Operators and assignments
www.techfaq360.com

Determine the result of applying any operator, including assignment operators, instanceof, and casts to operands of any type, class, scope, or accessibility, or any combination of these.

There are different kinds of operators.
1. Unary operators.
1.1 Increment and Decrement operators : (++ and --)
You can apply these in two ways: postfix and prefix form. In post-fix form(eg. x++) value of the variable/expression is modified after the value is taken for the execution of statement. In prefix form(eg. ++x) , value of the variable/expression is modified before the value is taken for the execution of statement.

```
 x = 3;
 y = 1;
 y = x++;
 S.o.p(x+" "+y); //will print 4, 3
 x = 3;
 y = 1;
 y = ++x;
 S.o.p(x+" "+y); //will print 4, 4

 x = 3;
 x = x++; // Will print 3 !!! Why? Read on...
```

Steps:
.Take the value of x (ie.3) and keep in the register.
.Increment x (so, x becomes 4).
.Assign the value kept in the register to x. So, x again becomes 3.
1.2 Unary minus and unary plus(+ -) : + has no effect other than to stress positivity. - negates an expression’s value. (2’s complement for integral expressions)

```
 int i = 3;
 i = -i;
 S.o.p(i); //will print -3
 i = Integer.MIN_VALUE;
 i = -i; //Here, i will still be Integer.MIN_VALUE because -ive of Integer.MIN_VALUE exceeds Integer.MAX_VALUE by 1 so it does not fit into an int.
```

1.3 Boolean Negation (!) : Inverts the value of a boolean expression.

```
boolean flag = false;
S.o.p( !flag); //will print 'true'
```

1.4 Complement ~ (Only for integral types) Inverts the bit pattern of an integral expression.

```
int i = 12; // 0000 0000 0000 0000 0000 0000 0000 1100
i   =  ~i;  //  1111 1111 1111 1111 1111 1111 1111 0011
```

1.5 Cast () : Forces the compiler to cast one type of value to another type. Compiler still checks whether this cast is possible or not. If a compiler can prove that the
given cast can never be valid then it give a compile time error.
int 300; // 0000 0000 0000 0000 0000 0001 0010 1100 , doesn't fit into byte.
byte b = (byte) i; //the cast fits shoves 300 into a byte but takes only last 8 bits: 0010 1100, other bits are lost. So, b gets 44 instead of 300.
Consider this:
class A { }
class B { }
....
A a = new A();
B b = new B();
Object o = a; //All objects are Objects, so no cast is needed.
A a1 = (A) o; //All objects are not objects of class A, so cast is needed. Compiler sees that a variable of class Object can point to an object of class A, so ok.
B b1 = (B) a; //ERROR, compiler sees that a variable of class A, can NEVER point to an object of class B so there is no point in trying to cast a to b.

Arithmetic operators - (*, / , %, +, -) (Applied only to numeric types except + which can be applied to Strings) :
Important Point: All arithmetic operations are done after promoting (if needed) both the operands to 'int'. And the result is always atleast an int. That means, is any of the operands is smaller than an int, then it will be promoted to either an int or to the type of other operand if it is bigger than int. ie.
byte b1 = 10;
byte b2 = 20;
byte b3 = b1 + b2; //Will not compile as b1 and b2 will be promoted to int and the result is an int which can't be put into a byte without a cast.
byte b1 = 10;
long g = 20;
byte b3 = g + b2; //Will not compile as b2 will be promoted to long and the result is a long which can't be put into a byte without a cast.
EXCEPTION Compound operators: +=. -=, etc. b += 1; will compile because this is interpreted as: b = (byte) (b + 1); //Note the explicit casting. This is automatically done by the compiler for compound operators.
Points to Remember:
• Division by zero or % by 0, for integral values throws an exception but for float and double, no exception occurs as the result is Float/Double.POSITIVE_INFINITY or Float/Double.NEGATIVE_INFINITY (For /) OR Float/Double.NaN (for %) (NaN means: Not a Number)
% : Modulus operator. : Divide LHS by RHS and return the remainder. eg
32%7 = 4 (32 - 7*4 = 4),

