Java Operators: Performing Operations on Primitive Data Types

Source http://www.informit.com/articles/article.asp?p=30868&rl=1

Operators work in conjunction with operands, or the literal values or variables involved in the operation. There are unary operators, which are operators that operate on a single operand, as well as operators that operate on two or more variables.

Arithmetic Operators

Arithmetic operators refer to the standard mathematical operators you learned in elementary school: addition, subtraction, multiplication, and division.

Addition

Addition, as you would expect, is accomplished using the plus sign (+) operator. The form of an addition operation is

```
operand + operand
```

For example:

```
// Add two literal values
int result = 5 + 5;

// Add two variables
int a = 5;
int b = 6;
int result = a + b;

// Add two variables and a literal
int result = a + b + 15;
```

An addition operation, can add two or more operands, whereas an operand can be a variable, a literal, or a constant.

Subtraction

Subtraction, again as, you would expect, is accomplished using the minus sign (–) operator. The form of a subtraction operation is

```
operand – operand
```

For example:

```
// Subtract a literal from a literal; the result is 5
int result = 10 - 5;

// Subtract a variable from another variable; the result is -1
int a = 5;
int b = 6;
int result = a - b;

// Subtract a variable and a literal from a variable
// The result is 5 – 6 – 15 = -1 – 15 = -16
int result = a - b - 15;
```

A subtraction operation can compute the difference between two or more operands, where an operand can be a variable, a literal, or a constant.

Multiplication

Multiplication is accomplished using the asterisk (*) operator. The form of a multiplication operation is

```
operand * operand
```

For example:

```
// Multiply two literal values; result is 25
int result = 5 * 5;

// Multiply two variables; result is 30
int a = 5;
int b = 6;
int result = a * b;

// Multiply two variables and a literal
// The result is 5 * 6 * 15 = 30 * 15 = 450
int result = a * b * 15;
```

A multiplication operation can multiply two or more operands, where an operand can be a variable, a literal, or a constant.

Division

Division is accomplished using the forward slash (/) operator. The form of a division operation is:

```
operand / operand
```

For example:

```
// Divide a literal by a literal; result is 5
int result = 10 / 2;

// Divide a variable by another variable; result is 3
int a = 15;
int b = 5;
int result = a / b;
```

When dividing integer types, the result is an integer type (see the previous chapter for the exact data type conversions for mathematical operations). This means that if you divide an integer unevenly by another integer, it returns the whole number part of the result; it does not perform any rounding. For example, consider the following two operations that both result to 1.

```
int result1 = 10 / 6; // Float value would be 1.6666
int result2 = 10 / 9; // Float value would be 1.1111
```

Both result1 and result2 resolve to be 1, even though result1 would typically resolve to 2 if you were rounding off the result. Therefore, be cognizant of the fact that integer division in Java results in only the whole number part of the result, any fractional part is dropped.

When dividing floating-point variables or values, this caution can be safely ignored. Floating-point division results in the correct result: The fractional part of the answer is represented in the floating-point variable.

```
float f = 10.0f / 6.0f; // result is 1.6666
double d = 10.0 / 9.0; // result is 1.1111
```

Note the appearance of the f following each literal value in the first line. When creating a floating-point literal value (a value that has a fractional element), the default assumption by the compiler is that the values are double. So, to explicitly tell the compiler that the value is a float and not a double, you can suffix the value with either a lowercase or uppercase F.

Modulus

If integer division results in dropping the remainder of the operation, what happens to it? For example if you divide 10 by 6:

```
int i = 10 / 6;
```

The Java result is 1, but the true result is 1 Remainder 4. What happened to the remainder 4?

Java provides a mechanism to get the remainder of a division operation through the modulus operator, denoted by the percent character (%). Although the previous example had a result of 1, the modulus of the operation would give you that missing 4. The form of a modulus operation is

```
operand % operand
```

For example:

```
int i = 10 / 6; // i = 1
int r = 10 % 6; // r = 4
```

Similar to the other arithmetic operators in this chapter, the modulus of an operation can be performed between variables, literals, and constants.

