[Previous Section | Next Section | Chapter Index | Main Index]

Section 2.5

Details of Expressions

Subsections

THIS SECTION TAKES A CLOSER LOOK at expressions. Recall that an expression is a piece of program code that represents or computes a value. An expression can be a literal, a variable, a function call, or several of these things combined with operators such as + and >. The value of an expression can be assigned to a variable, used as a parameter in a subroutine call, or combined with other values into a more complicated expression. (The value can even, in some cases, be ignored, if that's what you want to do; this is more common than you might think.) Expressions are an essential part of programming. So far, this book has dealt only informally with expressions. This section tells you the more-or-less complete story (leaving out some of the less commonly used operators).

The basic building blocks of expressions are literals (such as `674`, `3.14`, `true`, and `'X'`), variables, and function calls. Recall that a function is a subroutine that returns a value. You've already seen some examples of functions, such as the input routines from the TextIO class and the mathematical functions from the Math class.

The Math class also contains a couple of mathematical constants that are useful in mathematical expressions: `Math.PI` represents π (the ratio of the circumference of a circle to its diameter), and `Math.E` represents *e* (the base of the natural logarithms). These "constants" are actually member variables in Math of type double. They are only approximations for the mathematical constants, which would require an infinite number of digits to specify exactly. The standard class Integer contains a couple of constants related to the int data type: `Integer.MAX_VALUE` is the largest possible int, 2147483647, and `Integer.MIN_VALUE` is the smallest int, -2147483648. Similarly, the class Double contains some constants related to type double. `Double.MAX_VALUE` is the largest value of type double, and `Double.MIN_VALUE` is the smallest **positive** value. It also has constants to represent infinite values, `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY`, and the special value `Double.NaN` to represent an undefined value. For example, the value of `Math.sqrt(-1)` is Double.NaN.

Literals, variables, and function calls are simple expressions. More complex expressions can be built up by using operators to combine simpler expressions. Operators include + for adding two numbers, > for comparing two values, and so on. When several operators appear in an expression, there is a question of precedence, which determines how the operators are grouped for evaluation. For example, in the expression "`A + B * C`", `B*C` is computed first and then the result is added to `A`. We say that multiplication (*) has higher precedence than addition (+). If the default precedence is not what you want, you can use parentheses to explicitly specify the grouping you want. For example, you could use "`(A + B) * C`" if you want to add `A` to `B` first and then multiply the result by `C`.

The rest of this section gives details of operators in Java. The number of operators in Java is quite large. I will not cover them all here, but most of the important ones are here.

2.5.1 Arithmetic Operators

Arithmetic operators include addition, subtraction, multiplication, and division. They are indicated by +, -, *, and /. These operations can be used on values of any numeric type: byte, short, int, long, float, or double. (They can also be used with values of type char, which are treated as integers in this

context; a char is converted into its Unicode code number when it is used with an arithmetic operator.) When the computer actually calculates one of these operations, the two values that it combines must be of the same type. If your program tells the computer to combine two values of different types, the computer will convert one of the values from one type to another. For example, to compute 37.4 + 10, the computer will convert the integer 10 to a real number 10.0 and will then compute 37.4 + 10.0. This is called a type conversion. Ordinarily, you don't have to worry about type conversion in expressions, because the computer does it automatically.

When two numerical values are combined (after doing type conversion on one of them, if necessary), the answer will be of the same type. If you multiply two ints, you get an int; if you multiply two doubles, you get a double. This is what you would expect, but you have to be very careful when you use the division operator `/`. When you divide two integers, the answer will always be an integer; if the quotient has a fractional part, it is discarded. For example, the value of `7/2` is `3`, not `3.5`. If `N` is an integer variable, then `N/100` is an integer, and `1/N` is equal to zero for any `N` greater than one! This fact is a common source of programming errors. You can force the computer to compute a real number as the answer by making one of the operands real: For example, when the computer evaluates `1.0/N`, it first converts `N` to a real number in order to match the type of `1.0`, so you get a real number as the answer.

Java also has an operator for computing the remainder when one number is divided by another. This operator is indicated by `%`. If `A` and `B` are integers, then `A % B` represents the remainder when `A` is divided by `B`. (However, for negative operands, `%` is not quite the same as the usual mathematical "modulus" operator, since if one of `A` or `B` is negative, then the value of `A % B` will be negative.) For example, `7 % 2` is `1`, while `34577 % 100` is `77`, and `50 % 8` is `2`. A common use of `%` is to test whether a given integer is even or odd: `N` is even if `N % 2` is zero, and it is odd if `N % 2` is `1`. More generally, you can check whether an integer `N` is evenly divisible by an integer `M` by checking whether `N % M` is zero.

