NAMING CONVENTIONS AND DATA TYPES

CHAPTER 4

In the previous chapter, we understood Java by writing a couple of programs. In those programs, you might have observed that sometimes we have used small letters and sometimes capital letters. For example, we have written the class `System` starting with a capital letter. You cannot write this class name as `system` or `SYSTEM`. Since Java is a case sensitive programming language, it recognizes capital and small letters as different. So the programmer should take care of upper and lower case while writing a program in Java. But how to know where to use which case—upper or lower? For this purpose, certain conventions (rules) are followed by JavaSoft people while naming the variables, classes, methods, etc. These naming conventions should be followed by every programmer in his programs for maintaining uniformity and also for clarity of distinction between different elements of a program. These rules also reduce the possibility of any spelling mistakes while writing the names of variables, classes, etc. in the programs. For example, a Java program will not compile, if `main()` method is written as:

```
Public static void Main(String args[])
```

The problem in the preceding statement is using capital `P` for public and capital `M` for `main()`. Such errors can be eliminated if we follow the naming conventions.

Naming Conventions in Java

Naming conventions specify the rules to be followed by a Java programmer while writing the names of packages, classes, methods, etc. Now, let us see some of the major naming conventions to be followed in Java.

- A *package* represents a sub directory that contains a group of classes and interfaces. Names of packages in Java are written in small letters as:

```
java.awt
java.io
javax.swing
```

- A *class* is a model for creating objects. A class specifies the properties and actions of objects. An interface is also similar to a class. Each word of class names and interface names start with a capital letter as:

```
String
DataInputStream
ActionListener
```

- A class and an interface contain methods and variables. The first word of a method name is in small letters; then from second word onwards, each new word starts with a capital letter as shown here:

```
println()
readLine()
getNumberInstance()
```

- The naming convention for variables names is same as that for methods as given here:

```
age
empName
employee_Net_Sal
```

Note

Now, the question that is commonly asked is that if same rule is applied for both variables and methods, how can we distinguish between them? The answer is that since a method's name ends with a pair of simple braces (), it can be distinguished easily from a variable whose name will not have any braces.

- *Constants* represent fixed values that cannot be altered. For example, PI is a constant whose value is 22/7 or 3.14159, which is fixed. Such constants should be written by using all capital letters as shown here:

```
PI
MAX_VALUE
Font.BOLD
```

Here, BOLD is a constant in Font class. This is the way most of the inbuilt constants in Java are referenced.

- All keywords should be written by using all small letters as follows:

```
public
void
static
```

Data Types in Java

We know we need variables to store the data. Internally, a variable represents a memory location which holds data. When we want to use a variable in a program, we should first declare it as:

```
int x;
```

Here, we are declaring that x is a variable, which can store int (integer) type data. This means int is representing the nature of data to be stored into x. int is also called a data type. For example x can store an integer number like 125 as:

```
x = 125;
```

Here, x is a variable and = represents that the value 125 is stored into x. This value 125 stored into x is also called literal. There are various data types and literals defined in Java, which we will be discussing in this chapter.

Integer Data Types

These data types represent integer numbers, i.e. numbers without any fractional parts or decimal points. For example, 125, -225678, 0, 1022, etc. come under this category. Integer data types are again sub divided into byte, short, int, and long types. Table 4.1 lists the data types.

Table 4.1

Data type	Memory size	Minimum and Maximum values
Byte	1 byte	-128 to +127
Short	2 bytes	-32768 to +32767
Int	4 bytes	-2147483648 to +2147483647
Long	8 bytes	-9223372036854775808 to +9223372036854775807

Let us try to understand this through an example given here:

```
byte rno=10;
```

In the preceding statement, we are declaring byte data type for the variable rno and the value 10, which is stored into rno. byte represents any value between -128 to +127.

long x=150L;

Here, 150 is stored into x, which is declared as long type. Notice the L at the end of the statement. If this L is not there, then JVM allots only 2 bytes of memory to x as against the usual 8 bytes memory that should be allotted to a long type. The reason for this is that 2 bytes are sufficient to store the value 150. But if we attach l or L at the end of the value as shown in the preceding example, then JVM will consider it as a long value and will allot 8 bytes to it.