Increment and Decrement Operators

In computer programming it is quite common to want to increase or decrease the value of an integer type by 1. Because of this Java provides the increment and decrement operators that add 1 to a variable and subtract 1 from a variable, respectively. The increment operator is denoted by two plus signs (++), and the decrement operator is denoted by two minus signs (--). The form of the increment and decrement operators is

```
variable++;
++variable;
variable--;
--variable;
```

For example:

```
int i = 10;
i++; // New value of i is 11
```

You might notice that the variable could either be prefixed or suffixed by the increment or decrement operator. If the variable is always modified appropriately (either incremented by 1 or decremented by 1), then what is the difference? The difference has to do with the value of the variable that is returned for use in an operation.

Prefixing a variable with the increment or decrement operator performs the increment or decrement, and then returns the value to be used in an operation.

For example:

```
int i = 10;
int a = ++i; // Value of both i and a is 11
i = 10;
int b = 5 + --i; // Value of b is 14 (5 + 9) and i is 9
```

Suffixing a variable with the increment or decrement operator returns the value to be used in the operation, and then performs the increment or decrement. For example:

```
// Value of i is 11, but the value of a is 10
// Note that the assignment preceded the increment
int i = 10;
int a = i++;

// Value of i is 9 as before, but the value of b is 15 (5 + 10)
i = 10;
int b = 5 + i--;
```

Pay particular attention to your code when you use prefix and postfix versions of these operators and be sure that you completely understand the difference. That difference has led many programmers on a search for unexplained behavior in their testing!

Relational Operators

A very necessary part of computer programming is performing certain actions based off the value of a variable; for example if the nuclear reactor is about to blow up, then shut it down. The next chapter will speak at length about the mechanism for implementing this type of logic, but this section addresses the mechanism for comparing two variables through a set of relational operators.

Relational operators compare the values of two variables and return a boolean value. The general form of a relation operation is

```
LeftOperand RelationalOperator RightOperand
```

For example:

```
int a = 10;
int b = 10;
int c = 20;
boolean b1 = a == c; // false, 10 is not equal to 20
boolean b2 = a == b; // true, 10 is equal to 10
boolean b3 = a < c; // true, 10 is less than 20
boolean b4 = a < b; // false, 10 is not less than 10
boolean b5 = a <= b; // true, 10 is less than or equal to 10 (equal to)
boolean b6 = a != c; // true, 10 is not equal to 20
```

Bit-Wise Operators

The operators covered thus far in this chapter have been manipulating the interpreted values of these bits (for example, if a was an int with the value 10, a + 1 is equal to 11), but there are some things that can be done directly with those 1s and 0s. Java provides a set of bit-wise operators that looks at each bit in two variables, performs a comparison, and returns the result. The nature of the operator is defined by its truth table.

For the examples in this section, consider the following two bytes:

```
byte a = 10;
byte b = 6;
```

The bits for these values are

```
a = 0000 1010
b = 0000 0110
```

The bit-wise operators are going to define rules in the form of truth tables to apply to these two values for the purpose of building a result. The general form of a bit-wise operation is

```
result = operand
```

AND Operator

The AND operator specifies that both Signals A and B must be charged for the result to be charged. Therefore, AND-ing the bytes 10 and 6 results in 2, as follows:

```
a = 0000 1010 (10)
b = 0000 0110 (6)
  ---- ----
r = 0000 0010 (2)
```

OR Operator

The OR operator (|) defines a bit-wise comparison between two variables.

The OR operator specifies that the result is charged if either Signals A or B are charged. Therefore, OR-ing the bytes 10 and 6 results in 14, as follows:

```
a = 0000 1010 (10)
b = 0000 0110 (6)
  ---- ----
r = 0000 1110 (14)
```

Exclusive OR

The Exclusive OR (XOR) operator (^) defines a bit-wise comparison between two variables according to the truth table shown in Table 3.6.

