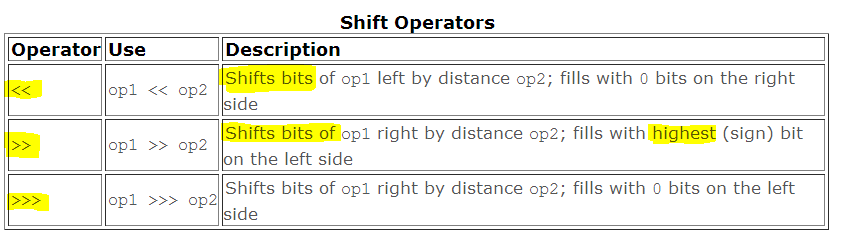
Operators in Java

<http://docstore.mik.ua/orelly/java/langref/ch04_07.htm>



**Left Shift Operator <<**

The left shift operator << produces a pure value that is its left operand left-shifted by the number of bits specified by its right operand. The << operator may appear in a shift expression.

(3<<2) == 12

(-3<<2) == -12

(0x01234567<<4) == 0x12345670

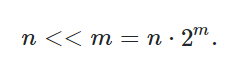
(0xF1234567<<4) == 0x12345670

If the converted type of the left operand is int, only the five least significant bits of the value of the right operand are used as the shift distance. Therefore, the shift distance is in the range 0 through 31. In this case, the value produced by r << s is mathematically equivalent to:

http://docstore.mik.ua/orelly/java/langref/figs/eq4_2.gif **int** ---- r << s = http://docstore.mik.ua/orelly/java/langref/figs/eq4_2.gif , 3 << 2 = 3 \* 2 2 MOD 32 = 3 \* 22 = 3 \* 4 = 12

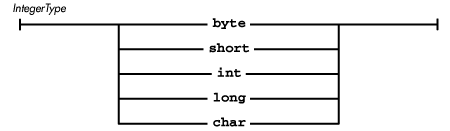
If the type of the left operand is long, only the six least significant bits of the value of the right operand are used as the shift distance. Therefore, the shift distance is in the range 0 through 63. In this case, the value produced by r << s is mathematically equivalent to:

http://docstore.mik.ua/orelly/java/langref/figs/eq4_3.gif ---- **long**



#### Integer types

Java provides integer data types in a variety of sizes. Unlike C/C++, however, the sizes of these types are part of the language specification; they are not platform-dependent. Formally:



 The representation shown is used on all platforms and is independent of the native platform architecture.

|  |  |  |
| --- | --- | --- |
| Table 3.1: Integer Types and Their Representations | | |
| **Type** | **Representation** | **Range** |
| byte | 8-bit, signed, two's complement | -128 to 127 |
| short | 16-bit, signed, two's complement | -32768 to 32767 |
| int | 32-bit, signed, two's complement | -2147483648 to 2147483647 |
| long | 64-bit, signed, two's complement | -9223372036854775808 to 9223372036854775807 |
| char | 16-bit, unsigned, Unicode | '\u0000' to '\uffff' |

All of the signed integer types in Java use a two's complement representation.

### Right Shift Operator >>

### 

The right shift operator >> produces a pure value that is its left operand right-shifted with sign extension by the number of bits specified by its right operand. Right-shifting with sign extension means that shifting a value n places to the right causes the n high order bits to contain the same value as the sign bit of the unshifted value. The >> operator may appear as part of a shift expression.

### Unsigned Right Shift Operator >>>

The unsigned right shift operator >>> produces a pure value that is its left operand right-shifted with zero extension by the number of bits specified by its right operand. Right-shifting with zero extension means that shifting a value n places to the right causes the n high order bits to contain zero.

If the converted type of the left operand is int, only the five least significant bits of the value of the right operand are used as the shift distance. Therefore, the shift distance is in the range 0 through 31.

In this case, the value produced by r >> s is mathematically equivalent to:

http://docstore.mik.ua/orelly/java/langref/figs/eq4_4.gif

The notation http://docstore.mik.ua/orelly/java/langref/figs/floorx.gif means the greatest integer less than or equal to *x* ; this is called the floor operation.

If the type of the left operand is long, only the six least significant bits of the value of the right operand are used as the shift distance. Therefore, the shift distance is in the range 0 through 63. In this case, the value produced by r >> s is mathematically equivalent to:

http://docstore.mik.ua/orelly/java/langref/figs/eq4_5.gif

<http://javarevisited.blogspot.in/2015/02/difference-between-right-shift-and.html>

Right shift ">>" keeps the sign extension while shifting bit patterns, but right shift without sign doesn't keep the original sign bit intact, it fills with zero. **Which means after using ">>>" a negative number can turned into positive number**. For positive inputs, both signed and unsigned right shift will produce same result but for negative numbers they will produce different result. **Also remember, ">>" is equal to divide by 2** e.g. ">> 1" will divide number by two, ">>2" will divide number twice by two e.g. by 4. It is well known fast way to divide a number by two in Java.  
  
<https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators>

### >>> (Zero-fill right shift)

This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left. The sign bit becomes 0, so the result is always non-negative.

