean operands, the & operator behaves like the && operator, except that it always evaluates both operands, regardless of the value of the first operand. This operator is almost always used as a bitwise operator with integer operands, however, and many Java programmers would not even recognize its use with boolean operands as legal Java code.

Boolean OR (|)

This operator performs a boolean OR operation on its two boolean operands. It is like the || operator, except that it always evaluates both operands, even if the first one is true. The | operator is almost always used as a bitwise operator on integer operands; its use with boolean operands is very rare.

Boolean XOR (^)

When used with boolean operands, this operator computes the Exclusive OR (XOR) of its operands. It evaluates to true if exactly one of the two operands is true. In other words, it evaluates to false if both operands are false or if both operands are true. Unlike the && and || operators, this one must always evaluate both operands. The ^ operator is much more commonly used as a bitwise operator on integer operands. With boolean operands, this operator is equivalent to the != operator.

Excerpts from Flanagan, D. (2005). *Java in a Nutshell*. Cambridge, MA. O'Reilly.

71

The assignment operators store, or assign, a value into some kind of variable. The left operand must evaluate to an appropriate local variable, array element, or object field. The right side can be any value of a type compatible with the variable. An assignment expression evaluates to the value that is assigned to the variable. More importantly, however, the expression has the side effect of actually performing the assignment. Unlike all other binary operators, the assignment operators are right-associative, which means that the assignments in `a=b=c` are performed right-to-left, as follows: `a=(b=c)`.

The basic assignment operator is `=`. Do not confuse it with the equality operator, `==`. In order to keep these two operators distinct, I recommend that you read `=` as "is assigned the value."

The `instanceof` operator requires an object or array value as its left operand and the name of a reference type as its right operand. It evaluates to `true` if the object or array is an *instance* of the specified type; it returns `false` otherwise. If the left operand is `null`, `instanceof` always evaluates to `false`. If an `instanceof` expression evaluates to `true`, it means that you can safely cast and assign the left operand to a variable of the type of the right operand.

The `instanceof` operator can be used only with reference types and objects, not primitive types and values. Examples of `instanceof` are:

```
"string" instanceof String      // True: all strings are instances of String
"" instanceof Object            // True: strings are also instances of Object
null instanceof String          // False: null is never an instance of anything

Object o = new int[] {1,2,3};
o instanceof int[]   // True: the array value is an int array
o instanceof byte[]  // False: the array value is not a byte array
o instanceof Object  // True: all arrays are instances of Object

// Use instanceof to make sure that it is safe to cast an object
if (object instanceof Point) {
   Point p = (Point) object;
}
```

Excerpts from Flanagan, D. (2005). *Java in a Nutshell*. Cambridge, MA. O'Reilly.

72

Object member access (.)

An *object* is a collection of data and methods that operate on that data; the data fields and methods of an object are called its members. The dot (`.`) operator accesses these members. If `o` is an expression that evaluates to an object reference, and `f` is the name of a field of the object, `o.f` evaluates to the value contained in that field. If `m` is the name of a method, `o.m` refers to that method and allows it to be invoked using the `( )` operator shown later.

Array element access ([ ])

An *array* is a numbered list of values. Each element of an array can be referred to by its number, or *index*. The `[ ]` operator allows you to refer to the individual elements of an array. If `a` is an array, and `i` is an expression that evaluates to an `int`, `a[i]` refers to one of the elements of `a`. Unlike other operators that work with integer values, this operator restricts array index values to be of type `int` or narrower.

Method invocation (( ))

A *method* is a named collection of Java code that can be run, or *invoked*, by following the name of the method with zero or more comma-separated expressions contained within parentheses. The values of these expressions are the *arguments* to the method. The method processes the arguments and optionally returns a value that becomes the value of the method invocation expression. If `o.m` is a method that expects no arguments, the method can be invoked with `o.m( )`. If the method expects three arguments, for example, it can be invoked with an expression such as `o.m(x,y,z)`. Before the Java interpreter invokes a method, it evaluates each of the arguments to be passed to the method. These expressions are guaranteed to be evaluated in order from left to right (which matters if any of the arguments have side effects).

Object creation (new)

In Java, objects (and arrays) are created with the `new` operator, which is followed by the type of the object to be created and a parenthesized list of arguments to be passed to the object *constructor*. A constructor is a special method that initializes a newly created object, so the object creation syntax is similar to the Java method invocation syntax. For example:

```
new ArrayList();
new Point(1,2)
```