From Balaguru +

CHAPTER 4

Operators and Statements

In Chapter 3 we have seen the various data types used in the Java programming language. In this chapter, we discuss the various commonly used operators and statements available in the Java programming language.

4.1 OPERATORS

Most of the operators in Java work similar to C. In the following sections we discuss the operators available in Java. Assuming the knowledge of C, we do not discuss each and every operator in detail except for places where it is felt that the operators differ from C or they require an in-depth discussion. The operators in Java may be categorized as follows.

4.1.1 Arithmetic Operators

```
+ - * / and %
```

The above operators when used as binary operators can have operands of numeric types only, except for the + operator, which is also used for string concatenation. In Java, arithmetic is done on atleast `int` values. When we use these as binary operators, the two operands may not be the same type. In case the two numeric operands are not the same type, then the result of the expression will be the bigger (super-type) of the two types, and it would be at atleast an `int`. The arithmetic is carried out by first converting the operands to the result type and then the arithmetic is done. The super-type and sub-type relationships among the numeric types are given in Figure 3.7.

Among the numeric types the relationships about the super-type and sub-type are as follows.

`double` is the super-type for all numeric primitive types, `float` is the sub-type of `double`, `long` is the sub-type of `float`, `int` is the sub-type of `long`, `short` and `char` are sub-types of `int` and `byte` is the sub-type of `short`.

Therefore, if an arithmetic operator has two operands, one of type `byte` and other of type `short`, the resulting expression will be of type `int` since both the types are smaller than

int. Also when arithmetic is to be done, then both the operands will be converted to an int, and then arithmetic will be done on two int values.

If in the two operands for an arithmetic operator, one is an int and the other is a float, then the int operand will be converted to float and finally the arithmetic will be done on two float values. The result of the expression will be a float.

4.1.1.1 Conversion of Numeric Types

There are various types of conversions involving numeric types. The conversion may be a widening conversion, a narrowing conversion or a mixed conversion from one numeric type to another. A string conversion is done from any data type to string. This is discussed in Section 4.1.2.1. From Java 5 onwards two more types of conversion are also supported involving any primitive type and a wrapper-class type. These conversions involving the wrapper-class are known as the boxing and the unboxing conversions. The boxing and unboxing conversions are discussed in Section 10.6.2. Let us look at some of these conversions.

Widening conversion: A widening conversion is the conversion of a sub-type to one of its super-types. For numeric types, the following are the widening conversions:

- byte to short, int, long, float or double
- short to int, long, float or double
- char to int, long, float or double
- int to long, float or double
- long to float or double
- float to double

How is the widening conversion of numeric types done? The widening conversion of an integral to a higher integral type is simple. In this case the additional high order bits are simply filled with the sign bit, except in case of char type where the additional high order bits are filled with zeros.

In case of widening conversion from int to float and long to float or double, there can be a loss of precision, in which case some of the low order bits are lost; i.e. if we have a widening conversion from int to a float and then a narrowing conversion from float back to int then it may not be the same value in all the cases, especially if the values are of very high magnitude. Consider the code in Listing 4.1.

Listing 4.1. Loss of precision

```
1 class Test {
2     public static void main(String[] args) {
3         int i = 123456789;
4         float f = i;
5         int j = (int)f;
6         System.out.println("i = "+i+", j = "+j);
7     }
8 }
```

The output of the above code is:

```
i = 123456789, j = 123456792
```

The above result shows that there is a loss of precision when converting from `int` to `float`. A similar loss of precision is also seen when converting from `long` values of higher magnitude to `float` or `double`.

Narrowing conversion: A narrowing conversion is the conversion of a super-type to one of its sub-types. For numeric types, the following are the narrowing conversions:

- `double` to `byte`, `short`, `char`, `int`, `long` or `float`
- `float` to `byte`, `short`, `char`, `int` or `long`
- `long` to `byte`, `short`, `char` or `int`
- `int` to `byte`, `short` or `char`
- `short` to `byte`

How is the narrowing conversion of numeric types done? The narrowing conversion of an integral to a smaller integral type is done by dropping the most significant bits. In this case the additional high order bits are simply truncated. This may even result in a change in the sign as the sign bit is the most significant bit. In case of conversion to `char` type, the sign will always be positive since the `char` type always has unsigned values.