Float Data Types

These data types are useful to represent numbers with decimal point. For example, 3.14, 0.0012, -123.11, etc. are called floating point numbers. These are again classified as float (single precision floating point number) and double (double precision floating point number). The difference exists essentially in the number of digits, they can represent accurately after the decimal point. This accuracy is also called *precision*. Table 4.2 depicts the size of float and double.

Table 4.2

Data type	Memory size	Minimum and Maximum values
Float	4 bytes	-3.4e38 to -1.4e-45 for negative values and 1.4e-45 to 3.4e38 for positive values
Double	8 bytes	-1.8e308 to -4.9e-324 for negative values and 4.9e-324 to 1.8e308 for positive values

Important Interview Question

What is the difference between float and double?

`Float` can represent up to 7 digits accurately after decimal point, whereas `double` can represent up to 15 digits accurately after decimal point.

Let us now look at an example given here:

```
float pi=3.142F;
```

Here, the variable `pi` is containing the value `3.142`. If `F` is not written at the end, then JVM would have allotted 8 bytes assuming the value to be `double`. The reason for this is that in `float` and `double` data types, the default is taken as `double`. By attaching `F` or `f`, we can ask the JVM to consider it as a `float` value and allot only 4 bytes.

`double distance=1.98e8;`

Here, `e` or `E` represents `X 10 to the power`. Hence, `1.98e8` means `1.98X108`. This is also called *scientific notation* of representing numbers.

Character Data Type

This data type represents a single character like a, `P`, `&`, `*`, etc. Table 4.3 shows `char` data type details.

Table 4.3

Data type	Memory size	Minimum and Maximum values
Char	2 bytes	0 to 65535

Here is an example of `character` data type:

```
char ch='X';
```


Here, we are storing the single character 'X' into the variable `ch`. Here, 'X' is also called *character literal*. Whenever character literals are written, they should be enclosed inside the single quotes.

By observing the minimum and maximum values in the table earlier, we shall get a doubt regarding why the range is expressed in integer numbers (`0` to `65535`). We know that all the characters on the keyboard are translated into integer values called ASCII (American Standard Code for Information Interchange), which is a standard maintained by every keyboard manufacturer. The processor recognizes the character uniquely from its ASCII value. Hence, let us understand the range mentioned in the table is nothing but the ASCII value range only.

The ASCII value range, from `0` to `65535`, given in the table can uniquely represent a total of `65536` characters. This means a total of 65536 distinct characters can be recognized by Java. But we never use these many characters since our keyboard contains English alphabets and some other characters whose total does not exceed `256`. This means 1 byte is sufficient to represent all the available characters of the keyboard. Then why 2 bytes are used to represent the `char` data type in Java?

JavaSoft people wanted to provide a facility to include characters not only from English but also from all other human languages to be used in Java programs. This will enable the programmers to write and execute a Java program in any language, which becomes an advantage on Internet. This system is also called *Unicode system*. Unicode uses 2 bytes so that any character from any language can be encoded successfully.

Important Interview Question

What is a Unicode system?

Unicode system is an encoding standard that provides a unique number for every character, no matter what the platform, program, or language is. Unicode uses 2 bytes to represent a single character.

String Data Types

A `String` represents a group of characters, like New Delhi, AP123, etc. The simplest way to create a `String` is by storing a group of characters into a `String` type variable as:

```
String str = "New Delhi";
```

Now, the `String` type variable `str` contains "`New Delhi`". Note that any string written directly in a program should be enclosed by using double quotes.

There is a class with the name `String` in Java, where several methods are provided to perform different operations on strings. Any string is considered as an object of `String` class. But in C/C++, a string is considered as a character array containing some characters where the last character would be `\0`. This is not valid in Java, since in Java, we got strings and character arrays both separately.

Now the question arises that if `String` is a class, why are we taking it as a data type? The answer is that every class is a data type and is also called *user-defined data type*.

