



# Chapter 2

## ***Fundamental Elements in Java***

### **Learning Objectives**

***After completing this chapter, you will be able to understand the concept of:***

- *Data types.*
- *Variables.*
- *Type conversion.*
- *Operators.*

## INTRODUCTION

In this chapter, you will learn about the fundamental elements of Java such as data types, variables, operators, and so on. The data type of an element specifies the kind of data stored in it and the range of values that a data element can hold. A variable is a named storage location where the data can be stored. An operator is defined as a symbol that represents an operation. In this chapter, you will also learn about the concept of type conversion. In type conversion, one type of data is converted into another type.

## DATA TYPES

While programming, you need to store a particular type of data in the computer's memory. The compiler should know the amount of memory that has to be allocated to that particular data. For this purpose, the data types are used. The main role of a data type is to direct the compiler to allocate a specific amount of memory to a particular type of data. Java is a strongly typed language, which means each type of data is predefined as a part of the language. Java provides eight primitive data types that are as follows:

- a. byte
- b. short
- c. int
- d. long
- e. float
- f. double
- g. char
- h. boolean

The above eight primitive data types are grouped into four different categories that are as follows:

- a. **Integers:** The first four primitive data types such as **byte**, **short**, **int**, and **long** are included in this category and these data types are used only for signed whole numbers. These type of numbers do not contain any decimal values.
- b. **Floating-point numbers:** The next two primitive data types, **float** and **double** are included in this category. These data types are used only for those numbers that contain real values.
- c. **Characters:** The next primitive data type **char** is included in this category. This data type is used to represent a character from the unicode character set. You will learn more about unicode character set later in this chapter.
- d. **Boolean:** The last primitive data type **boolean** is included in this category. This data type is used only to represent true or false value.



### Note

*In most of the programming languages such as C++, the amount of memory allocated to a particular data type depends upon the machine architecture. But in Java, the size of all data types is strictly defined and it does not depend upon the machine architecture.*

## Integers

The integer data type is used only for those numbers that do not contain any fractional part or any decimal point. In other words, this data type is used only for signed whole numbers, either negative or positive. In Java, four integer types are defined, **byte**, **short**, **int**, and **long**. The main difference among them is the amount of memory allocated to each of them and the maximum range of values that can be stored using each data type. Table 2-1 shows the size and range of all integer data types.

Name	Size (in bytes)	Ranges
<b>byte</b>	1	-128 to 127
<b>short</b>	2	-32,768 to 32,767
<b>int</b>	4	-2,147,483,648 to 2,147,483,647
<b>long</b>	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

*Table 2-1 Integer types, their size and ranges*

### byte

The **byte** is the smallest integer type. The size of **byte** data type is 8-bits (1 byte is equal to 8 bits) and it ranges from -128 to 127. Here, range means that the **byte** data type can store -128 as the minimum value and 127 as the maximum value. This data type is very useful while working with files or streams in Java. You can create a variable of **byte** type by using the keyword **byte** with the variable name as given below:

```
byte var_name;
```

In the above syntax, the variable **var\_name** is declared as the **byte** data type.



#### Note

*Variables will be discussed later in this chapter.*

### short

The **short** data type is used rarely in Java. The size of **short** data type is 16-bits and it ranges from -32,768 to 32,767. You can create a variable of the **short** type as given below:

```
short var_name;
```

In this syntax, the variable **var\_name** is declared as the **short** data type.

**int**

Among the integers, **int** is the most commonly used data type in Java. The size of **int** type is 32-bits and it ranges from -2,147,483,648 to 2,147,483,647. You can create a variable of the **int** type as given below:

```
int var_name;
```

In this syntax, the variable **var\_name** is declared as the **int** data type.

**long**

Among the integers, **long** is the largest storage data type. This data type is required in those cases when the range of **int** type is not large enough to hold the resultant value. The size of the **long** type is 64-bits and the range is large enough to hold the large whole numbers. You can create a variable of the **long** type as given below:

```
long var_name;
```

In this syntax, the variable **var\_name** is declared as the **long** data type.

**Floating-point Types**

The floating-point data types are used only for those numbers that contain a decimal point or that have a fractional part. These types of numbers are also known as real numbers. In Java, two floating-point types are defined, **float** and **double**. Table 2-2 shows the size and range of these two data types.

Name	Size (in bytes)	Ranges (Approx.)
<b>float</b>	4	1.4e-45 to 3.4028235e38
<b>double</b>	8	4.9e-324 to 1.7976931348623157e308

*Table 2-2 Floating-point types, their size and ranges*

**float**

The **float** type is used for single-precision values (the values, which contain upto 8 digits after the decimal point). The size of **float** data type is 32-bits and it ranges from 1.4e-45 to 3.4028235e38. You can create a variable of **float** type as given below:

```
float var_name;
```

In this syntax, the variable **var\_name** is declared as the **float** data type.

**double**

As the name implies, the **double** type is used for double precision values (the values, which contain upto 15 digits after the decimal point). This data type is mostly used in scientific

operations where the end user wants accuracy in the resultant values. The size of **double** data type is 64-bits and it ranges from 4.9e-324 to 1.7976931348623157e308. You can create a variable of **double** type, as given below:

```
double var_name;
```

In this syntax, the variable **var\_name** is declared as the **double** type.

## Characters

The **char** type is included in this category. In Java, the **char** data type is used to hold the character values.

### char

As already discussed, the **char** data type is used to hold the character values that belong to the unicode character set. But the **char** type in Java is completely different from the **char** type in other programming languages such as C, C++, and so on. In C/C++, the size of **char** type is 8-bits and it can support only a few character sets such as English, German, and so on. In Java, the size of **char** type is 16-bits and it is used to hold the values of unicode character set. Unicode character set is a collection of those characters, which exist in all human languages. The range of **char** type is from 0 (minimum) to 65,536 (maximum). You can create a variable of **char** type, as given below:

```
char var_name;
```

In this syntax, the variable **var\_name** is declared as the **char** data type.