In case of the narrowing conversion from the floating-point type to an integral type, the conversion is done by first converting to either `long` or `int`, and then a narrowing conversion to the target integral type is done. The first step would convert to `long`, only in case the target is of type `long`, otherwise the first step of the conversion is to the type `int`. The first step of conversion is done by rounding towards zero in the target integral type, e.g. if the floating-point value is `3.8`, then it would be rounded off to `3`. If the floating-point value being converted is a `NaN`, then the value would be converted to 0. In case of converting to `long` if the floating-point value is greater than the value of the maximum long value or it is a positive infinity, then the converted value will be the maximum long value. Similarly if the floating-point value is less than the value of the minimum long value (the most negative value) or is a negative infinity, then the converted value will be the minimum long value. Similarly, when converting to the `int` value in step one, the NaN value will be converted to 0. Positive infinity and values greater than the maximum `int` value will be converted to the maximum `int` value, and the negative infinity and values less than the minimum `int` values will be converted to the minimum `int` value.

Mixed conversion: A mixed conversion is the conversion that involves first a widening conversion followed by a narrowing conversion. For numeric types, the following are the cases for mixed conversions:

- `char` to `byte` or `short`
- `short` to `char`
- `byte` to `char`

The mixed conversions involve the conversion of `char` with the other two sub-types of `int`. In these conversions, first a widening conversion is done to the `int` type and then the narrowing conversion is done to the target type.

4.1.1.2 unary + and −

The first two arithmetic operators given earlier, i.e. the + and the −, are also used as unary operators with numeric operands only. If the operands are smaller than `int`, then the operands are first converted to `int` and the resulting expression will also be of type `int`. In other cases, there is no conversion of the operand, and the expression type is the same as the operand type. The + operator keeps the same value of the numeric operand after the possible conversion as mentioned earlier. The − operator changes the sign of the value. Note the special cases when using − with the minimum value of the integer and the long types. In these cases, the result is the same value as the value of the operand. It does not change the sign, i.e. `int i = -2147483648; System.out.println(-i);` would print the same value as the value of `i`.

4.1.2 String Concatenation

```
+
```

String concatenation is done whenever any one of the two operands is a `String` type.

```
5 + 5 + " is ten" results in "10 is ten" and
"Fifty five is " + 5 + 5, results in "Fifty five is 55"
```

4.1.2.1 String Conversion

Therefore, if any of the two operands for a + operator is a `String`, the other operand would be automatically converted to a `String` type, and then the `String` concatenation would be carried out using the two strings. Every data type in Java can be converted to a `String`. The result of `String` concatenation is a new instance of `String`. The `String` concatenation first involves the string conversion of the non-string operand to the `String` type and then carrying out the `String` concatenation. The string conversion of the `null` results in a string "null". The string conversion of the floating-point values of positive infinity, negative infinity and NaN result in "Infinity", "-Infinity" and "NaN", respectively. The string conversion for the boolean types `true` and `false` results in "true" and "false", respectively.

4.1.3 Relational Operators

The following are the relational operators in Java:

```
< > <= >= == !=
```

The result of all the relational operators is always `boolean`. The first four operators can have only numeric operands. Among the numeric types, the floating-point data types have a special representation for the infinities and Not-a-Number. In IEEE-754, there are representations for a positive infinity (a positive floating-point value divided by zero), a negative infinity (a negative floating-point divided by zero) and a set of bit representations for the Not-a-Numbers,

as shown in Figure 3.2. When comparing numeric values involving infinities, the positive infinity is always considered to be greater than any other numeric value, and the negative infinity is always considered to be less than any other numeric value. When comparing numeric values involving the NaN, the result is always `false` except in case of comparision using the inequality operator `!=`, which would always be true.

The last two relational operators `==` and `!=` can have any kind of operands, provided they are related types, i.e. one of them has to be assignable to the other. In this case, the sub-type is converted to the super-type. We can neither compare a numeric value with a `boolean` or a reference data type for equality (`==`) or inequality (`!=`) nor can we compare a `boolean` with a reference data type for equality (`==`) or inequality (`!=`). Even among the reference types, we cannot compare for equality or inequality between unrelated types, e.g. we cannot compare a `String` with a `Color` for equality or inequality using the `==` or the `!=` operators. We can compare an expression of type `Vehicle` with `Car`, where `Vehicle` is a super-class of `Car`.