Boolean Data Types

`Boolean` data types represent any of the two values—true or false. JVM uses 1 bit to represent a boolean value internally, for example:

```
boolean response=true;
```

As shown earlier, we should not enclose the boolean value `true` (or `false`) in any quotation marks. In C/C++, 0 represents false and any other number represents true. This is not valid in Java.

Literals

A *literal* represents a value that is stored into a variable directly in the program. See the following examples:

```
boolean result = false;
char gender = 'M';
short s = 10000;
int i = -1256;
```

In the preceding statements, the right hand side values are called literals because these values are being stored into the variables shown at the left hand side. As the data type of the variable changes, the type of the literal also changes. So we have different types of literals. These are as follows:

- Integer literals
- Float literals
- Character literals
- String literals
- Boolean literals

Read on to get familiar with them.

Integer Literals

Integer literals represent the fixed integer values like 100, -55, 123450, etc. All these numbers belong to decimal system, which uses 10 digits (from 0 to 9) to represent any number. Suppose, we want to write an integer in octal number system (octal number system uses 8 digits, from 0 to 7), then we should prefix *0* before the number. To represent hexadecimal number (hexadecimal number system uses 16 digits from 0 to 9 and from A to F), we should prefix *0x* before the value.

```
int decVal = 26;   // The number 26, in decimal
int octVal = 032;  // The number 26, in octal
int hexVal = 0x1a; // The number 26, in hexadecimal
```

Float Literals

Float literals represent fractional numbers. These are the numbers with decimal points like 2.0, -0.005, 3.14, 6.1e-22, etc. which should be used with float or double type variables. While writing these literals, we can use E or e for scientific notation, F or f for float literal, and D or d for double literal (this is the default and generally omitted).

```
double d1 = 123.4;
double d2 = 1.234e2; // same value as d1, but in scientific notation
float f1  = 123.4f;
```

Character Literals

Character literals indicate the following:

- General characters, like A, b, 9, etc.
- Special characters, like ?, @, etc.
- Unicode characters, like \u0042 (this represents *a* in ISO Latin 1 character set).
- Escape sequence (backslash codes) like \n, \b, etc.

Character literals should be enclosed in single quotation marks. The preceding unicode characters and escape sequence can also be represented as strings.

String Literals

String literals represent objects of String class. For example, Hello, Anil Kumar, AP1201, etc. wil come under string literals, which can be directly stored into a String object.

Boolean Literals

Boolean literals represent only two values—true and false. It means we can store either true or fals into a boolean type variable.

Conclusion

By following naming conventions, a programmer can avoid a lot of mistakes in writing the names c packages, classes, methods, variables, etc. This will improve readability as well as understandabilit of a program. So maintaining naming conventions is a good programming habit.

A program generally acts on data, processes it, and provides the results. So data is very importan element of a program. In this chapter, we also discussed different *data types*, in terms of type c data received, stored, or used in a program.

CHAPTER 5

OPERATORS IN JAVA

A programmer generally wants to do some operations in a program. For example, in a program, you want to perform addition of two numbers. How can you do it? Just by using + symbol, we can add the numbers. This means + is a symbol that performs an operation, i.e. addition. Such symbols are called *operators* and programming becomes easy because of them. Let us discuss about operators in this chapter and how to use them with examples.

Operators

An *operator* is a symbol that performs an operation. An operator acts on some variables, called *operands* to get the desired result, as shown in Figure 5.1.

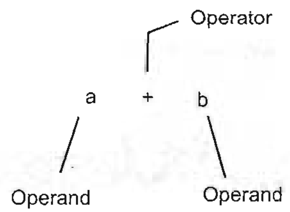


Figure 5.1 Operator and operands

If an operator acts on a single variable, it is called *unary operator;* if it acts on two variables, it is called *binary operator;* and if it acts on three variables, then it is called *ternary operator*. This is one type of classification. Let us now examine various types of operators in detail.