#### Note

*A character that is assigned to a **char** variable should be enclosed in single quotes ‘ ’.*

## Boolean

The primitive data type **boolean** comes under this category.

### boolean

This data type can hold only one value, either true or false. The size of **boolean** data type is 1-bit. This data type is used to hold only logical values. You can create a variable of **boolean** type, as given below:

```
boolean var_name;
```

In this syntax, the variable **var\_name** is declared as the **boolean** data type. Therefore, the variable **var\_name** can hold either true or false value.

## VARIABLES

A variable is a named location where the data can be stored. It is a location in the computer's memory with a specific address, where a value can be stored and retrieved, when required. The value of a variable can vary while the program is being executed.

### Variable Name

While naming a variable, you must follow certain rules. These rules are as follows:

- Only alphabetic characters, both uppercase and lowercase, digits from 0 to 9, and underscore ( `_` ) can be used.
- The name can start with an alphabet or an underscore but not with a digit.
- The uppercase and lowercase characters are considered individually by the compiler.
- Keywords cannot be used as identifiers.

The following variable names are invalid in Java:

```
6_varname    //Starting with a digit
varname#     //Using a special character (#) such as %, *, #, and soon
```

The following variable names are valid in Java:

```
varname_6
var_name
```

### Declaring a Variable

A variable must be declared before it is used in a program. The syntax for declaring a variable is as follows:

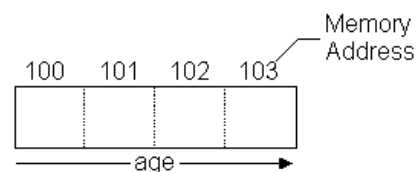
```
data_type var_name;
```

In the above syntax, the declaration has two parts. The first part **data\_type** represents a data type, which specifies the type of value to be stored in the variable and the amount of memory to be allocated. The second part **var\_name** represents the variable name.

For example, you can declare an integer type variable **age** with the help of the following statement:

```
int age;
```

When this statement executes, the compiler will allocate 4 bytes (size of **int** data type is 4 bytes) of memory to the variable **age** as shown in Figure 2-1. Now, the variable **age**, will be treated as a reference to the allocated memory location.



**Figure 2-1** Memory allocation

You can also declare multiple variables of the

same type in a single statement. These variables are separated by commas. The syntax for declaring multiple variables is as follows:

```
data_type var1, var2, var3;
```

In the above syntax, **var1**, **var2**, and **var3** are declared as the variables of the particular data type, which is represented by **data\_type**. Here, all the three variables are of the same data type.

For example:

```
float highest_temp, lowest_temp;
```

In this example, the **highest\_temp** and the **lowest\_temp** are declared as the **float** type variables.

## Initializing a Variable

Initialization means to assign an initial value to a variable. You can assign a value to a variable by using the assignment operator (**=**). The assignment operator will be discussed later in this chapter. The syntax for initializing a variable is as follows:

```
data_type var_name = value;
```

In the above syntax, the **data\_type** specifies the type of data, the **var\_name** specifies the name of the variable, and the **value** specifies the initial value, which is assigned to the variable **var\_name**.

For example:

```
char ch = 'y';
```

In this example, the character value **y** is assigned to the character variable **ch** as an initial value. Now, the character value **y** is stored at the memory location, which is referred by the variable **ch**.



### Note

*If you do not assign an initial value to a variable, an error message will be displayed on the output screen.*

## Initializing a Variable Dynamically

In the previous section, you have learned that a variable is initialized at the time of its declaration. In Java, you can also initialize a variable dynamically( at the time of program execution).

For example:

```
int sum=a+b;
```

When the above statement is executed, first the values of variables a and b are added with the help of the addition operator(+). Next, the resultant value is assigned to the integer variable **sum**.



#### Note

*All the operators will be discussed later in this chapter.*

The following example illustrates the concept of dynamic initialization of a variable:

### Example 1

Write a program to calculate the average of three numbers.

The following program will calculate the average of three numbers, assign the resultant value to another variable, and display it on the screen. The line numbers on the right are not a part of the program and are for reference only.

```
//Write a program to calculate the average of three numbers      1
class average                                                    2
{                                                                    3
    public static void main(String arg[])                        4
    {                                                            5
        int a=10, b=14, c=33;                                    6
        float avg;                                              7
        avg= (a+b+c)/3; //Dynamic initialization of variable avg 8
        System.out.println("The average of three numbers is: " +avg); 9
    }                                                            10
}                                                                    11
```

#### Explanation

Line 2

**class average**

In this line, the **class** keyword is used to define a new class and the identifier **average** is the name of the class.

Line 3

```
{
```

This line indicates the start of the definition of the class **average**.



Line 4

```
public static void main(String arg[])
```

This line contains the **main()** method, which is treated as the starting point of every Java program. So, the execution of the program will start from this line.

Line 5

```
{
```

This line indicates the start of the definition of the **main()** method.

Line 6

```
int a=10, b=14, c=33;
```

In this line, a, b, and c are declared as integer type variables and the initial values **10**, **14**, and **33** are assigned to them, respectively with the help of the assignment operator(=).

Line 7

```
float avg;
```

In this line, avg is declared as a **float** type variable.

Line 8

```
avg=(a+b+c)/3;
```

This line represents the dynamic initialization of the variable avg. In this line, first the values **10**, **14**, and **33** of the variables **a**, **b**, and **c** will be added. After that, the resultant value **57** will be divided by **3**. Next, the resultant value **19.0** will be assigned to the variable avg at the execution time.

