5.9 Logical Operators

The `if`, `if…else`, `while`, `do…while` and `for` statements each require a *condition* to determine how to continue a program's flow of control. So far, we've studied only simple conditions, such as `count <= 10`, `number != sentinelValue` and `total > 1000`. Simple conditions are expressed in terms of the relational operators `>`, `<`, `>=` and `<=` and the equality operators `==` and `!=`, and each expression tests only one condition. To test *multiple* conditions in the process of making a decision, we performed these tests in separate statements or in nested `if` or `if…else` statements. Sometimes control statements require more complex conditions to determine a program's flow of control.

Java's **logical operators** enable you to *combine* simple conditions into more complex ones. The logical operators are `&&` (conditional AND), `||` (conditional OR), `&` (boolean logical AND), `|` (boolean logical inclusive OR), `^` (boolean logical exclusive OR) and `!` (logical NOT). [*Note:* The `&`, `|` and `^` operators are also bitwise operators when they're applied to integral operands. We discuss the bitwise operators in online Appendix K.]

5.9.1 Conditional AND (&&)

Operator

Suppose we wish to ensure at some point in a program that two conditions are *both* `true` before we choose a certain path of execution. In this case, we can use the && (**conditional AND**) operator, as follows:

```
if (gender == FEMALE && age >= 65) {
   ++seniorFemales;
}
```

This `if` statement contains two simple conditions. The condition `gender == FEMALE` com-pares variable `gender` to the constant `FEMALE` to determine whether a person is female. The condition `age >= 65` might be evaluated to determine whether a person is a senior citizen. The `if` statement considers the combined condition

```
gender == FEMALE && age >= 65
```

which is `true` if and only if *both* simple conditions are `true`. In this case, the `if` statement's body increments `seniorFemales` by `1`. If either or both of the simple conditions are `false`, the program skips the increment.

Similarly, the following condition ensures that a `grade` is in the range 1–100:

```
grade >= 1 && grade <= 100
```

This condition is `true` if and only if `grade` is greater than or equal to `1` *and* `grade` is less than or equal to `100`. Some programmers find that the preceding combined condition is more readable when *redundant* parentheses are added, as in:

```
(grade >= 1) && (grade <= 100)
```

The table in Fig. 5.15 summarizes the && operator, showing all four possible combinations of `false` and `true` values for *expression1* and *expression2*. Such tables are called **truth tables**. Java evaluates to `false` or `true` all expressions that include relational operators, equality operators or logical operators.

expression1	**expression2**	**expression1** && **expression2**
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Fig. 5.15

&& (conditional AND) operator truth table.

5.9.2 Conditional OR (||) Operator

Now suppose we wish to ensure that *either or both* of two conditions are `true` before we choose a certain path of execution. In this case, we use the `||` (**conditional OR**) operator, as in the following program segment:

```
if ((semesterAverage >= 90) || (finalExam >= 90)) {
   System.out.println ("Student grade is A");
}
```

This statement also contains two simple conditions. The condition `semesterAverage >= 90` evaluates to determine whether the student deserves an A in the course because of a solid performance throughout the semester. The condition `finalExam >= 90` evaluates to determine whether the student deserves an A in the course because of an outstanding performance on the final exam. The `if` statement then considers the combined condition

```
(semesterAverage >= 90) || (finalExam >= 90)
```

and awards the student an A if *either or both* of the simple conditions are `true`. The only time the message `"Student grade is A"` is *not* printed is when *both* of the simple conditions are `false`. Figure 5.16 is a truth table for operator conditional OR (`||`). Operator &&has a higher precedence

than operator `||`. Both operators associate from left to right.

expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 5.16

`||` (conditional OR) operator truth table.

5.9.3 Short-Circuit Evaluation of Complex Conditions

The parts of an expression containing `&&` or `||` operators are evaluated *only* until it's known whether the condition is `true` or `false`. Thus, evaluation of the expression

```
(gender == FEMALE) && (age >= 65)
```

stops immediately if `gender` *is not* equal to `FEMALE` (that is,

the entire expression is `false`) and continues if `gender` *is* equal to `FEMALE` (that is, the entire expression could still be `true` if the condition `age >= 65` is `true`). This feature of conditional AND and conditional OR expressions is called **short-circuit evaluation**.

 Common Programming Error 5.8

In expressions using &&, a condition—we'll call this the dependent condition*—may require another condition to be* `true` *for the dependent condition's evaluation to be meaningful. In this case, the dependent condition should be placed* after *the* && *operator to prevent errors. Consider the expression* `(i != 0) && (10 / i == 2)`. *The dependent condition* `(10/i==2)` must *appear after the* && *to prevent the possibility of division by zero.*

5.9.4 Boolean Logical AND (&) and Boolean Logical Inclusive OR (|) Operators

The **boolean logical AND** (&) and **boolean logical inclusive OR (|)** operators are identical to the && and || operators, except that the & and | operators *always* evaluate *both* of their

operands (i.e., they do *not* perform short-circuit evaluation). So, the expression

```
(gender == 1) & (age >= 65)
```

evaluates `age >= 65` *regardless* of whether `gender` is equal to `1`. This is useful if the right operand has a required **side effect**—a modification of a variable's value. For example,

```
(birthday == true) | (++age >= 65)
```

guarantees that the condition `++age >= 65` will be evaluated. Thus, the variable `age` is incremented, regardless of whether the overall expression is `true` or `false`.

Error-Prevention Tip 5.11

For clarity, avoid expressions with side effects (such as assignments) in conditions. They can make code harder to understand and can lead to subtle logic errors.