The equality and inequality for reference types only check if the two reference expressions refer to the same instance or not. It is not checking if the two reference expressions refer to instances that are equivalent. The equivalence of two instances is discussed in Chapter 8, Section 8.1.1. If we have two variables `s1` and `s2` of `String` class as given in the code below:

```
String s1 = new String("Hello");
String s2 = new String("Hello");
```

then the expression `s1 == s2` results in a `false` value, since `s1` and `s2` both refer to distinct instances even though the instances referred by `s1` and `s2` are equivalent.

4.1.4 Logical Operators

The following are the logical operators:

```
&  |  ^  !  &&  ||
```

All the logical operators take only boolean operands and the result of the expression is also boolean. The `!` is a unary operator, and all others are binary operators.

The first three operators are also used with integral operands, which is discussed in Section 4.1.5.

The `&` operator performs a logical and operation, i.e. it evaluates to `true` only if both the operands evaluate to `true`, otherwise it evaluates to `false`. The `|` operator performs a logical or operation, i.e. it evaluates to `true` if either of the two operands is `true`. It would evaluate to `false` only if both the operands evaluate to `false`. The `^` is an XOR operation. It evaluates to `true` only if exactly one of the two operands is `true`. If both operands evaluate to `true` or both operands evaluate to `false`, then the result of the XOR operation is false.

The `!` unary operator reverses the `boolean` value. Therefore, if the operand evaluates to `false` then the `!` operator will evaluate it to `true`, and vice versa.

The last two operators `&&` and `||` are also known as short-circuit operators. When using these operators, the second operand does not get evaluated, in case the boolean result of the expression can be determined from the evaluation of the first operand. Therefore, in case of `&&` operator, if the first operand evaluates to `false`, then the second operand does not get evaluated. Similarly, in case of the `||` operator, if the first operand evalutes to `true`, then the second operand is not evaluated.

4.1.5 Bitwise Operators

The following are the bitwise operators:

```
&  |  ^   ~  <<  >>  >>>
```

All the bitwise operators take only integral operands (`byte`, `short`, `int`, `long` or `char`), and the result of the expression is either `int` or `long`. The result would be `long` only in case any of the operands is a `long`. The `~` is a unary operator, and all others are binary operators.

The last three operators <<, >> and >>> are also known as shift operators. These operators result in shifting bits of the first operand by the count evaluated from the second operand. The number of bits shifted in a single shift operation cannot be greater or equal to the number of bits in the first operand. The first operand is either a 32-bit or a 64-bit value. So, the number of bits that can be shifted by this operator can be a maximum of 32 bits if the first operand is an `int` or 64 bits if the first operand is a `long`. The number of bits shifted is the modulus of the value of the second operand and the number of bits in the first operand. So, if we have the first operand of type `long` and the second operand's value is 67, then the number of bits shifted will be taken as (67% 64 = 3), 64 is the number of bits in `long`. In this case, shifting will be done by only 3 bit positions.

The << operator shifts the bits in the first operand in the left direction, filling the least significant bits with zeros. This normally results in the value of the operand being doubled, unless the sign bit changes because of the loss of the most significant bits. The >> operator shifts the bits in the first operand in the right direction, filling the most significant bits with the sign bit. This results in halving the value for each bit shifted towards right. The >>> operator shifts the bits in the first operand in the right direction, filling the most significant bits with zeros. So, in case the count of the bits shifted is non-zero then this would result in a positive value.

4.1.6 Increment–Decrement Operators

The following are the increment–decrement operators:

```
++  --
```

The ++ and -- are unary operators. In Java, unlike `C`, where these operators work only for `int` and its sub-types, these operators work on any numeric type, i.e. these operators are available even for the floating-point types. Also the resulting expression is of the same type as the operand, i.e., in case the operand is of type `byte`, then the result is also a `byte` type. There is no conversion of the operand in this case. These operators simply increment or decrement the numeric value by one. Like `C`, in Java also these operators may be used in post- or pre-positions, with a similar effect as in `C` language.