Line 9

```
System.out.println("The average of three numbers is:" +avg);
```

This line will display the following message on the screen:

The average of three numbers is: 19.0

**Note**

*In line 9, the + sign is used to concatenate the value of the variable **avg** to the given string.*

Line 10

```
}
```

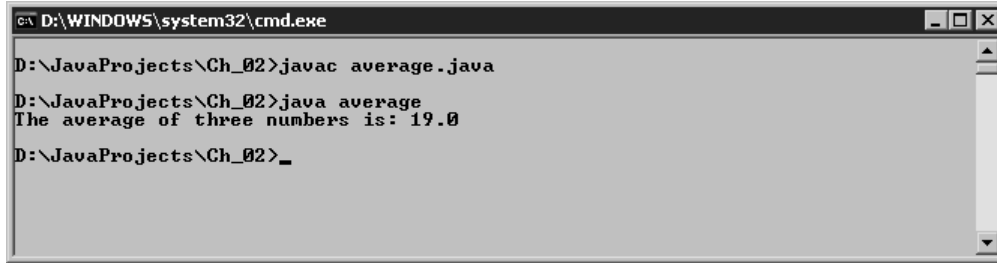
This line indicates the end of the definition of the **main()** method.

Line 11

```
}
```

This line indicates the end of the definition of the class average.

The output will be displayed on the screen as follows:



```
C:\D:\WINDOWS\system32\cmd.exe
D:\JavaProjects\Ch_02>javac average.java
D:\JavaProjects\Ch_02>java average
The average of three numbers is: 19.0
D:\JavaProjects\Ch_02>_
```

## Types of Variables

There are three types of variables in Java as given below:

- Local Variables
- Instance Variables
- Class Variables

### Local Variables

The variables that are declared inside a block of code or within the body of a method are known as local variables. These variables are accessible only within that particular block.

For example:

```
int mul( )
{
    int a=10, b=10, c;
    c=a*b;
    -----;
    -----;
}
```

In the above example, **mul()** is a method and the variables **a**, **b**, and **c** are declared inside it. These variables are local to this method and can be accessed or manipulated within this method only. You will learn about the methods in the later chapters.

### Instance Variables

The variables that are declared inside a class but outside the method are known as instance variables. It is related to a single instance of a class. These variables can be used by different methods of the same class. They are also known as member variables.

For example:

```
class Demo
{
    public static void main(String arg[])
```

```
{  
    int a, b;  
    -----;  
    -----;  
}  
}
```

In the above example, the variables **a** and **b** are declared inside the class definition but outside the methods. Therefore, they are treated as instance variables and they can be used by different methods of this class.

### Class Variables

Class variables are the same as the instance variables except that these variables are declared with the keyword **static**. You will learn about the **static** data in a later chapter.

For example:

```
class Demo  
{  
    public static void main(String arg[])  
    {  
        static int a, b;  
        -----;  
        -----;  
    }  
}
```

In this example, the variables **a** and **b** are declared with the keyword **static** and treated as the class variables.

### Scope and Lifetime of Variables

The scope of a variable refers to that part of the program within which it can be accessed and manipulated. The scope also specifies when to allocate or deallocate memory to a variable. The lifetime specifies the life-span of a variable in the computer's memory. The three types of variables discussed above have different scopes and lifetime. The scope of a local variable is only limited to that block or method within which it is declared, and the lifetime of a local variable is only till the time when that particular block or method is being executed. Once that particular block or method is terminated, the variable gets deleted from the computer's memory.

The following example illustrates the concept of scope and lifetime of variables:

#### Example 2

Write a program to add two numbers using the concept of scope and lifetime of variables.

The following program will add two numbers and display the resultant value on the screen:

```
//Write a program to add two numbers          1
class Scope_demo                               2
{                                               3
    public static void main(String arg[])      4
    {                                           5
        int c=100;                             6
        if(c==100)                             7
        {                                       8
            int a=20;                           9
            c= a+c;                            10
        }                                       11
        System.out.println("Value of c is: " +c); 12
    }                                           13
}                                              14
```

### Explanation

Line 6

**int c=100;**

In this line, **c** is declared as an integer type variable and 100 is assigned as an initial value to it. Here, variable **c** is declared inside the class definition and it can be used by any method or block of this class. The scope of the variable **c** is limited only to the class **Scope\_demo**.

Line 7

**if(c==100)**

In this line, the **if** statement is used to check whether the value of variable **c** is equal to 100 with the help of the equality operator (**==**). In this case, the value of variable **c** is equal to 100 and therefore, the control will be transferred to the next line. You will learn more about the **if** statement in the next chapter.

Line 8

{

This line indicates the start of the **if** block.

Line 9

**int a=20;**

In this line, **a** is declared as an integer type variable and 20 is assigned as an initial value to it. The variable **a** is declared inside the **if** block and treated as a local variable to that particular block. The scope of the variable **a** is limited only to the **if** block and it cannot be used outside this block.

Line 10

**c= a+c;**

In this line, the value 20 of the variable **a** is added to the value (100) of the variable **c**. After the addition, the resultant value (120) is assigned to the variable **c**.

Line 11

```
}
```

This line indicates the end of the **if** block.