Arithmetic Operators

These operators are used to perform fundamental arithmetic operations like addition, subtraction, etc. There are 5 arithmetic operators in Java. Since these operators act on two operands at a time, these are called *binary operators*. Table 5.1 displays the functioning of these operators. Here, we are assuming the value of `a` as `13` and `b` as `5`.

Table 5.1

Operator	Meaning	Example	Result
+	Addition operator	a + b	18
-	Subtraction operator	a – b	8
*	Multiplication operator	a * b	65
/	Division operator	a / b	2.6
%	Modulus operator (This gives the remainder of division)	a % b	3

Addition operator (+) is also used to join two strings, as shown in the following code snippet:

- `String s1= "wel";`
- `String s2= "come";`
- `String s3= s1+s2;  //here, '+' is joining s1 and s2.`

Now, we get `welcome` in s3. In this case, + is called *String concatenation operator.*

Unary Operators

As the name indicates, unary operators act on only one operand. There are 3 kinds of unary operators:

- Unary minus operator (–)
- Increment operator (++)
- Decrement operator (--)

Unary Minus Operator (-)

This operator is used to negate a given value. Negation means converting a negative value into positive and vice versa, for example:

```
int x = 5;
System.out.println(-x);      will display -5.
System.out.println(-(-x));  will display 5.
```

In this code snippet, the unary minus (-) operator is used on variable x to negate its value. The value of x is 5 in the beginning. It became -5 when unary minus is applied on it.

Increment Operator (++)

This operator increases the value of a variable by 1, for example:

```
int x = 1;
++x   will make x = 2
x++   now x = 3
```

Here, the value of the variable x is incremented by 1 when ++ operator is used before or after it. Both are valid expressions in Java. Let us take an example to understand it better:

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

In this statement, if x is 3, then x+1 value will be 4. This value is stored again in the left hand side variable x. So, the value of x now becomes 4. The same thing is done by ++ operator also.

Writing ++ before a variable is called *pre incrementation* and writing ++ after a variable is called *post incrementation*. In pre incrementation, incrementation is done first and any other operation is done next. In post incrementation, all the other operations are done first and incrementation is done only at the end. To understand the difference between pre and post incremenations, let us take a couple of examples.

Example 1: Finding the difference between pre- and post- increment of x

```
int x=1;
System.out.println(x);
System.out.println(++x);
System.out.println(x);
```

```
int x=1;
System.out.println(x);
System.out.println(x++);
System.out.println(x);
```

Output:

```
1
2
2
```

Output:

```
1
1
2
```

In this example, see the left-hand side and right-hand side statements. The second statement on the left uses pre-incrementation, while the second statement on the right uses post-incrementation.

At the left hand side:

- `System.out.println(x);` // displays the value of x as 1
- `System.out.println(++x);` // first increments the value of x and then displays it as 2
- `System.out.println(x);` // displays the value of x, which is already incremented, i.e. 2

At the right hand side:

- `System.out.println(x);` // displays the value of x, i.e. 1
- `System.out.println(x++);` // first displays the value of x as 1 and then increments it.
- `System.out.println(x);` // displays the incremented value of x, i.e. 2

Example 2: Finding the difference between pre- and post- increment of a and b

```
a=1;
b=2;
a=++b;
What are the values of a and b?
```

```
a=1;
b=2;
a=b++;
what are the values of a and b?
```

Result:

```
a=3
b=3
```

Result:

```
a=2
b=3
```

At the left hand side, `a=++b;`

This is called pre-incrementation. So increment the value of b first (it becomes 3) and then store it into a. Now, the value of a also becomes 3.

At the right hand side, `a=b++;`

This is called post-incrementation. So incrementation of b will not be done first. Without incrementing, the value of b, i.e. 2 is stored into a (so the value of a becomes 2) and then incrementation of b is done (so the value of b becomes 3).

Example 3: Finding the value of the following expression, given that the value of a is 7

`++a*a++;`

Here, the value of a is given as 7. So ++a makes the value of a as 8. Then a++ is there. Since, this is post-incrementation, the value of a will not be incremented in the same statement and hence the value of a, i.e. 8 will stay like that only. As a result, we get 8*8= 64.