The `%` operator also works with real numbers. In general, `A % B` is what is left over after you remove as many copies of `B` as possible from `A`. For example, `7.52 % 0.5` is 0.02.

Finally, you might need the unary minus operator, which takes the negative of a number. For example, `-X` has the same value as `(-1)*X`. For completeness, Java also has a unary plus operator, as in `+X`, even though it doesn't really do anything.

By the way, recall that the `+` operator can also be used to concatenate a value of any type onto a String. When you use `+` to combine a string with a value of some other type, it is another example of type conversion, since any type can be automatically converted into type String.

2.5.2 Increment and Decrement

You'll find that adding `1` to a variable is an extremely common operation in programming. Subtracting `1` from a variable is also pretty common. You might perform the operation of adding `1` to a variable with assignment statements such as:

```
counter  =  counter + 1;
goalsScored  =  goalsScored + 1;
```

The effect of the assignment statement `x = x + 1` is to take the old value of the variable `x`, compute the result of adding `1` to that value, and store the answer as the new value of `x`. The same operation can be accomplished by writing `x++` (or, if you prefer, `++x`). This actually changes the value of `x`, so that it has the same effect as writing "`x = x + 1`". The two statements above could be written

```
counter++;
goalsScored++;
```

Similarly, you could write `x--` (or `--x`) to subtract `1` from `x`. That is, `x--` performs the same computation as `x = x - 1`. Adding `1` to a variable is called incrementing that variable, and subtracting `1` is called decrementing. The operators `++` and `--` are called the increment operator and the decrement operator, respectively. These operators can be used on variables belonging to any of the numerical types and also on variables of type char. (`'A'++` is `'B'`.)

Usually, the operators `++` or `--` are used in statements like "`x++;`" or "`x--;`". These statements are commands to change the value of `x`. However, it is also legal to use `x++`, `++x`, `x--`, or `--x` as expressions, or as parts of larger expressions. That is, you can write things like:

```
y = x++;
y = ++x;
TextIO.putln(--x);
z = (++x) * (y--);
```

The statement "`y = x++;`" has the effects of adding `1` to the value of `x` and, in addition, assigning some value to `y`. The value assigned to `y` is the value of the expression `x++`, which is defined to be the **old** value of `x`, before the `1` is added. Thus, if the value of `x` is `6`, the statement "`y = x++;`" will change the value of `x` to `7`, but it will change the value of `y` to `6` since the value assigned to `y` is the **old** value of `x`. On the other hand, the value of `++x` is defined to be the **new** value of `x`, after the `1` is added. So if `x` is `6`, then the statement "`y = ++x;`" changes the values of both `x` and `y` to `7`. The decrement operator, `--`, works in a similar way.

Note in particular that the statement `x = x++;` **does not change the value of x**! This is because the value that is being assigned to x is the old value of x, the one that it had before the statement was executed. The net result is that x is incremented but then immediately changed back to its previous value! You also need to remember that `x++` is **not** the same as `x + 1`. The expression `x++` changes the value of x; the expression `x + 1` does not.

This can be confusing, and I have seen many bugs in student programs resulting from the confusion. My advice is: Don't be confused. Use `++` and `--` only as stand-alone statements, not as expressions. I will follow this advice in almost all examples in these notes.

2.5.3 Relational Operators

Java has boolean variables and boolean-valued expressions that can be used to express conditions that can be either `true` or `false`. One way to form a boolean-valued expression is to compare two values using a relational operator. Relational operators are used to test whether two values are equal, whether one value is greater than another, and so forth. The relational operators in Java are: `==`, `!=`, `<`, `>`, `<=`, and `>=`. The meanings of these operators are:

```
A == B       Is A "equal to" B?
A != B       Is A "not equal to" B?
A < B        Is A "less than" B?
A > B        Is A "greater than" B?
A <= B       Is A "less than or equal to" B?
A >= B       Is A "greater than or equal to" B?
```

These operators can be used to compare values of any of the numeric types. They can also be used to compare values of type char. For characters, `<` and `>` are defined according the numeric Unicode values of the characters. (This might not always be what you want. It is not the same as alphabetical order because all the upper case letters come before all the lower case letters.)

When using boolean expressions, you should remember that as far as the computer is concerned, there is nothing special about boolean values. In the next chapter, you will see how to use them in loop and

branch statements. But you can also assign boolean-valued expressions to boolean variables, just as you can assign numeric values to numeric variables. And functions can return boolean values.