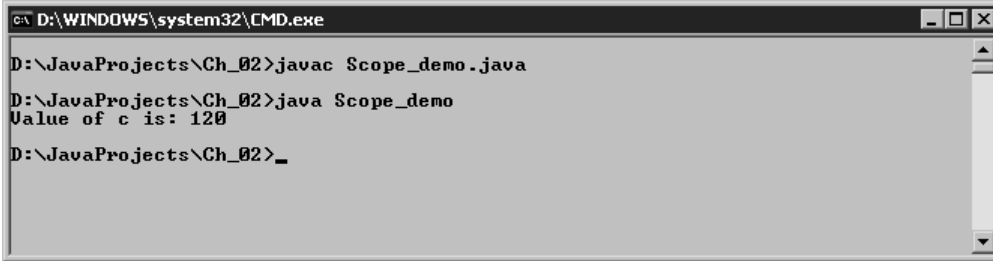
Line 12

```
System.out.println("Value of c is: " + c);
```

This line will display the following message on the screen:

Value of c is: 120

The output of the program will be displayed on the screen as follows:



```
c:\D:\WINDOWS\system32\CMD.exe

D:\JavaProjects\Ch_02>javac Scope_demo.java
D:\JavaProjects\Ch_02>java Scope_demo
Value of c is: 120
D:\JavaProjects\Ch_02>_
```

## TYPE CONVERSION

Type conversion means converting one data type into another. For example, a data element of **byte** type can be converted into the **int** type with the help of type conversion. Java supports two following types of conversion:

- a. Automatic or Compatible
- b. Explicit or Incompatible

### Automatic or Compatible Type Conversion

The automatic conversion takes place when the destination data type is larger than the source data type and both the data types are compatible. For example, a data element of **short** type is converted into the **int** type. In such cases, Java performs automatic conversion because the **int** data type is larger than the **short** data type and both the data types are compatible. In automatic conversion, no information is lost during the conversion. Table 2-3 shows some data types that can be automatically converted into another data type.

Source Type	Destination Types
byte	short, int, long, float, double
short	int, long, float, double
int	long, float, double
long	float, double
float	double

**Table 2-3** Representation source and destination data types

The following example illustrates the concept of automatic type conversion:

### Example 3

Write a program to convert a data element of **byte** type into the **int** type.

The following program will convert a data element of **byte** type into the integer type and display the result on the screen.

```
//Write a program to convert a data element of byte type into the integer type 1
class Type_demo 2
{ 3
    public static void main(String arg[]) 4
    { 5
        byte src=127; 6
        int dest; 7
        dest= src; 8
        System.out.println("dest = " +dest); 9
    } 10
} 11
```

### Explanation

Line 6

**byte src=127;**

In this line, **src** is declared as a **byte** type variable and 127 is assigned as an initial value to it.

Line 7

**int dest;**

In this line, **dest** is declared as an integer type variable.

Line 8

**dest= src;**

In this line, the automatic conversion takes place and the value 127 of the variable **src** is assigned to the integer type variable **dest**.

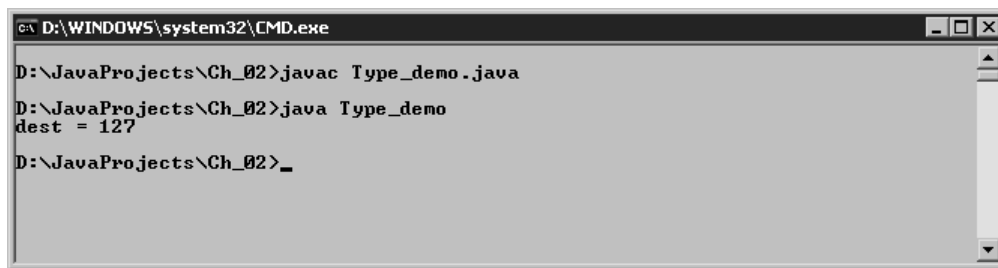
Line 9

**System.out.println("dest = " +dest);**

This line will display the following message on the screen:

dest = 127

The output of the program will be displayed on the screen as follows:



```
C:\WINDOWS\system32\CMD.exe
D:\JavaProjects\Ch_02>javac Type_demo.java
D:\JavaProjects\Ch_02>java Type_demo
dest = 127
D:\JavaProjects\Ch_02>_
```

## Explicit or Incompatible Type Conversion

In the previous section, you learned about the automatic type conversion in which the destination type was larger than the source type. But sometimes, you may need to convert a larger element type into a smaller one. For example, you may need to convert an **int** type into **byte** type. In such cases, explicit conversion is used. In explicit type of conversion, some information is always lost. Therefore, this type of conversion is also known as narrowing conversion. The syntax for explicit conversion is as follows:

(destination\_data\_type) value

In the above syntax, the **destination\_data\_type** specifies the data type in which you want to convert the value, which is specified by **value**.

For example, you can convert a value of **int** type into the **byte** type, as given below:

```
byte b;
int i =300;
b = (byte) i;
-----;
-----;
```

In the above example, **byte** in the parentheses directs the compiler to convert the value of the integer type **i** into the **byte** type. Now, the resultant value is assigned to the **byte** variable **b**.

The following example illustrates the concept of explicit type conversion:

### Example 4

Write a program to convert an **int** type into a **byte** type using the concept of explicit type conversion.

The following program will convert an **int** type into a **byte** type and display the resultant value on the screen:

```
//Write a program to convert an int type into a byte type      1
class Explicit_demo                                           2
{                                                             3
    public static void main(String arg[])                    4
    {                                                         5
        byte b;                                             6
        int val = 300;                                       7
        b = (byte) val;                                       8
        System.out.println("After conversion, value of b is: " +b); 9
    }                                                         10
}                                                             11
```

### Explanation

Line 6

**byte b;**

In this line, **b** is declared as a **byte** type variable.

Line 7

**int val = 300;**

In this line, **val** is declared as an integer type variable and **300** is assigned as an initial value to it.

Line 8

**b = (byte) val;**

In this line, the value of variable **val** is converted into **byte** because **byte** is the destination type and the resultant value will be assigned to the variable **b**.

