Bitwise Operators In Java

Melvyn Ian Drag

October 28, 2020

Abstract

Of course you know things like "+" and "-" and "/" and "*" in Java. And you know all about how Java programs are structured - i.e. public static void main blah blah. Tonight we will focus on 5 very important operators that you likely do not know, but which are used in many Java programs. These are the set of so called "bitwise operators" that allow you to fiddle with individual bits instead of whole values like int, char, etc..

1 Everything is Made of Bits

Do not forget that int, double, float, String, etc.. are all composed of bits. An int in java, for example, takes up 4 bytes, a.k.a 32 bits. A long takes up 8bytes - 64 bits in other words. A bit is a one or a zero. Do not forget that.

2 Writing ints with Hex Notation

A Java int is 4 bytes. We saw last week you can write an int literal in hex. For example:

```
int x = 0x000000FF; // equals 255
```

You don't have to specify all 8 characters in the hex string. For example, you can put two:

```
int y = 0xFF; // equals 0x000000FF, equals 255. Java fills in the
   missing hex characters with 0 until you get the full 4 bytes'
   worth.
```

2.1 There's a Bunch I don't know About Java

As I sit here writing these notes, I don't know if Java will allow you to specify and uneven number of hex characters in your literal. I've always written 2, 4, 6, or 8. I don't know if this is valid:

```
int foo = 0xF; // Must go read the Java documentation to see if
   this is valid.
```

Similarly I don't know what happens if you write 9 hex characters instead of 8. Will the compiler give an error? Will it truncate the number? I have no idea - but a quick bit of googling would give the answer.

```
int bar = 0x00FF1A1D3; // this has 9 hex chars: 0,0, f, f, 1, a, 1,
   d, 3
```

I could have found out the answer, but it's irrelevant to us right now what happens. We must go on to the next topic.

3 Bitwise AND

A common thing to do in computer programs is line up two integers and compute the bitwise AND. **Bitwise AND of 2 binary numbers lines up the bits and returns 1 where they are both 1, and 0 otherwise**

Example:

```
   11110010
AND 10010011
----------------
   10010010
```

The Java operator for AND is "&". So if we AND two Java ints:

```
    int x = 0xFFFF;            // 00000000 00000000 11111111 11111111
    int y = 0xFFFF111A;        // 11111111 11111111 00010001 00011010
                               // -----------------------------------
    int z = x & y;             // 00000000 00000000 00010001 00011010
    System.out.println(z);  // Convert the above to decimal: 4378
```

You can verify the above in the program *FirstBitwiseAnd.java* It works! Feel free to try a few more examples until you are comfortable with this.

4 Bitwise OR

Bitwise OR lines up the bits just like AND does, but returns one if either bit is 1, and 0 other wise.

Example:

```
   11110010
OR  10010011
----------------
   11110011
```

Let's have another look at some Java ints. Note that the Java operator for OR is "|".

```
    int x = 0x10FF;            // 00000000 00000000 00010000 11111111
    int y = 0x01FF111A;        // 00000001 11111111 00010001 00011010
                               // -----------------------------------
    int z = x | y;             // 00000001 11111111 00010001 11111111
    System.out.println(z);  // Convert the above to decimal:
       33493503
```

You can verify the above in the program *FirstBitwiseOr.java* It works! Feel free to try a few more examples until you are comfortable with this.

5 Left Bit Shift

Another common thing to do with a bunch of bits is shift them left. If you shift a binary number left, add a zero on the right and remove the leftmost bit. In Java we use the "<<" operator to shift left.

Let's consider an example:

```
00101100 Shift this right 5
001011000000
     10000000
```

When we shift to the left 5 times, we add 5 zero bits to the right. The 5 bits farthest left disappear. (In class make sure to demonstrate this by way of blocks on a shelf).

Consider this short Java example:

```
int y = 0x0000FFFF;
System.out.println(y << 8 ); // output is 16776960
```

In this example we take a 4 byte integer and shift it left 8 bits. So, we add 8 zeros on the right and drop 8 bits from the left.

```
00000000 00000000 11111111 11111111
00000000 00000000 11111111 11111111 00000000
         00000000 11111111 11111111 00000000
```

The result? 0x00FFFF00 aka 16776960. This example and one other is in *FirstLeftShift.java*

6 Right Bit Shift

If the number you are shifting in positive, the right shift operator adds zeros on the left and drops bits from the right. The case where the number you shift is negative is going to be covered next week. So right shifting is just like left shifting, but instead of adding zeros on the right, you add zeros on the left. Have a look at FirstRightShift.java.

6.1 Right bit shift is tricky!

What do you get in Java when you do this: 0xffffffff >>8? You would expect to get 0x00ffffff, but that's not the case. You stay with 0xffffffff. There is another interesting operator in Java: >>>. We'll learn about this one next week. You need to understand a bit about how negative numbers work in Java, and you need to understand the difference between signed and unsigned datatypes. Java has only signed datatypes :(