For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result. For example, 9 >>> 2 yields 2, the same as 9 >> 2:

. 9 (base 10): 00000000000000000000000000001001 (base 2)

--------------------------------

9 >>> 2 (base 10): 00000000000000000000000000000010 (base 2) = 2 (base 10)

However, this is not the case for negative numbers. For example, -9 >>> 2 yields 1073741821, which is different than -9 >> 2 (which yields -3):

. -9 (base 10): 11111111111111111111111111110111 (base 2)

--------------------------------

-9 >>> 2 (base 10): 00111111111111111111111111111101 (base 2) = 1073741821 (base 10)

# [**What is the purpose of the unsigned right shift operator “>>>” in Java?**](http://stackoverflow.com/questions/16763917/what-is-the-purpose-of-the-unsigned-right-shift-operator-in-java)

<http://stackoverflow.com/questions/16763917/what-is-the-purpose-of-the-unsigned-right-shift-operator-in-java>

The >>> operator lets you treat int and long as 32- and 64-bit unsigned integral types, which are missing from the Java language.

This is useful when you shift something that does not represent a numeric value. For example, you could represent a black and white bit map image using 32-bit ints, where each int encodes 32 pixels on the screen. If you need to scroll the image to the right, you would prefer the bits on the left of an int to become zeros, so that you could easily put the bits from the adjacent ints:

>>> is also the safe and efficient way of finding the rounded mean of two (large) integers:

int mid = (low + high) >>> 1;

If integers high and low are close to the the largest machine integer, the above will be correct but

int mid = (low + high) / 2;

can get a wrong result because of overflow.

A normal right shift >> of a negative number will keep it negative. I.e. the sign bit will be retained.

An **unsigned** right shift >>> will shift the sign bit too, replacing it with a zero bit.

There is no need to have the equivalent left shift because there is only one sign bit and it is the leftmost bit so it only interferes when shifting right.

Essentially, the difference is that one preserves the sign bit, the other shifts in zeros to replace the sign bit.

**XOR ^ in Java**

<http://stackoverflow.com/questions/1991380/what-does-the-operator-do-in-java>

^ in Java is the exclusive-or ("xor") operator.

Let's take 5^6 as example:

(decimal) (binary)

5 = 101

6 = 110

------------------ xor

3 = 011

This the truth table for bitwise ([JLS 15.22.1](http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.22.1)) and logical ([JLS 15.22.2](http://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.22.2)) xor:

^ | 0 1 ^ | F T

--+----- --+-----

0 | 0 1 F | F T

1 | 1 0 T | T F

**More simply, you can also think of xor as "this or that, but not both!".**

# [**Real world use cases of bitwise operators**](http://stackoverflow.com/questions/2096916/real-world-use-cases-of-bitwise-operators)

<http://stackoverflow.com/questions/2096916/real-world-use-cases-of-bitwise-operators>

* **Bit fields (flags)**  
  They're the most efficient way of representing something whose state is defined by several "yes or no" properties. ACLs are a good example; if you have let's say 4 discrete permissions (read, write, execute, change policy), it's better to store this in 1 byte rather than waste 4. These can be mapped to enumeration types in many languages for added convenience.
* **Communication over ports/sockets**  
  Always involves checksums, parity, stop bits, flow control algorithms, and so on, which usually depend on the logic values of individual bytes as opposed to numeric values, since the medium may only be capable of transmitting one bit at a time.
* **Compression, Encryption**  
  Both of these are heavily dependent on bitwise algorithms. Look at the [deflate](http://en.wikipedia.org/wiki/DEFLATE) algorithm for an example - everything is in bits, not bytes.
* **Finite State Machines**  
  I'm speaking primarily of the kind embedded in some piece of hardware, although they can be found in software too. These are combinatorial in nature - they might literally be getting "compiled" down to a bunch of logic gates, so they have to be expressed as AND, OR, NOT, etc.
* **Graphics** There's hardly enough space here to get into every area where these operators are used in graphics programming. XOR (or ^) is particularly interesting here because applying the same input a second time will undo the first. Older GUIs used to rely on this for selection highlighting and other overlays, in order to eliminate the need for costly redraws. They're still useful in slow graphics protocols (i.e. remote desktop).

**Conversion to Two's Complement**

Note that this works both ways. If you have -30, and want to represent it in 2's complement, you take the binary representation of 30:

0000 0000 0000 0000 0000 0000 0001 1110

Invert the digits.

1111 1111 1111 1111 1111 1111 1110 0001

And add one.

1111 1111 1111 1111 1111 1111 1110 0010

System.***out***.println(Integer.*toBinaryString*(-30));

// Result is 1111 1111 1111 1111 1111 1111 1110 0010

## Unsigned Right Shift Operator >>>

This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.

3 >>> 2, the result is 0

System.***out***.println("( 12 >>> 2) :::"+(12 >>> 2)); //3

* Let's do 00000000 00000000 00000000 00001100 >>> 2 (12>>>2) and the result is 00000000 00000000 00000000 00000011 (3).
* Let's do 11111111 11111111 11111111 11110100 >>> 2 (-12>>>2) and the result is 00111111 11111111 11111111 11111101 (1073741821).