Line 9

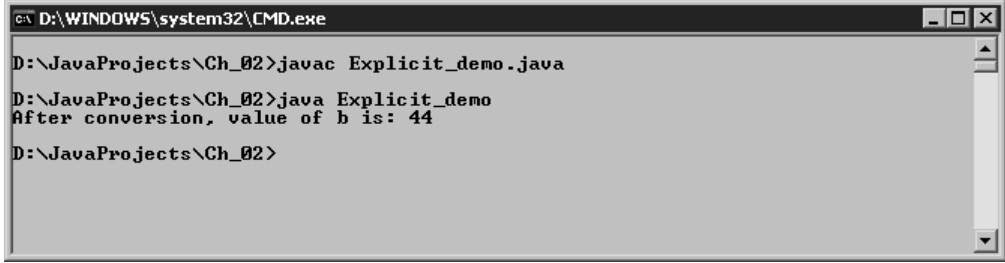
**System.out.println("After conversion, value of b is: " +b);**

This line will display the following message on the screen:

After conversion, value of b is: 44



The output of the program will be displayed on the screen as follows:



```
C:\D:\WINDOWS\system32\CMD.exe
D:\JavaProjects\Ch_02>javac Explicit_demo.java
D:\JavaProjects\Ch_02>java Explicit_demo
After conversion, value of b is: 44
D:\JavaProjects\Ch_02>
```

## OPERATORS

Operators are defined as the symbols that are used when an operation is performed on the variables or constants. Java provides with a rich variety of operators and these operators are divided into four different categories, which are as follows:

- a. Arithmetic Operators
- b. Bitwise Operators
- c. Relational Operators
- d. Logical Operators

Besides the above operators, some other important operators are also discussed in this section.

### Arithmetic Operators

Operators that are used in mathematical expressions are known as arithmetic operators. Table 2-4 lists all arithmetic operators that are used in Java.

Operator	Operation	Syntax
+	Addition	a+b
-	Subtraction(Unary Minus)	a-b, -a
*	Multiplication	a*b
/	Division	a/b
%	Modulus	a%b
++	Increment	++a, a++
+=	Addition assignment	a+=b
-=	Subtraction assignment	a-=b
*=	Multiplication assignment	a*=b
/=	Division assignment	a/=b
%=	Modulus assignment	a%=b
--	Decrement	--a, a--

*Table 2-4 Arithmetic operators and their syntaxes*

In the above table, the first four operators + (addition), - (subtraction), \* (multiplication), and / (division) are treated as the basic arithmetic operators and they work in the same way as they do in algebra. The minus operator is used for two purposes: subtraction and negating its operand.

The following example illustrates the use of basic arithmetic operators:

### Example 5

Write a program to perform various arithmetic operations on two numbers using basic arithmetic operators.

The following program will perform addition, subtraction, multiplication, and division on two numbers and display the resultant values on the screen.

```
//Write a program to perform various arithmetic operations
//on two numbers using the basic arithmetic operators:
class Basic_operators
```

1  
2

```
{
    public static void main(String arg[])
    {
        int init=10;
        int mul= init*2;
        int div= mul/init;
        int sum= init+mul;
        System.out.println("init = " +init);
        System.out.println("mul = " +mul);
        System.out.println("div = " +div);
        System.out.println("sum = " +sum);
    }
}
```

### Explanation

Line 6

**int init=10;**

In this line, **init** is declared as an integer type variable and 10 is assigned as an initial value to it.

Line 7

**int mul= init\*2;**

In this line, the value (10) of the variable **init** is multiplied by 2 and the resultant value (20) is assigned to the integer variable **mul**.

Line 8

**int div= mul/init;**

In this line, the value (20) of the variable **mul** is divided by the value (10) of the variable **init** and the resultant value (2), which represents the quotient, is assigned to the integer variable **div**.

Line 9

**int sum= init+mul;**

In this line, the value (10) of the variable **init** is added to the value (20) of the variable **mul** and the resultant value (30) is assigned to the integer variable **sum**.

Line 10

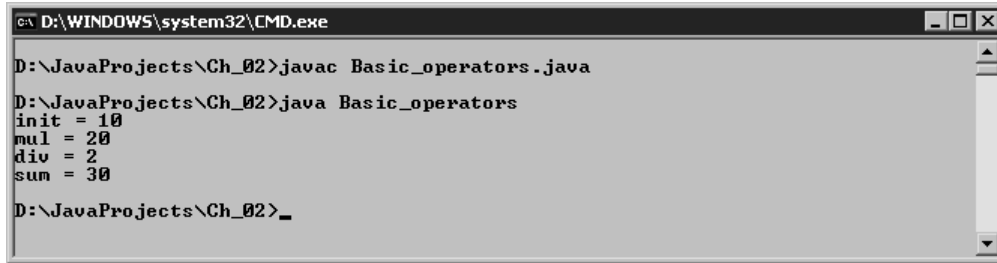
**System.out.println("init = " +init);**

This line will display the following message on the screen:

init = 10

The working of lines 11 to 13 is similar to that of line 10.

The output of the program will be displayed on the screen as follows:



```

D:\WINDOWS\system32\CMD.exe

D:\JavaProjects\Ch_02>javac Basic_operators.java

D:\JavaProjects\Ch_02>java Basic_operators
init = 10
mul = 20
div = 2
sum = 30

D:\JavaProjects\Ch_02>_

```

### The Modulus Operator (%)

A modulus operator returns the remainder after the division of two numbers. The percentage symbol (%) is used for this operator. You can use the modulus operator with both the integer and the floating-point data types. The syntax for using the modulus operator is as follows:

`var1 % var2`

In the above syntax, the modulus operator returns the remainder after the division of the values of variables **var1** and **var2**.

The following example illustrates the use of modulus operator:

#### Example 6

Write a program to find out remainders after the division of two **int** and two **double** type numbers, using the modulus operator (%).