Decrement Operator (--)

This operator is used to decrement the value of a variable by 1.

```
int x = 1;
--x   will make the value of x as 0
x--   now, the value of x is -1
```

This means, the value of x is decremented every time we use -- operator on it. This is same as writing x= x-1.

Writing -- before a variable is called *pre-decrementation* and writing -- after a variable is called *post-decrementation*. Like the incrementation operator, here also the same rules apply. Pre-decrementation is done immediately then and there itself and

Post-decrementation is done after all the other operations are carried out.

Assignment Operator (=)

This operator is used to store some value into a variable. It is used in 3 ways:

- It is used to store a value into a variable, for example `int x = 5;`
- It is used to store the value of a variable into another variable, for example:

```
int x = y;  //here the value of y is stored into x
```

- It is used to store the value of an expression into a variable, for example:

```
int x = y+z-4; //here the expression y+z-4 is evaluated and its result is
//stored into x.
```

Note

We cannot use more than one variable at the left hand side of the = operator. For example:

```
x+y = 10;  //this is invalid, since there is doubt for the compiler
           //regarding where the value 10 is stored.
```

We cannot use a literal or constant value at the left side of the = operator. For example:

```
15 = x;   //how can we store the value of x into a number?
```

Compact Notation

While using assignment operator (=), sometimes we may have to use same variable at both the sides of the operator. In such cases, we can eliminate repetition of the variable and use compact notation or short cut notation, as shown in Table 5.2.

Table 5.2

Expanded notation	**Compact notation**	**Name of operator**
`x = x + 10`	`x +=10`	+= is called addition assignment operator
`sal = sal * 10.5`	`sal *=10.5`	*= is called multiplication assignment operator
`value= value-discount`	`value -= discount`	-= is called subtraction assignment operator
`p = p / 1000`	`p /= 1000`	/= is called division assignment operator
`num = num % 5.5`	`num %= 5.5`	%= is called modulus assignment operator

Experienced programmers use compact notations. However, both the expanded and compact notations are valid in Java.

Relational Operator

These operators are used for the purpose of comparing. For example, to know which one is bigger or whether two quantities are equal or not. Relational operators are of 6 types:

- > greater than operator
- >= greater than or equal to
- < less than operator
- <= less than or equal to
- == equal to operator
- != not equal to operator

The main use of relational operators is in the construction of conditions in statements, like this:

```
if(condition_is_true) statement_to_be_ executed.
```

This statement could be applied in a program as follows:

```
if( a > b) System.out.println(a);
if( a == 100) System.out.println("a value equals to 100");
```

Observe, that in this example the two statements, `a>b` and `a==100`, are conditions. If they are true, then the corresponding statements will be executed. Note that = (assignment operator) is for storing the value into a variable and == (equal to operator) is for comparing the two quantities. Both are quite different.

Logical Operators

Logical operators are used to construct compound conditions. A compound condition is a combination of several simple conditions. Logical operators are of three types:

- ❑ `&&` *and* operator
- ❑ `||` *or* operator
- ❑ `!` *not* operator

```
if( a == 1 || b == 1 || c == 1 ) System.out.println("Yes");
```

Here, there are 3 conditions: `a==1`, `b==1`, and `c==1`, which are combined by `||` (*or* operator). In this case, if either of the `a` or `b` or `c` value becomes equal to 1, `Yes` will be displayed.

```
if( x > y && y < z ) System.out.print("Hello");
```

In the preceding statement, there are 2 conditions: `x>y` and `y<z`. Since they are combined by using `&&` (*and* operator); if both the conditions are true, then only `Hello` is displayed.

```
if( !(str1.equals(str2)) System.out.println("Not equal");
```

We are assuming that `str1` and `str2` are two string objects, which are being compared. See the `!` (`not` operator) and the `equals()` methods in the earlier condition telling that if `str1` is not equal to `str2`, then only `Not equal` will be displayed.

