

Relational Expressions

We have used the term *expression* quite a bit in the preceding notes. For the most part, the idea is straightforward. An expression is simply one or more variables and/or constants joined by operators. An expression is evaluated and produces a result. The result of all expressions thus far was either an integer value or a floating-point value.

But, an expression can also yield a **boolean**, a result that is either **true** or **false**. Such an expression is called a *relational expression*. The result reflects how something "relates to" something else. For example, "Is the value of *x* greater than the value of *y*?" Note that the preceding poses a question. Relational expressions are usually intended to answer yes/no, or true/false, questions. Obviously, `boolean` values and `boolean` variables play an important role in relational expressions.

Relational Operators

To build relational expressions, two types of operators are used: *relational operators* and *logical operators*. Let's deal first with the relational operators. There are six relational operators, four of equal precedence:

- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal to

and two just below these in precedence:

- == equal to
- != not equal to

These operators, like the arithmetic operators met earlier, are sometimes called *binary* operators, because they take two arguments — one on each side. The arguments are generally integers or floating-point variables or constants. The result is a `boolean`. The following are examples of relational expressions built from relational operators: (the result is also shown)

<code>3 < 4</code>	<code>true</code>
<code>7.6 <= 9</code>	<code>true</code>
<code>4 == 7</code>	<code>false</code>
<code>8.3 != 2.1</code>	<code>true</code>

In the second line, a `double` is compared with an `int`. The `int` is promoted to `double` before the expression is evaluated.

So, what can you do with a relational expression? Since a relational expression yields either `true` or `false`, the result can be assigned to a `boolean` variable. Furthermore, a `boolean` variable, like an `int` or `double`, can be printed. So, the following two statements

```
boolean b = 8.3 != 2.1;
System.out.println(b);
```

print `"true"` on the standard output. A much more common use of relational expressions is to control program flow in `if` statements or in `while`, `do/while`, or `for` loops. We will meet these later.

Of course, relational expressions can also employ variables, as in the following sequence of Java statements:

```
int x = 3;
int y = 4;
boolean b = x > y;
System.out.println(b);
```

In the third line, the relational expression `x > y` is evaluated and the result, `false`, is assigned to the boolean variable `b`. The fourth statement prints "false".

Logical Operators

There are three logical operators:

&& AND (true if both arguments are true, false otherwise)
|| OR (true if either argument is true, false otherwise)
! NOT (true if argument is false, false otherwise)

The descriptions in parentheses are usually laid out in *truth tables*, as shown in Figure 1.

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

(a)

a	b	a b
false	false	false
false	true	true
true	false	true
true	true	true

(b)

a	!a
false	true
true	false

(c)

Figure 1. Truth tables for logical operators. (a) AND (b) OR (c) NOT

Logical operators are similar to relational operators in that they both produce `boolean` results. However, they differ in that logical operators also use `boolean` arguments. So the following statements make sense:

```
boolean a = true;
boolean b = false;
boolean c = a && b;
```

whereas the third statement below is nonsense:

```
int a = 3;
int b = 4;
boolean c = a && b; // wrong! && requires boolean arguments
```

The following statements, however, are perfectly reasonable:

```
int a = 3;
int b = 4;
int c = 5;
int d = 6;
boolean e = a < b && c < d;
System.out.println(e);
```

and will print "true" on the standard output. This result is by no means obvious, until we acknowledge the precedence relationship between the relational and logical operators. The relational operators are of higher precedence than the logical AND and logical OR operators. The parentheses in the following statement illustrate this (although they are not necessary):

```
boolean e = (a < b) && (c < d);
```

The expression `a < b` is `true` (because 3 is less than 4) and the expression `c < d` is `true` (because 5 is less than 6). The `&&` (AND) operator yields `true` if both operands are `true`, which they are in this example. So, the expression `(a < b) && (c < d)` is `true`, and this result is assigned to the `boolean` variable `e`.

The `&&` (AND) and `||` (OR) operators are binary operators, because they take two arguments. The `!` (NOT) operator is a *unary* operator, because it takes one argument. It returns the logical complement of its argument. So, the following statements

```
boolean b = true;
System.out.println(!b);
```

print `"false"` on the standard output. Note the `!` operator does not change the value of the `boolean` variable `b`; it simply returns a value which is the logical complement of `b`. Unary operators, in general, bind very tightly to their arguments; and the unary `!` operator is no exception. It is of higher precedence than all the logical or relational operators.

Lazy Evaluation

Logical operators exhibit a behaviour known as *lazy evaluation*. The evaluation of an expression with logical operators "stops" as soon as the outcome is certain. Consider the following statements:

```
int a = 3;
int b = 4;
int c = 5;
int d = 6;
boolean result = a == b && c < d;
```

The expressions in the last statement would normally be evaluated in the following order:

```
1st:  a == b
2nd:  c < d
3rd:  a == b && c < d
4th:  result = a == b && c < d
```

However, the 2nd and 3rd steps are not needed. In the first step, `a == b` yields `false` because `a` (3) does not equal `b` (4). It is not necessary to evaluate the expression `c < d`, because the outcome of the `&&` (AND) operation is now certain. It is certain because `"false && anything"` equals `false`. Review the truth table in Figure 1a, if you need to convince yourself of this. The expressions in the last statement are evaluated as follows:

```
1st:  a == b           (the answer is false)
2nd:  result = false;  (done!)
```

Lazy evaluation can cause subtle side effects if it is not properly understood and accounted for in the design of Java programs. It can also be used to advantage to implement safeguards. For example, the following expression

```
(z != 0) && (x / z <= y / z)
```

uses lazy evaluation to ensure a divide-by-zero error does not occur. (Parentheses are shown to clarify the operation, but they are not needed.) If `z` equals zero the first expression yields `false` and, due to lazy evaluation, the second expression is not evaluated. If `z` does not equal zero, the

first expression yields `true` and, therefore, the second expression is evaluated. In this case, the expression `y / z` can be safely evaluated because `z` does not equal zero.

Lazy evaluation also occurs for the `||` (OR) operator. Consider the following relational expression:

```
rel_exp1 || rel_exp2
```

If `rel_exp1` is `true`, `rel_exp2` is not evaluated because the final result is certain: it is `true`. For a review of the `||` operator, see Figure 1b.

More examples of lazy evaluation will be given later.