Operators

1. Precedence and Associativity

➔ Operator's **precedence** defines how an expression evaluates when many operators are present.

➔ Multiplication and Division have higher precedence than Addition and Substraction.

Ex: `1+2*3` is treated as `1+(2*3)` and `1*2+3` is treated as (`1*2)+3` since multiplication has higher precedence over addition.

➔ When an expression has two operators with same precedence then the expression is evaluated according to its **associativity**.

Ex: `72/2/3` is treated as `(72/3)/2` since operator '`/`' has left-to-right associativity.

➔ Below we have java operators from lowest to highest precedence, along with their associativity. Many programmers dont memorize them all and use parantheses for clarity.

➔ Short circuiting : When using the conditional 'and' and 'or' operators (`&&` and `||`), Java does not evaluate the second operand unless it is necessary to resolve the result. This allows statements like `if (s != null && s.length() < 10)` to work reliably. Programmers rarely use the non short-circuiting versions (`&` and `|`) with boolean expressions.

➔ Some Examples

1)

```
System.out.println("1 + 2 = " + 1 + 2);
System.out.println("1 + 2 = " + (1 + 2));
Output:
1 + 2 = 12
1 + 2 = 3
```

2)

```
System.out.println(1 + 2 + "abc");
System.out.println("abc" + 1 + 2);
output:
3abc
abc12
```

Note: Above we can notice that when String is followed by '+' and then a non-String then '+' acts for String concatenation and compiler tries to convert non-string into string. But if '+' is followed by primitive type then it acts for addition.

```
package operators;
public class AdditionOperator
{
    public static void main(String [] args)
    {
        String str = "Hello";
        int a = 5;
        int b = 6;
        System.out.println(str+a+b);
        System.out.println(a+b+str);
        System.out.println(a+str);
        System.out.println(a*b+str);
        System.out.println(str+a*b);
    }
}
Output:
Hello56
11Hello
5Hello
30Hello
Hello30
```

Operator	Description	Level	Associativity

`[]`
`.`
`()`
`++`
`--`
	access array element
	access object member
	invoke a method
	post-increment
	post-decrement
		1	left to right

`++`
`--`
`+`
`-`
`!`
`~`
	pre-increment
	pre-decrement
	unary plus
	unary minus
	logical NOT
	bitwise NOT
		2	right to left

`()` `new`	cast object creation	3	right to left

`*`
`/`
`%`
	multiplicative	4	left to right

`+ -` `+`	additive string concatenation	5	left to right
`<< >>` `>>>`	shift	6	left to right

`< <=`
`> >=`
 	
`instanceof`
	relational
	type comparison
		7	left to right

`==` `!=`	equality	8	left to right
`&`	bitwise AND	9	left to right
`|`	bitwise OR	11	left to right
`&&`	conditional AND	12	left to right
`||`	conditional OR	13	left to right
`?:`	conditional	14	right to left

`= += -=`
`*= /= %=`
`&= ^= |=`
`<<= >>= >>>=`
	assignment	15	right to left

2. Assignement

➔ Assignement is performed with operator '='. Ex a=4. Here 'a' generally called 'lvalue' and 4 is 'rvalue'.

➔ 'lvalue' must be a distinct, named variable and 'rvalue' could be a constant, variable or expression that produce a value.

➔ One cannot use rvalue and assign it to lvalue like 6=b;

➔ Assignment is not reference to a object hence when one do a=b then content of 'b' is copied into 'a'. Later if one change 'a' then 'b' doesn't get affected.

➔ But when one assign one object reference to another then things works differently.

3. Mathematical Expression

➔ Operatos

+ Addition

- Substraction

* Multiplication

/ Division

% Modulus

3.1. Unary minus and plus operators

➔ Unary minus(-) and plus(+) is same as binary minus and plus.