The following program will find out remainders after the division operation. It will also display the resultant values on the screen:

```

//Write a program to find out the remainders
class Modulus_demo
{
    public static void main(String arg[])
    {
        int a = 20, b= 3;
        double f1 = 1.25, f2 = 0.5;
        System.out.println("a mod b = " + a%b);
        System.out.println("f1 mod f2 = " + f1%f2);
    }
}

```

## Explanation

Line 6

**int a = 20, b = 3;**

In this line, **a** and **b** are declared as the integer type variables, and 20 and 3 are assigned as their initial values, respectively.

Line 7

**double f1 = 1.25, f2 = 0.5;**

In this line, **f1** and **f2** are declared as the **double** type variables, and 1.25 and 0.5 are assigned as their initial values, respectively.

Line 8

**System.out.println("a mod b = " + a%b);**

In this line, **a%b** returns the remainder value (2), which is concatenated with the given string. Now, this line will display the following message on the screen:

a mod b = 2

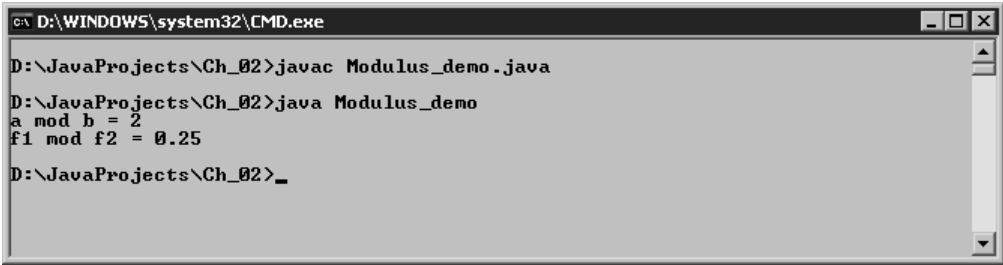
Line 9

**System.out.println("f1 mod f2 = " + f1%f2);**

In this line, **f1%f2** returns the remainder value (0.25), which is concatenated with the given string. Now, this line will display the following message on the screen:

f1 mod f2 = 0.25

The output of the program will be displayed on the screen as follows:



```
D:\WINDOWS\system32\CMD.exe
D:\JavaProjects\Ch_02>javac Modulus_demo.java
D:\JavaProjects\Ch_02>java Modulus_demo
a mod b = 2
f1 mod f2 = 0.25
D:\JavaProjects\Ch_02>_
```

## Compound Assignment Operators

Compound assignment operators are a combination of two operators: first that specifies the operation to be performed and the second is the assignment operator. The syntax for using the compound assignment operators is given next.

**var1 += var2**      //Similar to **var1 = var1 + var2**

In the above syntax, first the value of the variable **var1** is added to the value of the variable **var2**. Next, the resultant value is assigned back to **var1**.

For example, to add the value 4 to the value of the variable **a**, and again assign the resultant value to **a**, use the following statement:

```
a += 4
```

You can also perform the same operation in the following way:

```
a = a + 4
```

The following example illustrates the use of compound assignment operators:

### Example 7

Write a program to multiply and divide two numbers using the compound assignment operators.

The following program will apply the compound assignment operators on the given values and also display the resultant values on the screen:

```
//Write a program to multiply and divide two numbers
class Compound_demo
{
    public static void main(String arg[])
    {
        int a = 10;
        double b = 2.5;
        a *= 2;
        b /= 0.5;
        System.out.println("Value of a is: " + a);
        System.out.println("Value of b is: " + b);
    }
}
```

### Explanation

Line 6

**int a = 10;**

In this line, **a** is declared as an integer type variable and 10 is assigned as an initial value to it.

Line 7

**double b = 2.5;**

In this line, **b** is declared as a **double** type variable and 2.5 is assigned as an initial value to it.

Line 8

**a \*= 2;**

In this line, first the value (10) of the variable **a** is multiplied by 2. Next, the resultant value is assigned back to the variable **a**.

Line 9

**b /= 0.5;**

In this line, first the value (2.5) of the variable **b** is divided by 0.5. Next, the resultant value is assigned back to the variable **b**.

Line 10

**System.out.println("Value of a is: " +a);**

This line will display the following message on the screen:

Value of a is: **20**

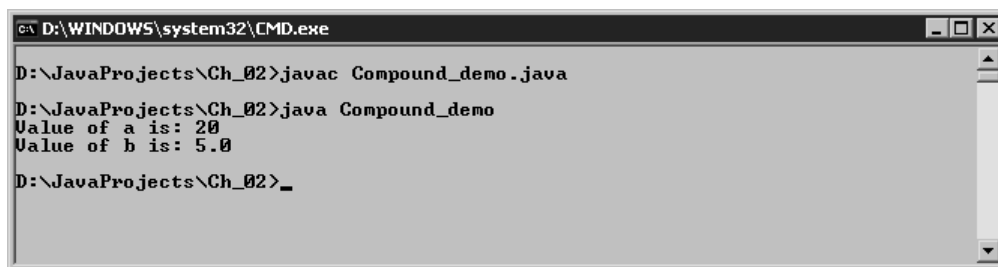
Line 11

**System.out.println("Value of b is: " +b);**

This line will display the following message on the screen:

Value of b is: **5.0**

The output of the program will be displayed on the screen as follows:



```
C:\D:\WINDOWS\system32\CMD.exe
D:\JavaProjects\Ch_02>javac Compound_demo.java
D:\JavaProjects\Ch_02>java Compound_demo
Value of a is: 20
Value of b is: 5.0
D:\JavaProjects\Ch_02>_
```

### Increment (++) and Decrement (--) Operators

The ++ operator is used to increase the value of its operand by one and the -- operator is used to decrease the value of its operand by one.

For example:

```
val++;    //Equivalent to val = val+1; or val + = 1
val--;    //Equivalent to val = val-1; or val - = 1
```

In the above example, the first statement **val++** will increase the value of the variable **val** by one and the second statement **val--** will decrease the value of the variable **val** by one.