32%-7 = 4 (32 - (-7 * -4) = 4),
(-32)%7 = -4 (-32 - (7*-4) = -4),
-32%-7 = -4 (-32 - (-7*4) = -4)

• You should observe that sign of the result is same as sign of the LHS.
• Floating point calculations can produce NaN (eg. 3.2%0 or square root of a negative number) or POSITIVE_INFINITY or NEGATIVE_INFINITY(division by zero).
• Float and Double wrapper classes have named constants for NaN and the two infinities.
• NaNs cannot be compared with any thing. (Float.NaN == Float.NaN is false!!!)
• To test a NaN, use Float.IsNaN(f); or Double.isNaN(d);
• Infinities can be compared to give appropriate results.
• System.out.println(1 + 2 + "3"); // Prints 33 System.out.println("1" + 2 + 3); // Prints 123 This is because, calculation is done from left to right.

Shift operators - (<<, >>, >>>) :
<< is used for shifting bits right to left. 0 bits are brought in from the right. Sign bit (MSB) is NOT preserved. Eg.
int i = 0x80000002;; int k = i<<1; i => 1000 0000 0000 0000 0000 0000 0000 0010 -2147483646 k => 0000 0000 0000 0000 0000 0000 0000 0100 => 4
Notice that sign bit of k (leftmost bit) is 0 that means it is a positive no. In affect, k is i*2 but as -2147483646*2 doesn't fit into an int, overflow occured and only last 32 bits were put into k.
>> is used for shifting bits left to right. The sign bit (the leftmost bit or the Most significant bit) keeps propagating towards right so Sign bit (MSB) IS preserved. Eg. int i = 0x80000002;; int k = i>>1; i => 1000 0000 0000 0000 0000 0000 0000 0010 -2147483646 k => 1100 0000 0000 0000 0000 0000 0000 0001 -1073741823 Notice that sign bit of k (leftmost bit) is 0 that means it is a negative no. In affect, k is i/2 .
There is never an overflow with >> or >>> But notice what happens with -1:
1111 1111 1111 1111 1111 1111 1111 1111 -1 1111 1111 1111 1111 1111 1111 1111 1111 -1 -1 >> -1 is -1
>>> is same as >> except that sign is not preserved. 0 bits are inserted from the right instead of the sign bits. Eg.
int i = 0x80000002;; int k = i>>>1; i => 1000 0000 0000 0000 0000 0000 0000 0010 -2147483646 k => 0100 0000 0000 0000 0000 0000 0000 0001 1073741825
Notice that sign bit of k (leftmost bit) is 0 that means it is a positive no. In affect, k is i/2 .
Points to remember:
• For int, i >> k is actually i >> (K%32) and For long, i >> k is actually i >> (K%64) So, i>>34 is i>>2 and i = i >> 32 will not change i at all !!!

Comparison operators :

<, <=, > , >= , instanceof <, <=, > , >= work as expected. instanceof operator checks the class of the object at LHS with the class name given at RHS.
eg. obj instanceof java.util.Collection
This will return true only if obj is refering to an object of class java.util.Collection or any subclass of java.util.Collection. If RHS is the name of an interface then LHS should point to an object of class that implements that interface.
In object oriented terminology: obj instanceof java.util.Collection will return true if obj is an instance of java.util.Collection! . Ie. If obj points to java.util.Set, it will return true as a set is-a collection.
Points to remember:
• LHS should be an object reference expression, variable or an array reference.
• RHS should be a class/interface. It throws compiler error if LHS & RHS are unrelated.
• You can also test for arrays.Eg.: obj instanceof Collection[] is legal.
• Returns false if LHS is null, no exception is thrown.
• == for objects tests whether the 2 references point to the same memory location or not.

Bit-wise operators : (&, ^ and |) : & (AND) operator. | (OR) operator work as expected. ^ (XOR) operator, returns 1 iff either and only one of LSH or RHS is 1/true.
Points to remember:
• These can be applied to numeric as well as boolean operands.
• In case of boolean operands, & and | DO NOT SHORT CIRCUIT the expression.