By the way, the operators == and != can be used to compare boolean values too. This is occasionally useful. For example, can you figure out what this does:

```
boolean sameSign;
sameSign = ((x > 0) == (y > 0));
```

One thing that you **cannot** do with the relational operators <, >, <=, and >= is to use them to compare values of type String. You can legally use == and != to compare Strings, but because of peculiarities in the way objects behave, they might not give the results you want. (The == operator checks whether two objects are stored in the same memory location, rather than whether they contain the same value. Occasionally, for some objects, you do want to make such a check -- but rarely for strings. I'll get back to this in a later chapter.) Instead, you should use the subroutines equals(), equalsIgnoreCase(), and compareTo(), which were described in Subsection 2.3.3, to compare two Strings.

Another place where == and != don't work as you would expect is with Double.NaN, the constant that represents an undefined value of type double. The values of x == Double.NaN and x != Double.NaN are both defined to be false in all cases, whether or not x is Double.NaN! To test whether a real value x is the undefined value Double.NaN, use the boolean-valued function Double.isNaN(x).

2.5.4 Boolean Operators

In English, complicated conditions can be formed using the words "and", "or", and "not." For example, "If there is a test **and** you did **not** study for it...". "And", "or", and "not" are boolean operators, and they exist in Java as well as in English.

In Java, the boolean operator "and" is represented by &&. The && operator is used to combine two boolean values. The result is also a boolean value. The result is true if **both** of the combined values are true, and the result is false if **either** of the combined values is false. For example, "(x == 0) && (y == 0)" is true if and only if both x is equal to 0 and y is equal to 0.

The boolean operator "or" is represented by ||. (That's supposed to be two of the vertical line characters, |.) The expression "A || B" is true if either A is true or B is true, or if both are true. "A || B" is false only if both A and B are false.

The operators && and || are said to be short-circuited versions of the boolean operators. This means that the second operand of && or || is not necessarily evaluated. Consider the test

```
(x != 0) && (y/x > 1)
```

Suppose that the value of x is in fact zero. In that case, the division y/x is undefined mathematically. However, the computer will never perform the division, since when the computer evaluates (x != 0), it finds that the result is false, and so it knows that ((x != 0) && **anything**) has to be false. Therefore, it doesn't bother to evaluate the second operand. The evaluation has been short-circuited and the division by zero is avoided. (This may seem like a technicality, and it is. But at times, it will make your programming life a little easier.)

The boolean operator "not" is a unary operator. In Java, it is indicated by ! and is written in front of its single operand. For example, if test is a boolean variable, then

```
test = ! test;
```

will reverse the value of test, changing it from true to false, or from false to true.

2.5.5 Conditional Operator

Any good programming language has some nifty little features that aren't really necessary but that let you feel cool when you use them. Java has the conditional operator. It's a ternary operator -- that is, it has three operands -- and it comes in two pieces, ? and :, that have to be used together. It takes the form

```
boolean-expression ? expression1 : expression2
```

The computer tests the value of **boolean-expression**. If the value is `true`, it evaluates **expression1**; otherwise, it evaluates **expression2**. For example:

```
next = (N % 2 == 0) ? (N/2) : (3*N+1);
```

will assign the value `N/2` to `next` if `N` is even (that is, if `N % 2 == 0` is `true`), and it will assign the value `(3*N+1)` to `next` if `N` is odd. (The parentheses in this example are not required, but they do make the expression easier to read.)

2.5.6 Assignment Operators and Type Conversion

You are already familiar with the assignment statement, which uses the symbol "=" to assign the value of an expression to a variable. In fact, = is really an operator in the sense that an assignment can itself be used as an expression or as part of a more complex expression. The value of an assignment such as `A=B` is the same as the value that is assigned to `A`. So, if you want to assign the value of `B` to `A` and test at the same time whether that value is zero, you could say:

```
if ( (A=B) == 0 )...
```

Usually, I would say, **don't do things like that**!

In general, the type of the expression on the right-hand side of an assignment statement must be the same as the type of the variable on the left-hand side. However, in some cases, the computer will automatically convert the value computed by the expression to match the type of the variable. Consider the list of numeric types: byte, short, int, long, float, double. A value of a type that occurs earlier in this list can be converted automatically to a value that occurs later. For example:

```
int A;
double X;
short B;
A = 17;
X = A;    // OK; A is converted to a double
B = A;    // illegal; no automatic conversion
          //       from int to short
```

The idea is that conversion should only be done automatically when it can be done without changing the semantics of the value. Any int can be converted to a double with the same numeric value. However, there are int values that lie outside the legal range of shorts. There is simply no way to represent the int 100000 as a short, for example, since the largest value of type short is 32767.

In some cases, you might want to force a conversion that wouldn't be done automatically. For this, you can use what is called a type cast. A type cast is indicated by putting a type name, in parentheses, in front of the value you want to convert. For example,

```
int A;
short B;
A = 17;
B = (short)A;  // OK; A is explicitly type cast
               //      to a value of type short
```

You can do type casts from any numeric type to any other numeric type. However, you should note that you might change the numeric value of a number by type-casting it. For example, `(short)100000` is -31072. (The -31072 is obtained by taking the 4-byte int 100000 and throwing away two of those bytes to obtain a short -- you've lost the real information that was in those two bytes.)

