Foundations of Java for ABAP Programmers

Alistair Rooney

Apress®

Foundations of Java for ABAP Programmers

Copyright © 2006 by Alistair Rooney

ISBN-13: 978-1-59059-625-8

ISBN-10: 1-59059-625-0

Library of Congress Cataloging-in-Publication data is available upon request.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewers: Gene Ames, Stefan Keuker
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editor: Andy Carroll
Assistant Production Director: Kari Brooks-Copony
Production Editor: Janet Vail
Compositor: Dina Quan
Proofreader: April Eddy
Indexer: Toma Mulligan/Book Indexers
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

LESSON 6

The Java Operators

Java operators can be broken down into several different groups:

- Arithmetic operators
- Relational operators
- Increment operators
- Logical operators

We'll look at each of these types in this lesson, along with the concept of block scope.

Arithmetic Operators

The most important thing to remember about arithmetic operators is their *order of precedence*. Let me give you an example:

```
2 + 3 * 5 + 4 = ?
```

If you answered 21, you are correct. If you said 54, you made the mistake of adding the numbers *before* multiplying them. At school we learned about BODMAS, which stands for Brackets, Of, Division, Multiplication, Addition, and Subtraction. Notice how adding and subtracting are the last things you do?

Remember Alistair's golden rule and you'll be OK: *If in doubt, use brackets!* So the previous equation would be better expressed like this:

```
2 + (3 * 5) + 4 = 21
```

These are the math operators in Java:

+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus

Be careful with the minus sign! It can also be used to denote a negative number. It is a unary operator (meaning it has only one operand).

Relational Operators

Relational operators are usually used when testing the relationship between two variables.

There are two big points to remember about relational operators:

- The result of these operations will always return a `true` or `false` boolean value.
- The equals operator is the double equals sign (`==`) *not* just a single one (`=`) as in ABAP.

You should be familiar with just about all of the relational operators already:

`<`	Less than
`>`	Greater than
`==`	Equal to
`!=`	Not equal to
`<=`	Less than or equal to
`>=`	Greater than or equal to

Here's a code example:

```
if (appleCount >= bananaCount)
{
    banana.eat();
}
```

We'll discuss these operators in more detail when we get to the topic of control flow in Lesson 8. Let's move on shall we?

Increment Operators

These are what I call "shorthand" operators:

- The code `x = x+1` can be shortened to `x++`.
- The code `y = y-1` can be shortened to `y--`.

Caution Here's a programmers' joke (but it's not really that funny): Java is the result of the operation C++, since C is only incremented after the operation.

The code for a loop (which we'll cover in Lesson 8) could be like this:

```
while(x<10)
{
    x++;
    System.out.println("The value of x is  "+x);
}
```

The loop will continue until x equals 10. Notice how x increments each time, as shown in the following output:

```
The value of x is 0
The value of x is 1
The value of x is 2
The value of x is 3
The value of x is 4
The value of x is 5
The value of x is 6
The value of x is 7
The value of x is 8
The value of x is 9
```

Get it?

Now that you have, I'll confuse the issue further. `++x` has a slightly different meaning than `x++`. When the plusses come before the variable name, this tells the interpreter to increment the variable *before* using it.

This means that if we had said `x = 3 * j++` and `j = 5`, then `x` would equal 15. However if we had said `x = 3* ++j`, then `x` would be 18. In other words, in this last case `j` would already have been incremented to 6 before it was used.

Don't forget this:

- `++h` will increment `h` *before* use.
- `h++` will increment `h` *after* use.

Now look at the joke again! It's still not funny. Let's move on to the next set of operators.

Logical Operators

There are three logical operators. As ABAP programmers you will know these as `AND`, `OR`, and `NOT`. In Java these are the `&&` operator, the `||` operator, and the `!` operator. The first two are the same as in ABAP, but the third has a slightly different meaning. It can negate any Boolean expression, and in doing so, it can be very handy. Commonly these operators are used in `if` and `while` statements.

Here's an example:

```
while(x!=10)
{
    x++;
    System.out.println("The value of x is  "+x);
}
```

And here's another:

```
if((a==7)&&(x!=10))
{
    do something . . .
}
```

The first example resolves to "while x is NOT equal to 10." The second one says "if a equals 7 AND x is not equal to 10."

As an ABAP programmer, you should be comfortable with nesting Boolean expressions, so I won't bore you with explanations of these basic principles. However the next section will introduce a new concept: bit manipulation.