Boolean Operators

These operators act on boolean variables and produce boolean type result. The following 3 are boolean operators:

- ❑ `&`boolean *and* operator
- ❑ `|`boolean *or* operator
- ❑ `!`boolean *not* operator

Boolean `&` operator returns *true* if both the variables are *true*. Boolean `|` operator returns *true* if any one of the variables is *true*. Boolean `!` operator converts *true* to *false* and vice versa.

`boolean a,b;` //declare two boolean type variables

- ❑ `a= true;` //store boolean value true into a
- ❑ `b= false;` //store boolean value false into b

```
Decimal 10             =   0000 1010

1's complement         =   1111 0101
(change 0 as 1 and
vice versa)
                                  +1
2's complement         =   ---------
(add 1 to 1's              1111 0110   = Decimal -10
complement)                ---------
```

Figure 5.4 Representation of negative numbers

Important Interview Question

How are positive and negative numbers represented internally?

Positive numbers are represented in binary using 1's complement notation and negative numbers are represented by using 2's complement notation.

There are 7 types of bitwise operators. Read on to understand them. Let us now discuss them one by one.

Bitwise Complement Operator (~)

This operator gives the complement form of a given number. This operator symbol is ~, which is pronounced as *tilde*. Complement form of a positive number can be obtained by changing 0's as 1's and vice versa.

```
If int x = 10. Find the ~x value.
x = 10 = 0000 1010.
```

By changing 0's as 1's and vice versa, we get 1111 0101. This is nothing but -11(in decimal). So, `~x = -11`.

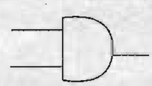
Bitwise and Operator (&)

This operator performs `and` operation on the individual bits of the numbers. The symbol for this operator is `&`, which is called *ampersand*. To understand the `and` operation, see the truth table given in Figure 5.5.

```
x = 10 = 0000  1010
y = 11 = 0000  1011
         ----------
x&y    = 0000  1010
```

x	y	x&y
0	0	0
0	1	0
1	0	0
1	1	1

Truth table



AND gate

Figure 5.5 AND operation

Truth table is a table that gives relationship between the inputs and the output. From the table, we can conclude that by multiplying the input bits, we can get the output bit. The `AND` gate circuit present in the computer chip will perform the `and` operation.

```
If int x = 10, y = 11. Find the value of x&y.
x = 10 = 0000 1010.
y = 11 = 0000 1011.
```

From the truth table, by multiplying the bits, we can get `x&y = 0000 1010`. This is nothing but 10 (in decimal).

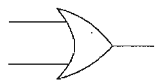
Bitwise or Operator (|)

This operator performs *or* operation on the bits of the numbers. The symbol is |, which is called *pipe* symbol. To understand this operation, see the truth table given in Figure 5.6. From the table, we can conclude that by adding the input bits, we can get the output bit. The OR gate circuit, which is present in the computer chip will perform the or operation.

```
x = 10 = 0000  1010
y = 11 = 0000  1011
-------------------
x|y    = 0000  1011
```

x	y	x|y
0	0	0
0	1	1
1	0	1
1	1	1

Truth table



.OR gate

Figure 5.6 OR operation

```
If int x = 10, y = 11. Find the value of x|y.
x = 10 = 0000 1010.
y = 11 = 0000 1011.
```


From the truth table, by adding the bits, we can get x|y = 0000 1011. This is nothing but 11 (in decimal).

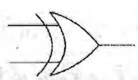
Bitwise xor Operator (^)

This operator performs *exclusive or (xor)* operation on the bits of the numbers. The symbol is ^, which is called *cap, carat,* or *circumflex* symbol. To understand the *xor* operation, see the truth table given in Figure 5.7. From the table, we can conclude that when we have odd number of 1's in the input bits, we can get the output bit as 1. The *XOR* gate circuit of the computer chip will perform this operation.

```
x = 10 = 0000  1010
y = 11 = 0000  1011
-------------------
x^y    = 0000  0001
```

x	y	x^y
0	0	0
0	1	1
1	0	1
1	1	0

Truth table



XOR gate

Figure 5.7 XOR operation

```
If int x = 10, y = 11. Find the value of x^y.
x = 10 = 0000 1010.
y = 11 = 0000 1011.
```

From the truth table, when odd number of 1's are there, we can get a 1 in the output. Thus, x^y = 0000 0001 is nothing but 1 (in decimal).