4.1.7 Conditional Operator

The following is the conditional operator:

```
?:
```

This is a ternary operator and has the following syntax:

```
<boolean-expression>?<expression-1>:<expression-2>
```

The first operand has to be a boolean expression. The expressions in the second and the third operands must be of a related type, i.e. one of the operands must be assignable to the other, i.e. they have a super-type and sub-type relationship. The result of the expression is the super-type of the second and the third operands. This was the rule upto Java 1.4. From Java 5 onwards, the second and the third operands could be of any types. The result of the expression would be the least common super-type. The Object class is considered to be the highest super-type of all the data types in this case. When there is a mix of primitive and reference types, then the boxing and unboxing conversions (discussed in Section 10.6.2) available from Java 5 onwards would be applied such that both are of the same time and the result would be of the converted type.

4.1.8 Assignment Cast and instanceof Operators

The following are the assignment, cast and the instanceof operators:

```
= += -= *= /= &= |= ^= %= <<= >>= >>>=
() instanceof
```

In the above list of operators, the first row shows the various assignment operators. Except for the first assignment operator = all assignment operators are of the form op=, where op is some operator. The types of operands of these op= operators have to be according to the types required for the op operator. All the assignment operators are binary, and the first operand, also known as the *target*, has to be a variable only. The type of the target has to be the same or a super-type of the type of expression for the second operand. There is one exception to this; in case of the += operator, the target could be of type String, in which case the string conversion will be done for the second operand and will result in a string concatenation and assignment.

In cases where we have some expression for the right operand, where the type is available as a super-type of the target operand, then it is very common to use the cast operator () to convert the type of any expression to one of its sub-type.

```
Down casting (<sub-type-name>)<expression>
Up casting (<super-type-name>)<expression>
```

Downcasting of the second operand in an assignment is very common. This is done in case the expression for the second operand is a super-type of the first operand. Java is a strongly typed language, i.e. the type of any expression is always known at the compile time. The cast operator is used to change the type of an expression to one of its sub-type (downcasting) or one of its super-type (upcasting). Downcasting in primitive types can result in loss of the original value, whereas in case of upcasting there can be loss of precision as discussed in the section on conversion of numeric types. Downcasting for the reference types can result in an error at the runtime, if at the runtime the expression being cast does not refer to the instance of the same type as the cast type. Let us look at an example to understand the casting.

Let us say we have a declaration of primitive types as given in the code below:

```
int i = 75;
byte b = (byte)i;
```

```
int i = 375;
byte b = (byte)i;
```

In this case the cast at the second line does not result in any loss of value for the code on the left-hand side, whereas the cast used in the right-hand side results in loss of the original value.

Similarly, let us consider some example for reference data types. Here let us assume that we have a class called `Vehicle` having two sub-classes `Car` and a `Scooter`. In addition, we have the codes using the cast operator as given below:

```
Vehicle v = new Car();          Vehicle v = new Scooter();
Car c = (Car)v;                 Car c = (Car)v;
```

In this case the cast at the second line for the code on the left-hand side has no errors, whereas the code on the right-hand side would compile successfully, but would result in an error at the runtime, because the reference expression that is being cast does not refer to the type it has been cast into. This casting conversion would fail at runtime.

Here for the code on the right-hand side, it might be a better idea to check if the reference variable `v` refers to an instance of type `Car` or not before making a cast at runtime.

The `instanceof` operator is used with reference data types. The first operand is the reference expression and the right-hand operand is the name of the reference type. The result of the expression is always a boolean value, e.g. (v instanceof Car) is a boolean expression, where the boolean result is determined based on the actual instance being referred to by the variable `v` at runtime. Let us try to look at the assignment and the `instanceof` operators for the reference data types using some examples.

