BCA PART- III

PAPER – XIX

TOPIC: OPERATORS AND EXPRESSIONS

PREPARED BY: DR. KIRAN PANDEY

(SCHOOL OF COMPUTER SCIENCE)

EMAIL ID: kiranpandey.nou@gmail.com

Introduction

Java supports a rich set of operators. Operators are used in programs to manipulate data and variables. They usually form a part of mathematical or logical expressions.

Java operators can be classified into a number of related categories as below:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

Arithmetic Operators

Arithmetic operators are used to construct mathematical expressions as in algebra. Java provides all the basic arithmetic operators. They are listed in Tabled 3.1. The operators +, –, *, and / all works the same way as they do in other languages. These can on any built-in numeric data type of Java. We cannot use these operators on Boolean type. The unary minus operator, in effect, multiplies its single operand by –1. Therefore, a number preceded by a minus sign changes its sign.

Table 3.1 Arithmetic Operators

Operator	*Meaning*
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division (Remainder)

Arithmetic operators are used as shown below:

a – b a + b

a * b a / b

a % b – a * b

Here **a** and **b** may be variables or constants and are known as operands.

Integer Arithmetic

When both the operands in a single arithmetic expression such as a + b are integers, the expression is called an *integer expression*, and the operation is called *integer arithmetic*. Integer arithmetic always yields an integer value. In the above examples, if a and b are integers, then for a = 14 and b = 4 we have the following results:

a – b = 10

a + b = 18

a * b = 56

a / b = 3 (decimal part truncated)

a % b = 2 (remainder of integer division)

a/b, when **a** and **b** are integer types, gives the result of division of **a** by **b** after truncating the divisor. This operation is called the *integer division*.

For modulo division, the sign of the result is always the sign of the first operand (the dividend). That is

– 14 % 3 = – 2

– 14 % – 3 = – 2

14 % – 3 = 2

(*Note that module division is defined as : a%b = a – (a/b)*b, where a/b is the integer division).*

Real Arithmetic

An arithmetic operation involving only real operands is called *real arithmetic*. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result.

Unlike **C** and **C++**, modulus operator % can be applied to the floating point data as well. The floating point modulus operator returns the floating point equivalent of an integer division. What this means is that the division is carried out with both floating point operands, but the resulting divisor is treated as an integer, resulting in a floating point remainder. Program 3.1 shows how arithmetic operators work on floating point values.

Program 3.1 Floating point arithmetic

```
class FloatPoint
{
  public static void main(String args[])
  {
      float a = 20.5F. b = 6.4;
      System.out.println(“ a = ” + a);
      System.out.println(“ b = ” + b);
      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/b = ” + (a/b));
      System.out.println(“ a%b = ” + (a%b));
  }
```

```
}
```

The output of Program 3.1 is as follows:

```
a = 20.5
b = 6.4
a+b = 26.9
a-b = 14.1
a*b = 131.2
a/b = 3.20313
a%b = 1.3
```

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a *mixed-mode arithmetic* expression. If either operand is of the real type, then the other operand is converted to real and the ral arithmetic is performed. The result will be a real. Thus

```
15/10.0   produces the result   1.5
```

Whereas

```
15/10     produces the result     1
```

More about mixed operations will be discussed later when we deal with the evaluation of expressions.

Relational Operators

We often compare two quantities, and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of *relational operators*. We have already used the symbol '<' meaning 'less than'. An expression such as

```
a < b or x < 20
```

containing a relational operator is termed as a *relational expression*. The value of relational expression is either true or false. For example, if x = 10, then

```
x < 20 is true
```

while

```
20 < x is false.
```

Java supports six relational operators in all. These operators and their meanings are shown in Table 3.2

Table 3.2 Relational Operators

Operator	*Meaning*
<	is less than
<=	is less than or equal to
>	Is greater than
>=	Is greater than or equal to
= =	Is equal to
!=	is not equal to

A simple relational expression contains only one relational operator and is of the following form:

```
ae-1 relational operator ae-2
```

ae – 1 and *ae – 2* are arithmetic expression, which may be simple constants, variables or combination of them. Table 3.2 shows some examples of simple relational expressions and their values.

Table 3.3 Relational Expressions

Expression	*Value*
4.5 <= 10	TRUE
4.5 < –10	FALSE
– 35 >= 0	FALSE
10 < 7 + 5	TRUE
a + b = = c + d	TRUE*

* Only if the sum of values of a and b is equal to the sum of values of c and d.