When you type-cast a real number to an integer, the fractional part is discarded. For example, `(int)7.9453` is `7`. As another example of type casts, consider the problem of getting a random integer between 1 and 6. The function `Math.random()` gives a real number between 0.0 and 0.9999..., and so `6*Math.random()` is between 0.0 and 5.999.... The type-cast operator, `(int)`, can be used to convert this to an integer: `(int)(6*Math.random())`. Thus, `(int)(6*Math.random())` is one of the integers 0, 1, 2, 3, 4, and 5. To get a number between 1 and 6, we can add 1: "`(int)(6*Math.random()) + 1`". (The parentheses around `6*Math.random()` are necessary because of precedence rules; without the parentheses, the type cast operator would apply only to the `6`.)

The type char is almost an integer type. You can assign char values to int variables, and you can assign numerical constants in the range 0 to 65535 to char variables. You can also use explicit type-casts between char and the numeric types. For example, `(char)97` is `'a'`, `(int)'+'` is `43`, and (char) ('A' + 2) is 'C'.

Type conversion between String and other types cannot be done with type-casts. One way to convert a value of any type into a string is to concatenate it with an empty string. For example, "`"" + 42`" is the string "`42`". But a better way is to use the function `String.valueOf(x)`, a static member function in the String class. `String.valueOf(x)` returns the value of `x`, converted into a string. For example, `String.valueOf(42)` is the string "`42`", and if `ch` is a char variable, then `String.valueOf(ch)` is a string of length one containing the single character that is the value of `ch`.

It is also possible to convert certain strings into values of other types. For example, the string "`10`" should be convertible into the int value 10, and the string "`17.42e-2`" into the double value 0.1742. In Java, these conversions are handled by built-in functions.

The standard class Integer contains a static member function for converting from String to int. In particular, if `str` is any expression of type String, then `Integer.parseInt(str)` is a function call that attempts to convert the value of `str` into a value of type int. For example, the value of `Integer.parseInt("10")` is the int value 10. If the parameter to `Integer.parseInt` does not represent a legal int value, then an error occurs.

Similarly, the standard class Double includes a function `Double.parseDouble`. If `str` is a String, then the function call `Double.parseDouble(str)` tries to convert `str` into a value of type double. An error occurs if `str` does not represent a legal double value.

Getting back to assignment statements, Java has several variations on the assignment operator, which exist to save typing. For example, "`A += B`" is defined to be the same as "`A = A + B`". Every operator in Java that applies to two operands, except for the relational operators, gives rise to a similar assignment operator. For example:

```
x -= y;     // same as:   x = x - y;
x *= y;     // same as:   x = x * y;
x /= y;     // same as:   x = x / y;
```

```
x %= y;     // same as:   x = x % y;
q &&= p;    // same as:   q = q && p;  (for booleans q and p)
```

The combined assignment operator `+=` even works with strings. Recall that when the `+` operator is used with a string as one of the operands, it represents concatenation. Since `str += x` is equivalent to `str = str + x`, when `+=` is used with a string on the left-hand side, it appends the value on the right-hand side onto the string. For example, if `str` has the value "tire", then the statement `str += 'd';` changes the value of `str` to "tired".

2.5.7 Precedence Rules

If you use several operators in one expression, and if you don't use parentheses to explicitly indicate the order of evaluation, then you have to worry about the precedence rules that determine the order of evaluation. (Advice: don't confuse yourself or the reader of your program; use parentheses liberally.)

Here is a listing of the operators discussed in this section, listed in order from highest precedence (evaluated first) to lowest precedence (evaluated last):

```
Unary operators:              ++, --, !, unary -, unary +, type-cast
Multiplication and division:  *,  /,  %
Addition and subtraction:     +,  -
Relational operators:         <,  >,  <=,  >=
Equality and inequality:      ==,  !=
Boolean and:                  &&
Boolean or:                   ||
Conditional operator:         ?:
Assignment operators:         =,  +=,  -=,  *=,  /=,  %=
```

Operators on the same line have the same precedence. When operators of the same precedence are strung together in the absence of parentheses, unary operators and assignment operators are evaluated right-to-left, while the remaining operators are evaluated left-to-right. For example, `A*B/C` means `(A*B)/C`, while `A=B=C` means `A=(B=C)`. (Can you see how the expression `A=B=C` might be useful, given that the value of `B=C` as an expression is the same as the value that is assigned to `B`?)

[Previous Section | Next Section | Chapter Index | Main Index]