➔ Examples

```
package operators;

import java.util.Random;

public class UnaryExpression
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = 20;
        int c;

        c = -a;
        System.out.println("c = -a :"+c);
        c = +b;
        System.out.println("c = +b : "+c);
        c = a* -b;
        System.out.println("c = a* -b : "+c);
        c = a+++ ++b;
        System.out.println("c = a+++ ++b : "+c);

    }
}
Output:
c = -a :-10
c = +b : 20
c = a* -b : -231
c = a+++ ++b : 31
```

3.2. Pre-Increment and Post-Increment

➔ Example

```
package operators;


public class PreNPostIncrement
{
    public static void main(String[] args)
    {
        int i = 1;
        System.out.println("i : " + i);
        System.out.println("++i : " + ++i); // Pre-increment
        System.out.println("i++ : " + i++); // Post-increment
        System.out.println("i : " + i);
        System.out.println("--i : " + --i); // Pre-decrement
        System.out.println("i-- : " + i--); // Post-decrement
        System.out.println("i : " + i);
    }
}
Output
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
```

4. Relational Operators

➔ Relational operators generate boolean results.

➔ They are

< less than

> greater than

<= less than equal to

>= greater than equal to

== equivalent

!= not equivalent

➔ equivalent and non-equivalent works with all primitives but former four dont works with boolean values.

➔ Example

```
package operators;
public class Equivalence
{
    public static void main(String[] args)
    {
        Integer n1 = new Integer(47);
```

```
        Integer n2 = new Integer(47);
        System.out.println("n1 == n2 :"+(n1 == n2));
        System.out.println("n1 == n2 :"+(n1 != n2));
        System.out.println("n1.equals(n2) = "+n1.equals(n2));
    }
}
Output:
n1 == n2 :false
n1 == n2 :true
n1.equals(n2) = true
```

➔ Above one will expect `n1==n2` should be '`true`' but it turns out to be '`false`'. This happens because while content of the objects are same but their references are not same. To compare the content of object one should use `n1.equals(n2)`.

➔ As we have seen earlier in assignment section that for `int a=10`, a is a primitive data type and not a reference and it holds value of 10. But for Integer a = new Integer(10) , a is reference to the object which wraps primitive type int.

➔ Similarly for string we have primitive data type 'String' which we initialize as String str = “Hello” and could be wrapped in a object with String str = new String(“Hello”).

➔ Example

```
package JavaInterviewPrep;
public class StringEquivalence {

    public static void main(String [] args)
    {
        String str1 = "Jayant";
        String str2 = "Jayant";
        String str3 = new String("Jayant");
        String str4 = new String("Jayant");

        boolean check = (str1 == str2);
        System.out.println("str1 == str2 : "+check);

        check = (str1 == str3);
        System.out.println("str1 == str3 : "+check);

        check = (str3 == str4);
        System.out.println("str3 == str4 : "+check);

        check = str1.equals(str2);
        System.out.println("str1.equals(str2) : "+check);

        check = str3.equals(str4);
        System.out.println("str3.equals(str4) : "+check);

        System.out.println("str1 hashcode : "+str1.hashCode());
        System.out.println("str2 hashcode : "+str2.hashCode());
        System.out.println("str3 hashcode : "+str3.hashCode());
  }
}
Output:
str1 == str2 : true
str1 == str3 : false
str3 == str4 : false
str1.equals(str2) : true
str3.equals(str4) : true
str1 hashcode : -2083127131
str2 hashcode : -2083127131
str3 hashcode : -2083127131
```

Note: Above we can see when objects are same their hashcode must be same.

➔ Also check

```
package operators;

import testingPgms.*;

class CheckEquals1
{
    int a;
    public CheckEquals1(int a)
    {
        this.a = a;
    }
}
public class CheckEquals {

    public static void main(String [] args)
    {
        CheckEquals1 ce1 = new CheckEquals1(10);
        CheckEquals1 ce2= new CheckEquals1(10);

        System.out.println(ce1.equals(ce2));

        ce1 = ce2;

        System.out.println(ce1.equals(ce2));
    }
}
Output:
false
true
```

Note: Above we can see two object can only be equals when their references are pointing to the same object. Because default behavior of method equals() is to compare references. But If one want to return it 'true' without references then equals() method should be overridden as they have done in case of '`Integer`' and '`String`' class.

➔ Most of the Java library classes implement equals( ) so that it compares the contents of objects instead of their references.

➔ The overridden equals() method of 'Integer' class is as shown below

```
public boolean equals(Object obj) {
        if (obj instanceof Integer) {
            return value == ((Integer)obj).intValue();
        }
        return false;
    }
```

Above its constructor is

```
public Integer(int value)
{
        this.value = value;
}
```

Hence we can understand in overridden equals() only 'value' of two objects has been compared. Without this overriden equal it will always be false as we have seen in case of class 'CheckEqual1'.