Error-Prevention Tip

5.12

Assignment (=) expressions generally should not *be used in conditions. Every condition* must *result in a* `boolean` *value; otherwise, a compilation error occurs. In a condition, an assignment will compile* only *if a* `boolean` *expression is assigned to a* `boolean` *variable.*

5.9.5 Boolean Logical Exclusive OR (^)

A simple condition containing the **boolean logical exclusive OR (^)** operator is `true` *if and only if one of its operands is* `true` *and the other is* `false`. If both are `true` or both are `false`, the entire condition is `false`. Figure 5.17 is a truth table for the boolean logical exclusive OR operator (^). This operator is guaranteed to evaluate *both* of its operands.

expression1	expression2	expression1 ^ expression2
false	false	false
false	true	true
true	false	true
true	true	false

Fig. 5.17

^ (boolean logical exclusive OR) operator truth table.

5.9.6 Logical Negation (!) Operator

The `!` (**logical NOT**, also called **logical negation** or **logical complement**) operator "reverses" the meaning of a condition. Unlike the logical operators `&&`, `||`, `&`, `|` and `^`, which are ***binary*** operators that combine two conditions, the logical negation operator is a *unary* operator that has only one condition as an operand. The operator is placed *before* a condition to choose a path of execution if the original condition (without the logical negation operator) is `false`, as in the program segment

```
if (! (grade == sentinelValue)) {
   System.out.printf("The next grade is %d%n", grade)
}
```

which executes the `printf` call only if `grade` is *not* equal to `sentinelValue`. The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a *higher* precedence than the equality operator.

In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the previous statement may also be written as follows:

```
if (grade != sentinelValue) {
   System.out.printf("The next grade is %d%n", grade)
}
```

This flexibility can help you express a condition in a more convenient manner. Figure 5.18 is a truth table for the logical negation operator.

expression	!expression
false	true
true	false

Fig. 5.18

! (logical NOT) operator truth table.

5.9.7 Logical Operators Example

Figure 5.19 uses logical operators to produce the truth tables

discussed in this section. The output shows the `boolean` expression that was evaluated and its result. We used the **`%b` format specifier** to display the word “true” or the word “false” based on a `boolean` expression’s value. Lines 7–11 produce the truth table for `&&`. Lines 14–18 produce the truth table for `||`. Lines 21–25 produce the truth table for `&`. Lines 28–33 produce the truth table for `|`. Lines 36–41 produce the truth table for `^`. Lines 44–45 produce the truth table for `!`.

```
      1  // Fig. 5.19: LogicalOperators.java
            2  // Logical operators.
                        3
          4  public class LogicalOperators {
  5       public static void main(String[] args) {
 6          // create truth table for && (conditional A
 7          System.out.printf("%s%n%s: %b%n%s: %b%n%s:
 8             "Conditional AND (&&)", "false && false"
  9             "false && true",(false && true),
  10            "true && false",(true && false),
   11            "true && true",)(true && true);
                        12
 13         // create truth table for || (conditional O
 14         System.out.printf("%s%n%s: %b%n%s: %b%n%s:
 15            "Conditional OR (||)", "false || false",
   17            "true || false",(false || true),
   16            "false || true",(true || false),
    18           "true || true",)(true || true);
                        19
 20         // create truth table for & (boolean logica
 21         System.out.printf("%s%n%s: %b%n%s: %b%n%s:
 22            "Boolean logical AND (&)", "false & fals
    23            "false & true",(false & true),
     24           "true & false",(true & false),
      25            "true & true",(true & true);
                        26
 27         // create truth table for | (boolean logica
```

```
28          System.out.printf("%s%n%s: %b%n%s: %b%n%s:
 29             "Boolean logical inclusive OR (|)",
  30             "false | false",(false | false),
   31             "false | true",(false | true),
   32             "true | false",(true | false),
    33             "true | true",)(true | true);
                          34
35          // create truth table for ^ (boolean logica
36          System.out.printf("%s%n%s: %b%n%s: %b%n%s:
 37             "Boolean logical exclusive OR (^)",
  38             "false ^ false",(false ^ false),
   39             "false ^ true",(false ^ true),
   40             "true ^ false",(true ^ false),
    41             "true ^ true",)(true ^ true);
                          42
43          // create truth table for ! (logical negati
44          System.out.printf("%s%n%s: %b%n%s: %b%n", "
 45             "!false",(!false),"!true",(!true));
                        46     }
                         47  }
```

```
Conditional AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Conditional OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Boolean logical AND (&)
false & false: false
false & true: false
true & false: false
true & true: true
```

```
Boolean logical inclusive OR (|)
      false | false: false
       false | true: true
       true | false: true
        true | true: true

Boolean logical exclusive OR (^)
      false ^ false: false
       false ^ true: true
       true ^ false: true
       true ^ true: false

         Logical NOT (!)
           !false: true
           !true: false
```

Fig. 5.19

Logical operators.

Precedence and Associativity of the Operators Presented So Far

Figure 5.20 shows the precedence and associativity of the Java operators introduced so far. The operators are shown from top to bottom in decreasing order of precedence.

Operators	Associativity	Type
`++ --`	right to left	unary postfix
`++ -- + - ! (`*`type`*`)`	right to left	unary prefix
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`&`	left to right	boolean logical AND
`^`	left to right	boolean logical exclusive OR
`|`	left to right	boolean logical inclusive OR
`&&`	left to right	conditional AND
`||`	left to right	conditional OR
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment

Fig. 5.20

Precedence/associativity of the operators discussed so far.