Here let us assume a class called `Date` and a sub-class of `Date` called `Time` and assume the following variable declarations:

```
Date d1, d2;
Time t1, t2;
```

Now which of the following assignments would be valid?

```
1 d1 = new Date();
2 d2 = new Time();
3 t1 = new Date()
4 t2 = new Time();
```

Here, Line 3 would not compile, as in this case the right-hand type is an expression of type `Date`, which is the super-type of the target. So, let us assume now that Line 3 has been commented and let us look at the evaluation of the following boolean expressions (using the `instanceof` operator):

```
1 (d1 instanceof Date)
2 (d1 instanceof Time)
3 (d2 instanceof Date)
4 (d2 instanceof Time)
5 (t2 instanceof Date)
6 (t2 instanceof Time)
```

In the above listing, Line 2 would result in a `false` value, whereas all other expressions evaluate to the `true` value. The `instanceof` check is done at runtime and is not based on the declaration type of the variable. Also all instances of `Time` are instances of the `Date` class since `Date` is a super-class of `Time`.

In the above code only the variable `d1` refers to the instance of the `Date`, and `d2` and `t2` both refer to instances of `Time`. So, the `instanceof` check for `d2` and `t2` would result in `true` for both `Date` and `Time`. Since the variable `d1` refers to an instance of the `Date`, its `instanceof` check with `Time` would result in `false`.

The `instanceof` check for a `null` value would always result in a `false` value.

4.1.9 Other Operators

```
new [] .
```

There are a few other operators as listed above.

The new operator is always used to create a new instance of some class or for allocating an array of any data type. This can result in an error at the runtime if there is not enough memory to do the allocation for the instance being created.

The `[]` operator is used to access the elements in an array. This operator requires an `int` expression. There can be an error at the runtime if the value of the expression used to access the element is not in the range from 0 to the length of the array − 1.

The `.` operator is used to access any member of an instance or invoking methods on an instance. This can also result in an error at the runtime in case we try to access any member for the `null` value.

Let us look at some of the commonly used statements in Java.

4.2 STATEMENTS

In Java we have the two common kinds of flow statements—conditional statements and loop statements. The `if`, `if-else` and the `switch-case` are the conditional statements. The `for`, `while` and the `do-while` are the loop statements. We have the `break` and the `continue` statements, which are used inside the loop and the switch-case statements.

4.2.1 Condition Statements—`if`, `if-else` and `switch-case`

The `if` and the `if-else` statements in Java work similar to C. The only difference is that unlike in C language, where there is no separate boolean type, the conditions in the `if` statement would accept any numeric value and the zero or non-zero decides whether the condition is considered to be false or true, In Java, we always need to give a boolean expression for the condition. A numeric expression cannot be used for the condition.

The switch-case in Java is also similar to C. From Java 5 onwards, an enum type (discussed in Section 12.4) can also be used in the switch statement, with the cases being the instances of the corresponding enum type only.

4.2.2 Loop Statements—`for`, `while` and `do-while`

The usage of the loop statements is also similar to C.

for-each: The for-each loop is introduced from Java 5 onwards. It could be used to iterate over all the elements of an array. This loop does not involve specifying any kind of an index value in the loop. In the loop, the element of the array is available using some local variable for the loop. Say for e.g. we want to write a method that takes an array of `double` and returns the sum of the elements in the array. It could be written as shown in Listing 4.2.

Listing 4.2. Example of for-each

```
1 public double sum(double[] values) {
2     double sum = 0;
3     for (double value : values) {
4         sum += value;
5     }
6     return sum;
7 }
```

In a for-each loop, we mention the following:

(a) A local variable declaration whose type is the same as the type of the array element we want to iterate.

(b) The array whose elements we want to iterate.

These two things are separated by a ':' character. In Listing 4.2 in Line 3, we declare a local variable called the `value` of type double and the array `values`, which we want to iterate over. The local variable that is declared makes the next value in the array available to us. This variable cannot be used to update the element in the array.

The for-each does not give anything extra compared to the normal for loops that we normally code. It is only making the code for iterating over elements of an array a little simpler. In fact the normal for loop where we use indexes is more flexible. The for-each loop is simple to use if we are simply interested in reading the values from an array and processing them. These cannot be used to manipulate values in the array.

