Crash Course to Binary Math in Java

What's the difference between `long`, `int`, `short`, `char`, and `byte`? The amount of size allocated to variables of each type. Specifically:

- a `long` represents a signed 64-bit value,
- an `int` represents a signed 32-bit value,
- a `short` and a `char` represent signed 16-bit values, and
- a `byte` represents a signed 8-bit value.

Java uses the leading (left-most) bit of such numbers to represent the sign. Specifically, a 0 means "positive", and a 1 means "negative." The remaining bits determine the actual number. Because of the differing numbers of bits between these types, this limits the maximum and minimum values that can be stored in each type. Specifically:

- a `long` represents an integer in the range [-9,223,372,036,854,775,808 - 9,223,372,036,854,775,807],
- an `int` represents an integer in the range [-2,147,483,648 - 2,147,483,647],
- both `short` and `char` represent integers in the range [-32,768 - 32,767], and
- a `byte` represents an integer in the range [-128 - 127].

There is a pattern here--a type that represents an n-bit binary integer can represent any integer in the range [$-(2^{n-1})$ - $2^{n-1} – 1$].

One special property of binary numbers relative to their usage in computers is that there are extra numerical operations we can perform on them. We will call them AND, OR, XOR, COMPLEMENT, LSHIFT, RSHIFT, and URSHIFT.

Before we describe how the operations work, there is one caveat we must address: Java doesn't allow you to declare numbers directly in binary. The next-best-thing is to declare them in hexadecimal. Since hexadecimal uses 16 distinct numerals to encode a number, one numeral in hexadecimal can be expressed by exactly four bits in binary. Here's the encryption:

Hexadecimal	**Binary**	**Decimal**
0x0	0000	0
0x1	0001	1
0x2	0010	2
0x3	0011	3
0x4	0100	4
0x5	0101	5
0x6	0110	6

0x7	0111	7
0x8	1000	8
0x9	1001	9
0xA	1010	10
0xB	1011	11
0xC	1100	12
0xD	1101	13
0xE	1110	14
0xF	1111	15

AND

Let a = 0001101100111101 and let b = 0011110000111111. We use the ampersand character "&" in Java to represent an AND. We define "a & b" as follows:

```
   0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1
 & 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1
 ---------------------------------
   0 0 0 1 1 0 0 0 0 0 1 1 1 1 0 1
```

Generally, for each corresponding bit in numbers a and b, the corresponding bit in "a & b" will be 1 if and only if both corresponding bits in a and b are 1. Otherwise, the corresponding bit in "a & b" will be 0. Here's how we do it in Java:

```
 short a = 0x1B3D;  // hexadecimal representation of a
 short b = 0x3C3F;  // hexadecimal representation of b
 short c = a & b;   // c is 0x183D
```

OR

We use the pipe character "|" in Java to represent an OR. Using a and b as before, we define "a | b" as follows:

```
   0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1
 | 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1
 ---------------------------------
   0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1
```

Generally, for each corresponding bit in numbers a and b, the corresponding bit in "a | c" will be 1 if at least one of the bits is 1, or 0 if they are both 0. Here's how to do it in Java:

```
 short a = 0x1B3D;
 short b = 0x3C3F;
 short c = a | c;  // c is 0x3F3F
```

XOR

We use the caret character "^" in Java to represent an XOR. Using a and b as before, we define "a ^ b" as follows:

```
   0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1
 ^ 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1
 ---------------------------------
   0 0 1 0 0 1 1 1 0 0 0 0 0 0 1 0
```

Generally, XOR works like OR, with the exception that 1 XOR 1 is 0 (with OR, 1 OR 1 = 1). XOR literally means "eXclusive OR", meaning that only one of the corresponding bits in a and b can be 1 for the corresponding bit in "a ^ b" to be 1.
In Java, this is:

```
 short a = 0x1B3D;
 short b = 0x3C3F;
 short c = a ^ b;  // c is 0x3702
```

COMPLEMENT

We use the tilde character "~" in Java to represent a COMPLEMENT. This operation is unary--you use it only one number. Here's an example:

```
 ~(0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1) == 1 1 1 0 0 1 0 0 1 1 0 0 0 0 1
0
```

The COMPLEMENT operator simply flips the value of each bit in the operand.
In Java, this is:

```
 short a = 0x1B3D;
 short c = ~a;  // c is 0xE4C2
```

LSHIFT

We use two left angle-brackets "<<" to mean LSHIFT. LSHIFT means "Left SHIFT"--we shift all bits in a binary number to the left a certain number of spaces. The vacant spaces are filled with 0's. Here's an example:

```
 (0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1) << 1  == 0 0 1 1 0 1 1 0 0 1 1 1
1 0 1 0  // shift left by 1 space
 (0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1) << 3  == 1 1 0 1 1 0 0 1 1 1 1 0
1 0 0 0  // shift left by 3 spaces
 (0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1) << 7  == 1 0 0 1 1 1 1 0 1 0 0 0
0 0 0 0  // shift left by 7 spaces
 (0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1) << 15 == 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0  // shift left by 15 spaces
```