➔ Summary

'==' returns true for only primitive data types.

equals() method returns true when both references are either pointing to the same object or they have overridden equals() methods.

5. Logical Operators

➔ Each of logical operators AND(&&), OR(||) and NOT (!) produces a boolean value of true and false.

➔ Also they can be applied to boolean values only.

Example:

```
int a = 10;
int b = 20;
System.out.println(a&&b);//this won't compile
```

➔ `if(a<20 || b<30) //will compile fine`

5.1. Short Circuting

➔ It is the phenomena when after evaluating one part of the expression the value of full expression could be determined hence later part can be ignored.

➔ Ex : `boolean b = (a<1) && (b<2) && (c<3);`

Above when it is found that (a<1) is true but (b<2) is false then value of entire the entire expression will be false. Hence expression (c<3) will not be evaluated. This will increase performance.

6. Bitwise operators

AND (&) - Produces 1 if both operands are one.

OR (|) - Produces 1 if either of the operand is one.

XOR (^) - See table below.

NOT (`) - Change every one into zero and zero into one.

a	b	XOR(^)
0	0	0
1	0	1
0	1	1
1	1	0

➔ Example

```
package operators;

public class BitwiseOperator
{
    public static void main(String [] args)
    {
        int a = 1;
        int b = 1;
        int c = 0;
        System.out.println("a = "+a+" b = "+b+" c = "+c);
        System.out.println("a & b = "+(a&b));
        System.out.println("a | b = "+(a|b));
        System.out.println("a ^ b = "+(a^b));
        System.out.println("a ^ c = "+(a^c));
        System.out.println("c ^ c = "+(c^c));
        System.out.println("~a = "+~a);
    }
}
Output:
```

```
a = 1 b = 1 c = 0
a & b = 1
a | b = 1
a ^ b = 0
a ^ c = 1
c ^ c = 0
~a = -2
```

7. Shift Operators

<< Left shift operator

>> Right shift operator(signed)

>>> unsigned right shift operator

8. Conditional (Ternary if-else)operator

➔ boolean-exp ? Value0 : value1

When boolean-exp is true then value0 is evaluated else value1 is evaluated.

➔ Example

```
package operators;

public class ConditionalOperator
{

    static int ternary(int i)
    {
        return i < 10 ? i * 100 : i * 10;
    }
    static int standardIfElse(int i)
    {
        if(i < 10)
        return i * 100;
        else
        return i * 10;
    }

    public static void main(String[] args)
    {
        System.out.println(ternary(9));
        System.out.println(ternary(10));
        System.out.println(standardIfElse(9));
        System.out.println(standardIfElse(10));
    }
}
Output:
900
100
900
100
```

- Assignment :

```
//: operators/Assignment.java
// Assignment with objects is a bit tricky.
import static net.mindview.util.Print.*;
class Tank {
int level;
}
public class Assignment {
public static void main(String[] args) {
```

```
Tank t1 = new Tank();
Tank t2 = new Tank();
t1.level = 9;
t2.level = 47;
print("1: t1.level: " + t1.level +
", t2.level: " + t2.level);
t1 = t2;
print("2: t1.level: " + t1.level +
", t2.level: " + t2.level);
t1.level = 27;
print("3: t1.level: " + t1.level +
", t2.level: " + t2.level);
}
} /* Output:
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27 */
```

Note: import static net.mindview.util.Print.* above provide direct use of print.

Above we can see that during 3rd output both t1.level and t2.level become 27. This happens because **reference** t2 is assigned to t1. This is called **ALIASING**. To avoid the aliasing use t1.level = t2.level.

- Aliasing during method call:

```
package ch3Operators;

class Letter
{
      char c;
}
public class Ch3Aliasing {
      static void f(Letter y)
      {
            y.c='z';
      }
      public static void main(String[] args) {
            // TODO Auto-generated method stub
            Letter x = new Letter();
            x.c='a';
            System.out.println("1:x.c: "+x.c);
            f(x);
            System.out.println("1:x.c: "+x.c);
            }                                          //Output:
}                                                      // 1:x.c: a
                                                       // 1:x.c: z
```