Bitwise Operators

There are 8 bits in a byte, and the number 15, for example, is represented as 1111 in binary. Table 6-1 shows it diagrammatically. The sign in our example is positive. If the sign was negative, the leftmost bit would be a 1.

Table 6-1. *Eight Bits Showing the Value 15*

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	1	1	1	1

■ **Note** Negative figures are held in *two's complement* on any computer. Explaining two's complement is not for this course, but feel free to research this if you have the time.

We've digressed a little here. Let's return to our logical operators:

&	ANDs two binary numbers (typically used for masking)
|	ORs two binary numbers (typically used to set bits)
^	XORs a binary number
~	Converts a binary to one's complement
>>	Shifts bits right
<<	Shift bits left
>>>	Shift bits right (logical)

If we were to AND our example value of 15 with 5, we could express this in Java as follows:

```
byte a = (byte) 0x0e;   // 15 in hex
byte b = (byte) 0x05;   // 5 in hex
byte c = (byte)a&b;     // the two are ANDed resulting in 5
```

Table 6-2 shows this diagrammatically. Remember your Boolean logic? *Both* bits must be 1 before the result can be 1. So we have effectively *masked* bits 1 and 3.

Table 6-2. *ANDing 15 and 5*

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	1	1	1	1
0	0	0	0	0	1	0	1
0	0	0	0	0	1	0	1

If we were to OR the same values, we would have the following:

```
byte a = (byte) 0x0e;    // 15 in hex
byte b = (byte) 0x05;    // 5 in hex
byte c = (byte)a|b;      // the 2 are ANDed resulting in 15
```

Table 6-3 shows the diagram. Again Boolean logic dictates that *either* bit can be 1 for the result to be 1. We have *set* bits 1 and 3 in this example.

Table 6-3. *ORing 15 and 5*

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	1	1	1	1
0	0	0	0	0	1	0	1
0	0	0	0	1	1	1	1

To XOR the bits, we use the caret sign (`^`). This is very useful when working at a bit level, but rarely used in workaday Java. XOR demands that either of the bits may be 1, but not both, to achieve a result of 1.

Shifting operations can also be extremely useful. Let's examine them.

The first is the shift right (`>>`). We'll use our value of 15 again and shift the bits two to the right:

```
byte a = (byte) 0x0e;     // 15 in hex
byte c = (byte)(a>>2);    // Shifted 2 to the right giving 3
```

Once again, Table 6-4 shows the diagram version.

Table 6-4. *Shifting Bits Two to the Right*

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	1	1	1	1
0	0	0	0	0	0	1	1

You have to be careful with negative numbers, though. Bit 7 is the "sign" bit. For negative numbers, bit 7 would be a 1, and each shift would move a 1 into the leftmost position. The example shown in Table 6-5 demonstrates this. (Remember that negative numbers are held as two's complement in Java. The example in Table 6-5 is not –15.)

Table 6-5. *Shifting Bits Two to the Right in a Negative Number*

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	0	0	0	1	1	1	1
1	1	1	0	0	0	1	1

To *force* the zero to be fed in on the left, regardless of the sign bit, we use the >>> operator.
Shifting left is easier, as the bits from the right are always filled with zeroes:

```
byte a = (byte) 0x0e;      // 15 in hex
byte c = (byte)(a<<2);     // Shifted 2 to the left, giving 60
```

Table 6-6 shows this diagrammatically.

Table 6-6. *Shifting Bits Two to the Left*

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	1	1	1	1
0	0	1	1	1	1	0	0

That's all there is for operators!

Block Scope

Conceptually, block scope is a very simple topic. There are just a few basic rules to be followed. First, though, a definition: A *block* is any section of code contained in curly brackets {}. Now for the rules.

Rule number 1: If a variable is defined inside a block, it is not visible to any code outside of that block.

In Java, the following is correctly expressed:

```
{
    int x = 5;
    int y = x + 5;
    System.out.println("x = "+x+" and y = "+y);
}
```

However, the following would produce an error, since the variables are *out of scope*:

```
{
    int x = 5;
    int y = x + 5;
}
System.out.println("x = "+x+" and y = "+y);
```

Rule number 2: If a variable is defined outside a block, it is visible to any code inside that block.

The following Java example would work just fine:

```
int x = 5;
int y = x + 5;
{
    System.out.println("x = "+x+" and y = "+y);
}
```

Easy enough isn't it? In the next lesson we're going to explore the wonders of strings!