When arithmetic expression are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, *arithmetic operators have a higher priority over relational operators*. Program 3.2 shows the implementation of relational operators.

Program 3.2 Implementation of relational operators

```
class RelationalOperators
{
  public static void main(String args[])
  {
      float a = 15.0F, b = 20.75F, c = 15.0F;
      System.out.println(" a = " + a);
      System.out.println(" b = " + b);
      System.out.println(" c = " + c);
      System.out.println(" a < b is " + (a<b));
      System.out.println(" a > b is " + (a>b));
      System.out.println(" a == c is " + (a==c));
      System.out.println(" a <= c is " + (a<=c));
      System.out.println(" a >= b is " + (a>=b));
      System.out.println(" b != c is " + (b!=c));
      System.out.println(" b == a+c is " + (b==a+c));
  }
}
```

The output of Program 5.2 would be:

```
a = 15
b = 20.75
c = 15
a < b is true
a > b is false
a == c is true
a <= c is true
a >= b is false
a != c is true
b == a+c is false
```

Relational expressions are used in *decision statements* such as, **if** and **while** to decide the course of action of a running program. Decision statements are discussed in detail in Chapters 6 and 7.

Logical Operators

In addition to the relational operators, Java has three logical operators, which are given in Table 3.4.

Table 3.4 Logical Operators

Operator	*Meaning*
&&	logical AND
||	logical OR
!	logical NOT

The logical operators && and || are used when we want to form compound conditions by combining two or more relations. An example is:

```
a > b && x == 10
```

An expression of this kind which combines two or more relational expressions is termed as a *logical expression* or a *compound relational expression*. Like the simple relational expressions, a logical expression also yields a value of true or false, according to the *truth table* shown in Table 3.5. The logical expression is given above is true only if both **a > b** and **x = 10** are true. If either (or both) of them are false the expression is false.

Table 3.5 Truth Table

op – 1	*op – 2*	*Value of the expression*	
		op – 1 && op – 2	*op – 1 || op – 2*
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Note:

- *op – 1 && op – 2 is true if both op – 1 and op – 2 are true and false otherwise.*
- *op – 1 || op – 2 is false if both op – 1 and op – 2 are false and true otherwise.*

Some examples of the usage of logical expression are:

```
1. if (age>55 && salary<1000)
2. if (number<0) || number>1000)
```

Assignment Operators

Assignment operators are used to assign the value of an expression to a variable. We have seen the usual assignment operator, '='. In addition, Java has a set of 'shorthand' assignment operators which are used in the form

```
v op= exp;
```

where v is a variable, *exp* is an expression and *op* is a Java binary operatory. The operator **op** = is known as the shorthand assignment operator.

The assignment statement

```
v op= exp;
```

is equivalent to

```
v = v op(exp);
```

with v accessed only once. Consider an example

```
x += y+1;
```

This is same as the statement

```
x = x+(y+1);
```

The shorthand operator += means 'add y + 1 to x' or 'increment x by y + 1'. For y = 2, the above statement becomes

```
x += 3;
```

and when this statement is executed, 3 is added to x. If the old value of x is, say 5, then the value of x is 8. Some of the commonly used shorthand assignment operators are illustrated in Table 3.6.

Table 3.6 Shorthand Assignment Operators

Statement with simple assignment operator	*Statement with shorthand operator*
`a = a+1`	`a += 1`
`a = a–1`	`a –= 1`
`a = a*(n+1)`	`a *= n+1`
`a = a/(n+1)`	`a /= n+1`
`a = a%b`	`a %= b`

The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. Use of shorthand operator results in a more efficient code.

Increment and Decrement Operators

Java has two very useful operators not generally found in many other languages. These are the increment and decrement operators.

```
++ and --
```

The operator ++ adds 1 to the operand while — subtracts 1. Both are unary operators and are used in the following form:

```
++m; or   m++;
--m; or   m--;
++m; is equivalent to m = m + 1; (or m += 1;)
--m; is equivalent to m = m - 1; (or m -= 1;)
```

We use the increment and decrement operators extensively in **for** and **while** loops.