The XOR operator specifies that the result is charged if Signal A or B is charged, but Signal A and B aren't both charged. It is called exclusive because it is charged only if one of the two is charged. Therefore, XOR-ing the bytes 10 and 6 results in 12, as follows:

```
a = 0000 1010 (10)
b = 0000 0110 (6)
  ---- ----
r = 0000 1100 (12)
```

NOT Operator

The NOT operator (~), also referred to as the bit-wise complement, flips the values of all the charges. If a bit is set to 1, it changes it to 0; if a bit is 0 it changes to 1.

The primary use for bit-wise operations originated in the data compression and communication applications. The problem was that you needed to either store or transmit the state of several different things. In the past, data storage was not as cheap as it is today and communication mechanisms were not over T3 or even a cable modem, but more like speeds of 1200 or 300 baud. To give you an idea about the difference in speed, cable modem and DSL companies tout that they achieve speeds about 20 times faster than 56K modems, but 56K modems are about 45 times faster than 1200-baud modems, and 180 times faster than 300-baud modems. So early communications were 3600 times slower than your cable modem. Thus, there was the need to compact data as much as possible!

If you have 8 different states to send to someone, you can simply assign a bit in a byte to each of the 8 states: 1 is defined to be true and 0 false (or on/off, and so on). Consider reporting the state of 8 different factory devices where devices 0 and 3 are active:

```
Device byte: 0000 1001 (9)
```

To determine whether device 3 is active, a Boolean expression can be determined with the following statement:

```
boolean is3Active = ( deviceByte & 8 ) == 8;
```

AND-ing the deviceByte (0000 1001) with the number 8 (0000 1000) returns 8, as the first bit is 0 (1 & 0 = 0), and using the equality operator it can be compared to 8 to see if that bit is set.

You should understand how bit-wise operators work (they are covered in Java certification exams) and, depending on what area of Java programming you delve into, you might use them in the future.

Logical Operators

Comparison operators enable you to compare two variables to determine whether they are equal or if one is greater than the other, and so on. But what happens when you want to check to see if a variable is in between a range of values? For example, consider validating that someone entered a correct value for an age field in your user interface. You might want to validate that the user is between the ages of 18 and 120; if someone claims to be 700 years old, you might want to check your calendar, prepare for rain, and see if anyone is building an ark! Furthermore, you might want to target a product to children and senior citizens, and therefore validate that the user's age is less than 8 or greater than 55.

To address this need, Java has provided a set of logical operators that enable you to make multiple comparisons and group the result into a single boolean value.

AND

The AND logical operator (&&) returns a true value if both of the variables it is comparing are true. The form of an AND operation is

```
boolean1 && boolean2
```

It compares two Boolean variables and returns true only if both of the Boolean variables are true. For example, consider verifying that someone is of age:

```
boolean isAdult = (age >= 18 ) && (age <= 120)
```

If the age is 20, then 20 is greater than 18 and less than 120, so isAdult is true. If the age is 17, then 17 is not greater than or equal to 18 although it is less than 120; both conditions are not satisfied, so isAdult is false.

OR

The OR logical operator (||) returns a true value if either of the variables it is comparing are true. The form of an OR operation is:

```
boolean1 && boolean2
```

It compares two Boolean variables and returns true if either of the Boolean variables are true. For example, consider verifying that someone is either a child or a senior citizen:

```
boolean isChildOrSenior = (age <= 10 ) || (age >= 55)
```

If the age is 7, then 7 is less than 18, so isChildOrSenior is true. If the age is 17, then 17 is not less than or equal to 10, and it is not greater than or equal to 55; neither condition is satisfied, so isChildOrSenior is false.

Not that this is an inclusive OR, not an exclusive OR; if either or both of the values are true, the result is true.