You can use these operators in two notations, which are as follows:

- Postfix notation
- Prefix notation

**The Postfix notation**

In postfix notation, the increment or decrement operator is used after the operand. The syntax for using the postfix operator is as follows:

```
var1 ++;
```

In this syntax, the increment operator (++) is used after the operand **var1**. It will increase the value of the variable **var1** by one.

If the postfix notation is used in an expression, then first the value of an operand is assigned to the variable at the left and then the value of the operand will be incremented or decremented by one.

For example:

```
y = x--;
```

In the above example, first the value of the variable **x** is assigned to the variable **y** and then it is decreased by one.

The following two statements produce the same result as produced by the **y = x--** statement given in the previous example.

```
y = x;  
x = x-1;
```

**The Prefix notation**

In prefix notation, the increment or decrement operator is used before the operand. The syntax for using the prefix operator is as follows:

```
++var1;
```

In the above syntax, the increment operator (++) is used before the operand **var1**. It will increase the value of the variable **var1** by one.

If the prefix notation is used in an expression, then first the value of an operand is incremented or decremented by one and then it will assign to the variable at the left.

For example:

```
y = --x;
```

In the above example, first the value of the variable **x** is decreased by one and then it is assigned to the variable **y**.

The following two statements produce the same result as was produced by the **y = --x** statement given in the previous example.



```
x = x-1;  
y = x;
```

The following example illustrates the use of increment and decrement operators:

### Example 8

Write a program to apply the increment and decrement operators on the values of the given variables.

The following example will first increase and then decrease the given values and finally display the resultant values on the screen:

```
//Write a program to apply the increment and decrement operators  
//on the given values  
class Inc_Dec_demo  
{  
    public static void main(String arg[])  
    {  
        int x=10;  
        int y, z, a;  
        y= x++;  
        z= ++y;  
        a= y--;  
        --a;  
        System.out.println("x = " +x);  
        System.out.println("y = " +y);  
        System.out.println("z = " +z);  
        System.out.println("a = " +a);  
    }  
}
```

### Explanation

Line 6

**int x=10;**

In this line, **x** is declared as an integer type variable and 10 is assigned as an initial value to it.

Line 7

**int y, z, a;**

In this line, **y**, **z**, and **a** are declared as integer type variables.

Line 8

**y= x++;**

In this line, first the value (10) of the variable **x** is assigned to the variable **y**. Next, the value of the variable **x** is incremented by 1.

Line 9

**z= ++y;**

In this line, first the value (10) of the variable **y** is incremented by 1. Next, it is assigned to the variable **z**.

Line 10

**a= y--;**

In this line, first the value (11) of the variable **y** is assigned to the variable **a**. Next, it is decremented by 1.

Line 11

**--a;**

In this line, the value (11) of the variable **a** is decremented by 1.

Line 12

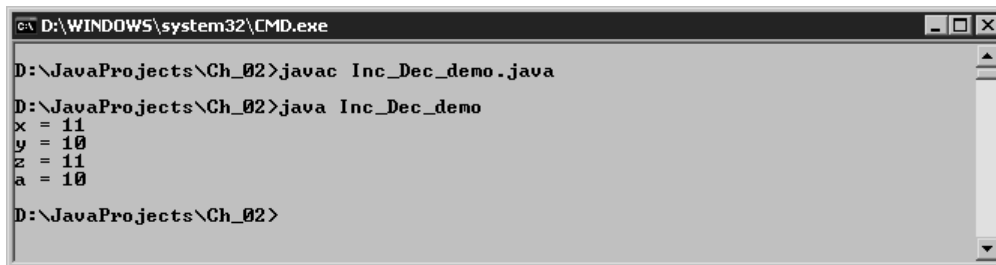
**System.out.println("x = " +x);**

This line will display the following statement on the screen:

x = 11

The working of lines 13 to 15 is similar as that of line 12.

The output of the program will be displayed on the screen as follows:



```
C:\D:\WINDOWS\system32\CMD.exe
D:\JavaProjects\Ch_02>javac Inc_Dec_demo.java
D:\JavaProjects\Ch_02>java Inc_Dec_demo
x = 11
y = 10
z = 11
a = 10
D:\JavaProjects\Ch_02>
```

## The Bitwise Operators

The data is stored in the computer's memory in the form of 0's and 1's, and these are known as bits. For example, a **byte** value 3 is stored in the computer's memory as 00000011. To operate or manipulate these bits individually, Java provides some operators that are known as bitwise operators. The bitwise operators are used to operate on the single bits of an operand. These operators are mostly applied on the integer types such as **byte**, **short**, **int**, and **long**. They can also be applied on the **char** type. Table 2-5 shows a list of bitwise operators.

Operator	Operation
~	Bitwise unary NOT
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right fill zero
<<	Shift left
&=	Bitwise AND assignment
=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right fill zero assignment
<<=	Shift left assignment

**Table 2-5** List of Bitwise operators

These operators are the least commonly used operators. Some of the bitwise operators are categorized under bitwise logical operators and these are discussed below:

### The Bitwise Unary NOT (~) Operator

The bitwise unary NOT (~) operator comes under the category of bitwise logical operators. The ~ operator inverts all bits of its operand; for example, 0 becomes 1 and 1 becomes 0. This operator is also known as the bitwise complement operator. The syntax for using the unary NOT (~) operator is as follows:

~ value or expression;

For example:

```
int a = 3;
int b = ~a;
```

In the above example, 3 is assigned to the integer variable **a** as an initial value, which is stored in the computer's memory as 00000011. In the next statement, **~** operator is used with the integer variable **a**. This operator inverts all the bits 00000011 of the value 3 into 11111100. Then, the resultant value is assigned to the integer variable **b**.