In Java, this is:

```
  short a = 0x1B3D;
  short c = a << 1;
  short d = a << 3;
  short e = a << 7;
  short f = a << 15;
```

There are some caveats here. First, you can inadvertently change the sign of a number with LSHIFT. Also, if you want to deal with a type other than `int`, you may need to explicitly cast to that type (e.g. you can't declare a as 0x9B3D, because 0x9B3D to Java is an `int` and it will complain that you are trying to declare a as `short`). Also, be careful when using LSHIFT as part of a larger calculation when dealing with types other than `int`--Java will attempt to promote the result of a LSHIFT operation to `int` when dealing with `char`, `byte`, and `short`.

<u>RSHIFT</u>

We use two right angle-brackets ">>" to mean RSHIFT. RSHIFT means "Right SHIFT"--we shift all bits in a binary number to the right by a certain number of spaces, and fill the vacant spaces on the left of the number with 0's. RSHIFT is tricky--it will preserve the sign of the number when you shift. This means that if you perform an RSHIFT on a negative number, the result will still have the sign bit set to 1.
Examples:
```
 (0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1) >> 1  == 0 0 0 0 1 1 0 1 1 0 0 1
1 1 1 0  // shift right by 1 space
 (0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1) >> 3  == 0 0 0 0 0 0 1 1 0 1 1 0
0 1 1 1  // shift right by 3 spaces
 (1 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1) >> 3  == 1 0 0 0 0 0 1 1 0 1 1 0
0 1 1 1  // shift right by 3 spaces, but preserve negative-ness
 (1 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1) >> 15 == 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0  // shift right by 15 spaces, but preserve negative-ness
```
In Java, this is:
```
 short a = 0x1B3D;
 short b = 0x9B3D;
 short c = a >> 1;
 short d = a >> 3;
 short e = b >> 3;
 short f = b >> 15;
```

<u>URSHIFT</u>

We use three right angle-brackets ">>>" to mean URSHIFT. URSHIFT is like RSHIFT, but it will not preserve the sign bit (e.g. it will fill it with 0). URSHIFT stands for "Unsigned Right SHIFT". This is the only bit operation in Java that is NOT found in C.

Testing and Setting Bits

Setting a Bit to 1

Using the aforementioned bit operations, it becomes trivial to set a bit to 0 or 1. Suppose we wanted to set the nth bit from the right in a number a to 1 (this act is simply referred to "setting" the bit). Here's how to do this in Java:
```
 a = a | (1 << n);
```
Or, more concisely,

```
 a |= (1 << n);
```

The aforementioned lines will set the nth bit from the right (henceforth known as simply "the nth bit"). For example, if n is 0, this sets the rightmost bit; if n is 1, this sets the second-rightmost bit; if n is 2, this sets the third-rightmost bit, etc. Note that the aforementioned lines set the bit in the nth place to 1 regardless of whether or not it was already set. This means that we start counting from 0 instead of 1.
Here's an example if a = 0x1B3D and n = 6:

```
   0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1
 | 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0  // 1 << 6
 ---------------------------------
   0 0 0 1 1 0 1 1 0 1 1 1 1 1 0 1  // now the 6th bit is set to 1.
```

There is a caveat here: The "6th" bit really refers to the 7th position. This is because we start counting bits with the "0th" bit, which is the 1st position.

Setting a Bit to 0

To set the nth bit to 0 (this act is also called "unsetting the nth bit"), we use this Java code:

```
 a = a & ~(1 << n);
```

Or, more concisely,

```
 a &= ~(1 << n);
```

Note that this will set the nth bit to 0 regardless of whether or not it is already 0. Here's an example where a = 0x1B3D and n = 5:

```
   0 0 0 1 1 0 1 1 0 0 1 1 1 1 0 1
 & 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1  // ~(1 << 5)
 ---------------------------------
   0 0 0 1 1 0 1 1 0 0 0 1 1 1 0 1  // now, the 5th bit is set to 0
```

Testing a Bit

Now, suppose we wanted to determine whether or not the nth bit is 0 or 1. Here's how we would do it in Java:

```
 if( (a & (1 << n)) != 0 ) {
    // the bit is set
 }
 else {
    // the bit is NOT set
 }
```

Some Fun

Finally, we give you this slightly-amusing method that turns an integer into a string of 0's and 1's (for your entertainment):

```
 String intToBinary( int theInt ) {
    String ret = "";

    // test each bit (all 32 of them)
    for( int i = 31; i >= 0; i-- ) {
        if( (theInt & (1 << i)) != 0 ) {
            ret += "1";   // the ith bit is set
        } else {
            ret += "0";   // the ith bit is NOT set
        }
     }

     return ret;
 }
```