Shift Operators

After looking at the structure of memory and all the 1s and 0s with their binary assignment, you might notice something interesting. What happens if you move all the bits in your variable to the left? Consider the following:

```
0000 1010 = 8 + 2 = 10
0001 0100 = 16 + 4 = 20
0010 1000 = 32 + 8 = 40Now take that value and move it to the right:
```

```
0010 1000 = 32 + 8 = 40
0001 0100 = 16 + 4 = 20
0000 1010 = 8 + 2 = 10
0000 0101 = 4 + 1 = 5
0000 0010 = 2
```

The basic properties of the binary numbering system demonstrates that moving the bits to the left multiplies the value by two and moving the bits to the right divides the value by two. That is interesting and all, but what is the value in that?

Today video cards have high-performance, floating-point arithmetic chips built into them that can perform high-speed mathematical operations, but that was not always the case. In the early days of computer game programming, every ounce of performance had to be squeezed out of code to deliver a decent performing game. To understand just how much math is behind computer animation in game programming, consider a rudimentary flight simulator. From the cockpit we had to compute the location of all objects in sight and draw them. All the objects in the game were drawn with polygons or in some cases, triangles. Some objects in the game could have 50,000 triangles, which results in 150,000 lines, and to maintain 30 frames per second that requires 4,500,000 lines drawn per second. Whoa! How fast can you draw a line? If you can shave off 100 nanoseconds from the line drawing algorithm, that would result in a savings of

```
100ns * 4,500,000 lines = 450,000,000ns = .45 second
```

So, an increase of just 100 nanoseconds results in almost a half a second of savings per second! The bottom line is that every tiny bit of performance that can be improved can have dramatic results.

Okay, fine, now that you have a background on drawing lines, how does this relate to moving bits around?

It turns out that moving bits is exceptionally fast, whereas multiplication and division are very slow. Combine these two statements and what do you get? It would sure be better to make use of the fact that shifting bits can perform multiplication and division instead of using the multiplication and division operators!

That works great for multiplication by two, but how do you handle multiplications by other numbers?

It turns out that addition is also a very inexpensive operation, so through a combination of shifting bits and adding their results can substitute for multiplication in a much more efficient manner. Consider multiplying a value by 35: this is the same as shifting it 5 bits to the left (32), adding that to its value shifted 1 bit to the left (2), and adding that to its unshifted value (1). For more information on this peruse the selection of video game–programming books at your local bookstore.

Shift Left Operator

Shifting a value to the left (<<) results in multiplying the value by a power of two. The general form of a left shift operation is

```
value << number-of-bits
```

The number of bits specifies how many bits to shift the value over. Stated more simply shifting a value, n bits, is the same as multiplying it by 2n. For example:

```
int i=10;
int result = i << 2; // result = 40
```

Shift Right

Shifting a value to the right (>>) is the same as dividing it by a power of two. The general form of a right shift operation is

```
value >> number-of-bits
```

For example:

```
int i = 20;
int result = i >> 2; // result = 5
int j = -20;
int result2 = j >> 2; // result = -5Shift Right (Fill with 0s)
```

Shifting a value to the right has the effect of dividing a value by a power of two, and as you just saw, it preserved the sign of the negative number. Recall that this highest bit in each of Java's numeric data types specified the sign. So, if a right shift operator truly did shift the bits, it would move the sign bit to the right, and hence the number would not be equivalent to a division by a power of 2. The right shift operator performs the function that you would expect it to, but does not actually do a true shift of the value.

If your intent is not to perform division but to perform some true bit operations, this side effect is not desirable. To address this, Java has provided a second version of the right shift operator (>>>) that shifts the bits to the right and fills the new bits with zero.

Therefore, the right-shift operator (>>>) can never result in a negative number because the sign bit will always be filled with a zero.