While ++m and m++ mean the same thing when they from statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m = 5;
y = ++m;
```

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statement as

```
m = 5;
y = m++;
```

then, the value of y would be 5 and m would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand. Program 3.3 illustrates this.

Program 3.3 Increment Operator Illustrated

```
class IncrementOperator
{
   public static void main(String args[])
   {
      int m = 10, n = 20
      System.out.println(“ m = ” + m);
      System.out.println(“ n = ” + n);
      System.out.println(“ ++m = ” +++m n);
      System.out.println(“ n++ = ” + n++);
      System.out.println(“ m = ” + m);
      System.out.println(“ n = ” + n);
   }
}
```

Output of Program 3.3 is as follows:

```
m = 10
n = 20
++m = 11
n++ = 20
m = 11
n = 21
```

Similar is the case, when we use ++ (or —) in subscripted variables. That is, the statement

```
a[i++] = 10
```

is equivalent to

```
a[i]  = 10
i     = i+1
```

Conditional Operator

The character pair ? : is a ternary operatory available in Java. This operator is used to construct conditional expressions of the form

exp1 ? exp2 : exp3

where *exp1*, *exp2*, and *exp3* are expressions.

The operator ? : works as follows: *exp1* is evaluated first. If it is nonzero (true), then the expression *exp2* is evaluated and becomes the value of the conditional expression. If *exp1* is false, *exp3* is evaluated and its value becomes the value of the conditional expression. None that only one of the expressions (either *exp2* or *exp3*) is evaluated. For example, consider the following statements:

```
a = 10;
b = 15;
x = (a > b) ? a : b;
```

In this example, x will be assigned the value of b. This can be achieved using the **if….else** statement as follows:

```
if(a > b)
       x = a;
else
```

```
x = b;
```

Bitwise Operators

Java has a distinction of supporting special operators known as bitwise operators for manipulation of data at values of bit level. These operators are used for testing the bits, or shifting them to the right or left. Bitwise operators may not be applied to **float** or **double**. Table 3.7 lists the bitwise operators. They are discussed in detail in Appendix D.

Table 3.7 Bitwise Operators

Operator	*Meaning*
&	bitwise AND
!	bitwise OR
^	Bitwise exclusive OR
~	one's complement
<<	shift left
>>	shift right
>>>	shift right with zero fill

Special Operators

Java supports some special operators of interest such as **instanceof** operatory and member selection operator (.).

Instaceof Operator

The **instanceof** is an object operator and returns ***true*** of the object on the left-hand side is an instance of the class given on the right-hand side. This operator allows us to determine whether the object belongs to a particular class or not.

Example:

```
person      instanceof      student
```

is ***true*** if the object **person** belongs to the class **student**; otherwise it is ***false***.

Dot Operator

The dot operator (.) is used to access the instance variables and methods of class objects. Examples:

```
personal.age        / /   Reference to the variable age
personal.salary     / /   Reference to the method salary()
```

It is also used to access classes and sub-packages from a package.

Arithmetic Expressions

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. We have used a number of simple expression in the examples discussed so far. Java can handle any complex mathematical expressions. Some of the examples of Java expressions are shown in Table 3.8. Remember that Java does not have an operator for exponentiation.

Table 3.8 Expressions

Algebraic expression	*Java expression*
a b–c	a*b–c
(m+n)(x+y)	(m+n)*(x+y)
$\frac{ab}{c}$	a*b/c

$3x^2+2x+1$	3*x*x+2*x+1
$\frac{x}{y}+c$	x/y+c

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

```
variable = expressions;
```

variable is any valid Java variable name. When the statement is encountered, the *expression* is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned value before evaluation is attempted. Examples of evaluation statements are

```
x = a*b-c;
y = b/c*a;
z = a-b/c+d;
```

The blank space around an operator is optional and is added only to improve readability. When these statements are used in program, the variables **a,b,c** and **d** must be defined before they are used in the expressions.

Precedence of Arithmetic Operators

An arithmetic expression without any parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in Java:

The basic evaluation procedure includes two left-to-right passed through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered.

```
High priority     * / %
Low priority      + -
```

During the second pass, the low priority operators (if any) are applied as they are encountered. Consider the following evaluation statement:

```
x = a-b/3 + c*2-1
```

When a = 9, b = 12, and c = 3, the statement becomes

```
x = 9-12/3+3*2-1
```

and is evaluated as follows:

First pass

```
Step1: x = 9-4+3*2-1     (12/3 evaluated)
Step2: x = 9-4+6-1       (3*2 evaluated)
```

Second pass

```
Step3: x = 5+6-1         (9-4 evaluated)
Step4: x = 11-1          (5+6 evaluated)
Step5: x = 10            (11-1 evaluated)
```

However, the order of evaluation can be changed by introducing parentheses into an expression. Consider the same expression with parentheses as shown below:

```
9-12/(3+3)*(2-1)
```

Whenever the parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the

expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

First pass

```
Step1: 9-12/6*(2-1)
Step2: 9-12/6*1
```

Second pass

```
Step3: 9-2*1
Step4: 9-2
```

Third pass

```
Step5: 7
```

This time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remain the same as 5 (i.e., equal to the number of arithmetic operators).

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. Just make sure that every parentheses has a matching closing one. For example

```
9-(12/3(3+3)*2)-1 = 4
```

Whereas

```
9-((12/3)+3*2)-1 = -2
```

While parentheses allow is to change the order of priority, we may also use them to improve understandability of the program. When in doubt, we can always add an extra pair just to make sure that the priority assumed is the one we require.

Type Conversions in Expressions

Automatic Type Conversion

Java permits mixing of constant and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversion. We know that the computer, considers one operator at a time, involving two operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of the higher type.

If **byte, short** and **int** variables are used in an expression, the result is always promoted to **int**, to avoid overflow. If a single **long** is used in the expression, the whole expression is promoted to **long.** Remember that all integer values are considered to be **int** unless they have the 1 or L appended to them. If an expression contains a **float** operand, the entire expression is promoted to **float**. If any operand is **double**, result is **double**. Table 5.9 provides a reference chart for type conversion.

Table 3.9 Automatic Type Conversion Chart

	char	*byte*	*short*	*int*	*long*	*float*	*double*
char	int	int	int	int	long	float	double
byte	int	int	int	int	long	float	double
short	int	int	int	int	long	float	double
int	int	int	int	int	long	float	double
long	long	long	long	long	long	float	double
float	float	float	float	float	float	float	double
double	double	double	double	double	double	double	double

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.

1. **float** to **int** causes truncation of the fractional part.

2. **double** to **float** causes rounding of digits.
3. **long** to **int** causes dropping of the excess higher order bits.

Casting a Value

We have already discussed how Java performs type conversion automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ration of females to males in a town.

```
ratio = female_number/male_number
```

Since **female_number and male_number** are declared as integers in the program, the decimal part of the result of the division would be lost and **ratio** would not represent a correct figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

```
ratio = (float)female_number/male_number
```

The operator **(float)** converts the **female_number** to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator **(float)** affect the value of the variable **female_number**. And also, the type of **female_number** remains as **int** in the other parts of the program.

The process of such a local conversion is known as *casting a value*. The general form of a cast is:

```
(type_name) expression
```

where type-name is one of the standard data types. The *expression* may be a constant, variable or an expression. Some examples of casts and their actions are shown in Table 3.10.

Examples	*Action*
`x = (int) 7.5`	7.5 is converted to integer by truncation
`a = (int)21.3/(int)4.5`	Evaluated as 21/4 and the result would be 5
`b = (double) sum/n`	Division is done in floating point mode.
`y = (int) (a+b)`	The result of a + b is converted to integer.
`z = (int) a+b`	**a** is converted to integer and then added to b.
`p = cost (double)x)`	Converts x to double before using it as parameter.

Casting can be used to round-off a given value to an integer. Consider the following statement:

```
x = (int) (y+0.5);
```

If y is 27.6, y + 0.5 is 28.1 an on casting, the result becomes 28, the value that is assigned to **x**. Of course, the expression being cast is not changed.

When combining two different types of variables in an expression, never assume the rules of automatic conversion. It is always a good practice to explicitly force the conversion. It is more safer. For example, whey **y** and **p** are **double** and **m** is **int**, the following two statements are equivalent.

```
y = p+m;
y = p+(double)m;
```

However, the second statement is preferable.

Program 3.4 illustrates the use of casting in evaluating the equation

$$\text{sum} = \sum_{i=1}^{n} \frac{1}{i}$$

Program 3.4 Illustration of used of casting operation

```
class Casting
{
  public static void main(String args[])
  {
    float sum;
    int i;
    sum = 0.0F;
    for(i = 1; i <= 10; i++)
    {
      sum = sum + 1/(float)i;
      System/out.print(“ i = ” + i);
      System.out.print(“ sum = ” + sum);
    }
  }
}
```

Program 5.4 produces the following output:

```
i = 1     sum = 1
i = 2     sum = 1.5
i = 3     sum = 1.83333
i = 4     sum = 2.08333
i = 5     sum = 2.28333
i = 6     sum = 2.45
i = 7     sum = 2.59286
i = 8     sum = 2.71786
i = 9     sum = 2.82897
i = 10    sum = 2.92897
```