4.2.3 `break`, `continue` and `return`

The various loop statements execute statements iteratively where the condition of repeating the iteration is taken either at the beginning of the loop or at the end, and then it is decided whether the iteration is to be repeated or not. Sometimes, we may have a condition within the body of the loop where we may like to terminate the loop and continue with the statements after the loop statement. The `break` statement allows to break out of a loop statement from within the body of the loop statement. The `break` statement is also used to break out of the body of a switch statement. Sometimes, we may have a condition within the body of the loop statement where we would not like to proceed with the next statements within the loop, but we may like to continue directly with the next iteration of the loop. The `continue` statement within the body of a loop transfers control back to the top or the bottom of the loop where the condition to continue with the next iteration of the loop may be checked and then the loop can continue.

The return statement is used inside a method to return a value and not to proceed further inside the method. The return is followed by an expression that matches the return type of the method. In case the method has a return type of void, then there is no expression following the return statement.

From Balaguru - pN 119

Labelled break and continue: The break statement is normally used to come out of a loop statement or from a switch-case statement. The continue statement is used in a loop statement to start continuing with the next iteration by finishing the current iteration immediately. When we have a nested loop statement or when we have the nesting of a switch-case and loop statements in various combinations and if we use the break statement, then which loop or switch-case statement does this break apply to? By default, it would apply to the innermost loop or the switch-case statement.

The break and continue statements can be used with labels to indicate the statement to which they are applicable, e.g. if we have a nested loop structure as given below:

```
1 firstFor:
2     for(...) {
3         ... // some statements
4 secondFor:
5         for(...) {
6             // some statements
7             if (...) {
8                 break ...;
9             }
10             // some statements
11         }
12         // some statements
13     }
```

In this code segment, we have used labels in Lines 1 and 4. The labels can be used at any point within a method. The label in Line 1 gives a label for the statement which immediately follows it; in this case the for loop statement starting in Line 2 has been labelled firstFor and the second for loop statement in Line 5 has been labelled secondFor. Now the break statement in Line 8 can be followed by a label, e.g. in Line 8 we could write break firstFor; or break secondFor;. Similarly a continue statment can also be followed by a label, which is a label for a loop statement. Therefore, in case of nested statements involving loop statements, a break and continue statment can also specify the target statement, which needs to be broken or which needs to continue with the next iteration.

LESSONS LEARNED

- In Java, all operators result in an expression of a type, which is known at the compile time, i.e. Java is a strongly typed language.

- Widening conversion is a conversion of a sub-type to its super-type, whereas narrowing conversion is a conversion of a super-type to one of its sub-type.
- Widening conversions are mostly implicit, whereas narrowing conversions are normally achieved using the cast operator.
- The + operator is used both for numeric addition as well as the string concatenation. A string concatenation occurs if one of the operands is of type `String`, if the other operand is not a `String`, then a string conversion also takes place for the other operand.
- The widening conversion of an `int` to a `float` or of a `long` to a `float` or `double` can result in a loss of precision.

EXERCISES

1. State which of the following are true or false:
   - **(a)** `goto` can be used to jump to a statement in Java.
   - **(b)** The `break` statement is required in the `default` case of a `switch` selection structure.
   - **(c)** The `default` case is required in the `switch` selection structure.
   - **(d)** A for-each statement can be used to iterate over all the elements of an array.
   - **(e)** Inside a for-each loop, the current index value is available by using the keyword index.
   - **(f)** A `boolean` value can be cast to an `int`.
   - **(g)** `instanceof` is a keyword in Java.
2. Fill in the blanks:
   - **(a)** The result of adding 2 `byte` values will be of type _____.
   - **(b)** The _____ statement is used in Java to make a conditional transfer of control based on the value of `int` type.
   - **(c)** The `instanceof` operator results in the value of type _____.
   - **(d)** Precision loss can occur when converting a _____ value to type `double`.
3. Explain the narrowing, widening and the mixed conversion for numeric types in Java.
4. Explain the difference between the short-circuit operators and their normal counter parts.
5. Explain the use of labelled `break` and `continue` statements.
6. What is the output when the following code is compiled and run?

```
1 public class Test {
2     public static void main(String[] args) {
3         float f1 = -75;
4         float f2 = 0;
5         System.out.println(f1 / f2);
6     }
7 }
```

(a) Compilation error
(b) Runtime error in Line 5
(c) Infinity
(d) –Infinity
(e) NaN
(f) None of the above