Bitwise Left Shift Operator (<<)

This operator shifts the bits of the number towards left a specified number of positions. The symbol for this operator is <<, read as *double less than*. If we write x<<n, the meaning is to shift the bits of x towards left n positions.

```
If int x = 10. Calculate x value if we write x<<2.
```

Shifting the value of x towards left 2 positions will make the leftmost 2 bits to be lost. The value of x is 10 = 0000 1010. Now x<<2 will be 0010 1000 = 40 (in decimal). The procedure to do this is explained in Figure 5.8.

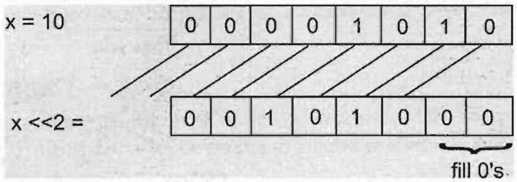


Figure 5.8 Shifting bits towards left 2 times

Bitwise Right Shift Operator (>>)

This operator shifts the bits of the number towards right a specified number of positions. The symbol for this operator is >>, read as *double greater than*. If we write x>>n, the meaning is to shift the bits of x towards right n positions.

>> shifts the bits towards right and also preserves the sign bit, which is the leftmost bit. Sign bit represents the sign of the number. Sign bit 0 represents a positive number and 1 represents a negative number. So, after performing >> operation on a positive number, we get a positive value in the result also. If right shifting is done on a negative number, again we get a negative value only.

```
If x = 10, then calculate x>>2 value.
```

Shifting the value of x towards right 2 positions will make the rightmost 2 bits to be lost. x value is 10 = 0000 1010. Now x>>2 will be: 0000 0010 = 2 (in decimal) (Figure 5.9).

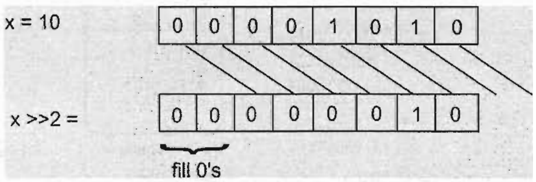


Figure 5.9 Shifting bits towards right 2 times

Bitwise Zero Fill Right Shift Operator (>>>)

This operator also shifts the bits of the number towards right a specified number of positions. But, it stores 0 in the sign bit. The symbol for this operator is >>>, read as *triple greater than*. Since, it always fills 0 in the sign bit, it is called *zero fill right shift* operator. If we apply >>> on a positive number, it gives same output as that of >>. But in case of negative numbers, the output will be positive, since the sign bit is replaced by a 0.

Important Interview Question

What is the difference between >>and >>> ?

Both bitwise right shift operator (>>) and bitwise zero fill right shift operator (>>>) are used to shift the bits towards right. The difference is that >> will protect the sign bit whereas the >>> operator will not protect the sign bit. It always fills 0 in the sign bit.

Program 1: Let us now write a program to observe the effects of various bitwise operators.

```
//Using bitwise operators
class Bits
{
  public static void main(String args[])
  {
     byte x,y;
     x = 10;
     y = 11;

     System.out.println("~x= " + (~x));
     System.out.println("x&y= "+ (x&y));
     System.out.println("x|y= "+ (x|y));
     System.out.println("x^y= "+ (x^y));
     System.out.println("x<<2= "+ (x<<2));
     System.out.println("x>>2= "+ (x>>2));
     System.out.println("x>>>2= "+ (x>>>2));
  }
}

Output:
C:\> javac Bits.java
C:\> java Bits
 ~x= -11
x&y= 10
x|y= 11
x^y= 1
x<<2= 40
x>>2= 2
x>>>2= 2
```


Ternary Operator or Conditional Operator (? :)

This operator is called *ternary* because it acts on 3 variables. The other name for this operator is *conditional operator*, since it represents a conditional statement. Two symbols are used for this operator? and:

Its syntax is `variable = expression1? expression2 expression3;`

This means that first of all, expression1 is evaluated. If it is true, then expression2 value is stored into the variable. If expression1 is false, then expression3 value is stored into the variable. It means:

```
if( expression1 is true )
   variable = expression2;
else variable = expression3;
```

Now, let us put the following condition

```
max = (a>b) ? a : b;
```

Here, (a>b) is evaluated first. If it is true, then the value of a is stored into the variable `max`, else the value of `b` is stored into `max`. This means:

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

Here, we are using 3 variables: a, b, and max—thus, the name *ternary*. The preceding statement, which is called *conditional statement,* is also represented by this operator. So it is also called *conditional operator.* Remember conditional operator is a compact form of conditional statement.

Member Operator (.)

Member operator is also called *dot* operator since its symbol is a . (dot or period). This operator tells about member of a package or a class. It is used in three ways:

- We know a package contains classes. We can use . operator to refer to the class of a package.

Syntax:

```
    packagename.classname;
```

This could be written as follows:

```
    java.io.BufferedReader      // BufferedReader is a class in the package:
                                //java.io.
```

- We know that each class contains variables or methods. To refer to the variables of a class, we can use this operator.

Syntax:

```
    classname.variablename;
```

Or

```
    objectname.variablename;
```

This could be written as:

```
    System.out    //out is a static variable in System class
          Emp.id           //id is a variable in Employee class. emp is Employee
class                            //object
```

- We know that a class also contains methods. Using dot operator, we can refer to the methods of a class.

Syntax:

```
    classname.methodname;
```

Or

```
    objectname.methodname;
```

Let us try to understand this with the help of an example.

```
    Math.sqrt()  //sqrt() is a method in Math class
    br.read()    //read() is a method in BufferedReader class. br is object of
                 //BufferedReader class.
```

instanceof Operator

This operator is used to test if an object belongs to a class or not. Note that the word *instance* means *object*. This operator can also be used to check if an object belongs to an interface or not.

Syntax:

```
boolean variable =  object instanceof class;
              boolean variable = object instanceof interface;
```

This could be written as:

```
boolean x = emp instanceof Emplyee;
```

Here, we are testing if `emp` is an object of `Employee` class or not. If `emp` is an object of `Employee` class, then `true` will be returned into `x`, otherwise `x` will contain `false`.

new Operator

`new` operator is often used to create objects to classes. We know that objects are created on *heap* memory by JVM, dynamically (at runtime).

Syntax:

```
classname  obj = new classname();
```

An example of this is as follows:

```
Employee emp = new Employee();    // emp is an object of the
                                    //Employee class
```


Cast Operator

Cast operator is used to convert one datatype into another datatype. This operator can be used by writing datatype inside simple braces.

```
double  x = 10.54;
      int y = x;  //error - because datatypes of x and y are different.
```

To store `x` value into `y`, we have to first convert the datatype of `x` into the datatype of `y`. It means double datatype should be converted into `int` type by writing `int` inside the simple braces as: `(int)`. This is called *cast* operator.

```
int y = (int)x; //here, x datatype is converted into int type and then//stored
into y.
```

In the preceding statement, `(int)` is called the cast operator. Cast operator is generally used before a variable or before a method.

Priority of Operators

When several operators are used in a statement, it is important to know which operator will execute first and which will come next. To determine that, certain rules of *operator precedence* are followed:

- First, the contents inside the braces: () and [] will be executed.

- Next, ++ and --.
- Next, *, /, and % will execute.
- + and – will come next.
- Relational operators are executed next.
- Boolean and bitwise operators
- Logical operators will come afterwards.
- Then ternary operator.
- Assignment operators are executed at the last.

Conclusion

In this chapter, you learnt that operators make programming easy by assisting the programmer to perform any operation by just mentioning a symbol. Just imagine a situation, where we have to add two numbers without any arithmetic or unary operators available to us. In such a case, a mere addition will become a very tedius job. Such programming difficulties can be escaped by using these operators.