Logical operators : && and || : Also known a Short Circuit operators.
Points to remember:
• Can only be applied to booleans.
• These are also known as short circuiting operators because the RHS might not even be evaluated if the result can be determined only by looking at LHS.
Ex. (false && m1()) : Here, m1() will not be called as the expression will always be false no matter what m1() returns. So there is no point in calling m1(). Similarly, (true || m1()): This will always return true, no matter what m1() returns.

Assignment Operators: =, ?: and various compound assignment operators(like +=)
Important Fact: You may know that other operators return a value. Eg. a + b returns the sum of a and b, a^b returns the XOR of a and b etc. What you may not know is the = operator also returns a value. I.e a = 3 assigns 3 to a but the whole expression (a=3) also returns a value which is equal to the RHS of the operator ie. 3. That's the reason b = a = 3; works. And that's the reason if(flag = true){} also works.

Compound operators: a += b; is actually interpreted by the compiler as: a = (type of a) (a + b); So, byte b = 3; b = b+1; //won't work as b + 1 returns an int and explicit cast is needed.
b += 1; // will work because it is interpreted as: b = (byte) (b+1); Note the cast.
Ternary Operator:
Possible uses:
int a, b, c; // initialize the values somehow.
boolean flag = ...//some way of setting it.
a = flag? 10:20; a = (b == c) ? m1() : m2();
Here, it is important to know that it will only compile if the return types of m1() and m2 are compatible with the type of a. It won't compile if m1() or m2() return void.

Points to remember:
• ?: does not evaluate both the RHS parameters. In the above example, if b==c, only m1() will be evaluated(called).
• Assignment of reference variables copies the value of reference at RHS to LHS. So, in affect both the variables start pointing to same memory.

• Determine the result of applying the boolean equals(Object) method to objects of any combination of the classes java.lang.String, java.lang.Boolean, and java.lang.Object.

Consider the declarations:
String s1, s2; //initialize them somehow
Boolean bool1, bool2;//initialize them somehow
boolean flag1, flag2; //initialize them somehow
Object obj1; //initialize it somehow
Points to remember:
• String implements (and thus overrides) equals() method of Object class. It returns true if both the String objects contain same sequence of character.
Ex:
s1.equals(s2); //will return true only if s1 and s2 contain same data. It doesn't matter whether s1 and s2 point to same object or not.
• Boolean also implements equals() method. bool1.equals(bool2) will return true only if bool1 and bool2 both contain same value.
• bool1.equals(flag1) will NOT compile.
• Object class's equals() method firsts checks whether both the objects are of same class or not. So, obj1.equals(bool1) will return false. If not, then it returns false. It then simply checks whether the two object references point to the same memory or not which is same as "==" operator.

• In an expression involving the operators &, |, &&, ||, and variables of known values state which operands are evaluated and the value of the expression.

Points to Remember:

- && and || are short circuit operators. Examples:
- int i = 10;
- boolean flag = true;
- if(flag || ++i = 11) { ... } // Here, i will NOT be incremented.
- if(!flag && ++i = 11) { ... }// Here, i will NOT be incremented.

if(flag || m1()) { ... } // Here, m1 will NOT be called.

In the above cases, the outcome of the whole expression can be determined by just looking at the first part, so second part is NOT evaluated.

• Determine the effect upon objects and primitive values of passing variables into methods and performing assignments or other modifying operations in that method.

VERY IMPORTANT FACT: In java EVERY THING is passed by value. For primitive, it's value is passed (as expected). For object, the value of it's reference is passed. Read a detailed example explanation here : Pass by value

```
void changeObjects(String str, StringBuffer sb)
{
  str = "123";  //makes the reference str to point to new string object containing "123". you are changing the reference here.
  str = str + "123"; //strings are immutable. It will create a new string containing "abc123". The original "abc" will remain as it is.
  sb.append("123");  //changes the actual object itself. you are NOT changing the reference here.
}
```

....

```
String s = "abc";
StringBuffer sb = new StringBuffer("abc");
changeString(s, sb);
System.out.println(s); //Will still print "abc".
System.out.println(sb); //Will still print "abc123".
```