Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java:Operators/5aa01d243dde0810c26d5447

TUTORIAL

Java: Operators

Topics

Arithmetic expressions are especially important. Java has many operators for arithmetic. These operators can be used on floating point numbers and on integer numbers. (However, the % operator is rarely used on floating point.) For instance, / means integer division if both operands are integers, and means floating point division if one or both operands are floating point.

```
Operator         Meaning                  Precedence
-                unary minus              Highest
+                unary plus               Highest
*                multiplication           Middle
/                division                 Middle
%                remainder                Middle
+                addition                 Low
-                subtraction              Low
```

An integer operation is always done with 32 bits or more. If one or both operand is 64 bits (data type long) then the operation is done with 64 bits. Otherwise the operation is done with 32 bits, even if both operands are smaller than 32 bits.

For example, with 16 bit short variables, the arithmetic is done using 32 bits:

```
short num1 = 2;        // 16 bit short
int totVar;                     // 32 bit int
totVar = num1 / 3;      // arithmetic will be done using 32 bits
```

The expression num1 / 3 divides a 32-bit value 2 by a 32-bit value 3 and puts the 32-bit answer in totVar. The literal 3 automatically represents a 32-bit value.

Here is another example:

```java
class Main
{
  public static void main(String ab[])
  {
    short x = 12;
    short y = 3;
    short value;
    value = x / y;
    System.out.println("x + y = " + value);
  }
}

```

The expression x / y divides a 32-bit value 12 by a 32-bit value 3, even though the variables x and y are only 16 bits wide. The calculation produces a 32-bit result. Because the 32-bit result does not fit in the 16 bits of value the compiler will not compile the last statement. This can be completely baffling when it happens to you.

```
code@cq:~$ javac ShortStuff.java
ShortStuff.java:9: possible loss of precision
found   : int
required: short
    result = x / y;
               ^
1 error
code@cq:~$
```

For professional programmers, details like these are sometimes important. But for most programs, just use int or long for integers and double for floating point. This will keep you out of trouble (usually). If you can't avoid the problem, use a type cast, described in later chapters.

The division operator / means integer division if there is an integer on both sides of it. If one or two sides has a floating point number, then it means floating point division. The result of integer division is always an integer. Integer division determines how many times one integer goes into another. The remainder after integer division is simply dropped, no matter how big it is.

There is a difference between what Java will do and what a calculator will do. A calculator will do floating point arithmetic for the expression:

```
7/4
```

A calculator will show this as 1.75. Java will regard this as integer arithmetic and give you:

```
7/4 = 1
```

because 4 goes into 7 just once. The result is not rounded up to 2. The remainder after division, 3, is simply dropped. Integer operations and floating point operations are both very common in programs. It is important to be clear about them.

So, the following lines produces awkward but correct (in rules of Java) result: -

```
1/2 + 1/2  = 0
```

It looks as though this is a mistake, but no: each 1/2 calls for integer division, resulting in an integer 0. The two zeros are added to get the final answer, zero. If you really want to add one half to one half you should write 1.0/2.0 + 1.0/2.0 because now the decimal points make each number a double.

The normal rules of arithmetic are used to determine the sign of the result of division:

```
+num/+div = +result
-num/+div = -result
+num/-div = -result
-num/-div = +result
```

13 / 5 = 2 with a remainder of 3. This is because 13 = 2*5 + 3. The symbol for finding the remainder is % (percent sign). This symbol is also called modulo operator. If you look in the table of operators you will see that it has the same precedence as / and *. Here is a program:

```java
class Main
{
  public static void main ( String[] args )
  {
    int quotient, remainder;
    quotient  =  17 / 3;
    remainder =  17 % 3;
    System.out.println("The quotient : " + quotient );
    System.out.println("The remainder: " + remainder );
    System.out.println("The original : " + (quotient *
3 + remainder));
  }
}

```

Increment and Decrement Operators

Programmers, of course, know that one of the most common operations with a numeric variable is to add or subtract 1. Java, following in the footsteps of C and C++, has both increment and decrement operators: n++ adds 1 to the current value of the variable n, and n-- subtracts 1 from it. For example, the code

```
int n = 12;
n++;
```

changes n to 13. Because these operators change the value of a variable, they cannot be applied to numbers themselves. For example, 4++ is not a legal statement. There are actually two forms of these operators; you have seen the “postfix” form of the operator that is placed after the operand. There is also a prefix form, ++n. Both change the value of the variable by 1. The difference between the two only appears when they are used inside expressions. The prefix form does the addition first; the postfix form evaluates to the old value of the variable.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // now a is 16, m is 8
int b = 2 * n++; // now b is 14, n is 8
```

Relational Operators

A boolean expression is an expression that evaluates to true or false. Boolean expressions often compare numbers. A relational operator says how the numbers are compared.

```
Operator                Meaning
A == B                  is A equal to B ?
A < B                   is A less than B ?
A <= B                  is A less than or equal to B ?
A > B                   is A Greater than B ?
A >= B                  is A Greater than or equal to B ?
A != B                  is A not equal to B ?
```

Java also supports the ternary ?: operator that is occasionally useful. The expression

```
condition ? expression1 : expression2
```

evaluates to the first expression if the condition is true, to the second expression otherwise.

For example,

```
x < y ? x : y
```

gives the smaller of x and y.

Bitwise operators

When working with any of the integer types, you have operators that can work directly with the bits that make up the integers. This means that you can use masking techniques to get at individual bits in a number. The bitwise operators are

```
& (“and”)        | (“or”)             ^ (“xor”)          ~ (“not”)
```

These operators work on bit patterns. For example, if n is an integer variable, then

```
int fourthBitFromRight = (n & 8) / 8;
```

gives you a 1 if the fourth bit from the right in the binary representation of n is 1, and 0 if not. Using & with the appropriate power of 2 lets you mask out all but a single bit.

Operator Precedence

Following table shows the precedence of operators. If no parentheses are used, operations are performed in the hierarchical order indicated. Operators on the same level are processed from left to right, except for those that are right associative, as indicated in the table.

Table : Operator Precedence and Hierarchy

```
Operators
Associativity
```

```
[] . () (function call)                              Left to
right
! ~ ++ -- + (unary) – (unary) () (cast) new          Right to
left
*    /    %                                          Left to
right
-    +                                               Left to
right
<<   >>   >>>                                        Left to
right
<   <=   >   >=   instanceof                         Left to
right
==   !=                                              Left to
right
&                                                    Left to
right
^                                                    Left to
right
|                                                    Left to
right
&&                                                   Left to
right
||                                                   Left to
right
?:                                                   Right to
left
= += -= *= /= %=  &=  |=  ^=  <<= >>=  >>>=          Right to
left
```