Operator Precedence

There are all these operators and the good news is that they can all be used together. Consider multiplying a value by 10 and adding 5 to it:

```
int result = a * 10 + 5;
```

But what happens when you want to add 5 to a value, and then multiply it by 10? Does the following work?

```
int result = a + 5 * 10;
```

Consider for a moment that in the compiler it performs its calculations left to right, now how would you add 5 multiplied by 10 to a?

```
int result = 5 * 10 + a;
```

This is starting to get confusing and complicated! Furthermore it is not intuitive! When you study mathematics, you learn that certain operations are reflective, meaning that they can be performed in any order and offer the same result. Thus, the following two statements are equivalent:

```
int result = a + 5 * 10;
int result = 5 * 10 + a;
```

The confusion between these two statements is what operation to perform first; the multiplication or the division? How would you solve this in your math classes? You would simply instrument the statements with parentheses to eliminate the confusion:

```
int result = ( a + 5 ) * 10;
int result = ( 5 * 10 ) + a;
```

These statements are now read as follows: add 5 to a, and then multiply the result by 10; and multiply 5 by 10, and then add a to the result, respectively. In mathematics, the operation enclosed in parentheses is completed before any other operation.

The same mechanism can be implemented in Java, and it is! To qualify what operations to perform first, you can eliminate all ambiguity by explicitly using parentheses to denote what operations are grouped together. But that does not solve the problem of how to handle a statement without parentheses. What is the result of the following operation?

```
int result = 8 + 5 * 10;Is the result 130 or 58?
```

The answer is that there needs to be a set of rules that defines the order of operation execution. This is defined by what is called the operator precedence. The operator precedence defines what operators take precedence over other operators, and hence get executed first. All programming languages define an operator precedence, which is very similar between programming languages, and you must be familiar with it.

Java Operator Precedence

Unary operators are those operators that operate on a single variable, such as increment and decrement (++), positive and negative signs (+ –), the bit-wise NOT operator (~), the logical NOT operator (!), parentheses, and the new operator (to be discussed later).

Arithmetic operators are those operators used in mathematical operations. Here it is important to note that this table is read from left to right, therefore multiplication and division have greater precedence than addition and subtraction. Thus the answer to the aforementioned question:

```
int result = 8 + 5 * 10;
```

The result is 58; multiplication has a higher precedence than addition, so the multiplication is performed first followed by the addition. So, the compiler reads this as multiply 5 by 10 (50) and add 8 to the result (50 + 8 = 58).

Shift operators refer to the bit-wise shift left, shift right, and shift right and fill with zeros operators.

The logical comparison operators follow with the familiar greater than, less than, and equality variations. Comparison operators return a boolean value, so there is one additional operator added: instanceof; this operator will be addressed later when you have a little more Java under your belt.

Next are the bitwise AND, OR, and XOR operators followed by the logical AND and OR (&& ||), referred to as the short-circuit operators.

Next is a new category of operators in the ternary operators; the sole operator in this category is referred to as ternary because it uses three operands when computing its result. It is the following form:

```
a ? b : c;
```

This statement is read as follows: If a is true, then the result of this operation is b, otherwise the result is c. The ternary operator does not have to be comprised of single values, the only requirement is that the first value or operation resolves to be a boolean. Consider the following examples:

```
int result = ( 5 > 3 ) ? 2 : 1; // result is 2
int result = ( 5 < 3 ) ? 2 : 1; // result is 1
```

In the first example 5 is greater than 3, therefore the result is 2; in the second example 5 is not less than 3, so the result is 1. So more clearly written, the form of the ternary operator is:

```
(boolean expression) ? (return if true) : (return if false)
```

The ternary operator is rarely used, and is mainly inherited from Java's initial syntactical base from C/C++. It is a somewhat cryptic shortcut, but is perfectly legal, so be sure to understand how it is used.

The final sets of operators in the operator precedence hierarchy are the assignment operators. The assignment operators include the familiar assignment (=) operator as well as a set of additional assignment operators referred to generically as op= (operator equal). These new operators are shortcut operators used when performing an operation on a variable and assigning the result back to that variable. Consider adding 5 to the variable a; this could be accomplished traditionally as follows:

```
a = a + 5;
```

Because this is such a common operation Java provides a shortcut for it:

```
a += 5;
```

This is read: a plus equal 5, or explicitly a equals a plus 5. The operator equal operator can be applied to all the arithmetic, shift, and bit-wise operators.

Finally, whenever there is ambiguity or you desire a higher degree of readability, you can use parentheses to explicitly qualify the operator precedence yourself.

*~~~ End of Article ~~~*