### The Bitwise AND (&) Operator

The bitwise AND (&) operator also comes under the category of bitwise logical operators. If both the operands consist of the value 1, then the & operator will produce bit 1 as the result. But, if one or both the operands consist of the value 0, then the & operator will produce 0 as the result. The syntax for using the & operator is as follows:

```
operand1 & operand2;
```

For example, you can use the AND (&) operator with two operands: 23 and 15, as given next.

```
00010111 //Bits representing the value 23
& 00001111 //Bits representing the value 15
-----
00000111 //Bits representing the value 7
```

### The Bitwise OR (|) Operator

The bitwise OR (|) operator also comes under the category of bitwise logical operators. If one or both the operands consist of the value 1, then the | operator will produce bit 1 as the result. But, if both the operands contain 0, then the | operator will produce 0 as the result. The syntax for using the | operator is as follows:

```
operand1 | operand2;
```

For example, you can use the OR (|) operator with two operands: 23 and 15, as follows:

```
00010111 //Bits representing the value 23
| 00001111 //Bits representing the value 15
-----
00011111 //Bits representing the value 31
```

### The Bitwise exclusive OR (^) Operator

The bitwise exclusive OR (^) or XOR operator also comes under the category of bitwise logical operators. The ^ operator produces bit 1 as the result, if only one of the operands consists of the value 1. Otherwise, it produces bit 0 as the result. The syntax for using the ^ operator is as follows:

```
operand1 ^ operand2;
```

For example, you can use the XOR (^) operator with two operands: 23 and 15, as follows:

```
00010111 //Bits representing the value 23
^ 00001111 //Bits representing the value 15
-----
00011000 //Bits representing the value 24
```

Table 2-6 represents all (~, &, |, and ^) bitwise logical operators.

X	Y	X&Y	X Y	X^Y	~X	~Y
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

*Table 2-6 Bitwise logical operators*

Other than the bitwise logical operators, the following operators are also available:

### The Right Shift (>>) Operators

The right shift (>>) operator is used to move all the bits of an operand to the right direction. The >> operator operates on the bits for a specified number of times. The syntax for using the right shift operator is as follows:

```
value or expression >> num
```

In the above syntax, the **num** specifies the total number of times you want to perform the right shift operation on all the bits of a value, which is specified by **value** or **expression**.

For example:

```
int a = 17;
int b = a>>2;
```

The above example will operate in the following way:

```
00010001 //Bits representing the value 17
```

When the `>>` operator operates on the above bits for the first time, the right most bit, bit 1, will be lost and all other bits will shift to the right. The bit pattern, which is produced after the first step is as follows:

```
00001000 //Bits representing the value 8
```

In the second step, the same process is repeated as in the first step and the bit pattern, which is produced after the second step is as follows:

```
00000100 //Bits representing the value 4
```

**Note**

*Whenever the right shift operator (`>>`) is applied to a value, it divides the given value by two.*

### The Left Shift (`<<`) Operator

The left shift (`<<`) operator is used to move all the bits of an operand to the left direction. The `<<` operator operates on the bits for a specified number of times. The syntax for using the left shift operator is as follows:

```
value or expression << num
```

In the above syntax, the **num** specifies the total number of times you want to perform the left shift operation on all bits of a **value** or **expression**.

For example:

```
int a = 17;  
int b = a<<2;
```

The above example will operate in the following way:

```
00010001 //Bits representing the value 17
```

When the `<<` operator operates on the above bits for the first time, the left most bit, bit 0 will be lost and all other bits will shift to the left. The bit pattern that is produced after the first step is as follows:

```
00100010 //Bits representing the value 34
```

In the second step, the same process is repeated as in the first step and the bit pattern, which is produced after the second step is as follows:

```
01000100 //Bits representing the value 68
```

**Note**

Whenever the left shift operator ( $\ll$ ) is applied to a value, it multiplies the given value by two.

### The Bitwise Compound Assignment Operators

The working of the bitwise compound assignment operators is similar to the compound assignment operators that were already discussed earlier in this chapter. The syntax for using the bitwise compound assignment operator is as follows:

```
var1 >>= Value;
```

In the above syntax, first the right shift operator shifts the value of the **var1** by a number of bits, which is specified by **Value**. Then, the resultant value is assigned back to the variable **var1**.

For example:

```
int a = 15;
int b = 25;
a |= b;
```

In the above example, first the OR operation is performed between the bits of value 15 (00001111) and the bits of value 25 (00011001), as given next.

```

00001111
| 00011001
-----
00011111      //Bits representing the value 31
```

Next, the resultant value 31 is assigned back to the variable **a**.

**Note**

You can use all bitwise compound assignment operators in the same way as the  $\mid=$  operator has been used in the above example.

### The Relational Operators

The relational operators are used to determine the relationship between two expressions. These operators are basically used to compare two values and the outcome of these operators is a **boolean** value, either **true** or **false**. Table 2-7 shows the list of relational operators.

Operator	Operation
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than equal to
<=	Less than equal to

**Table 2-7** List of relational operators

The syntax for using these operators is as follows:

```
var1 == var2
```

In the above syntax, the **==** (equal to) operator is used to check the equality between the two variables, **var1** and **var2**. If the value of the **var1** is equal to the value of the variable **var2**, the outcome of this operation is **true**. Otherwise, it is **false**.



**Note**

*The equality operator **==** used in the above syntax is not the same as the assignment operator **=**. The **==** operator is used to check equality between two operands and the **=** operator is used to assign the value of an operand to another operand.*

You will learn more about the working of the relational operators in the next chapter as these operators are mostly used in the control flow statements.

## The Logical Operators

In the previous section, you learned about the relational operators, which are used to compare two expressions or which operate on a single condition. But sometimes, you may need to compare two or more conditions in a single statement. For this purpose, Java provides a kind of operators, known as the logical operators. The logical operators are used to compare two or more relational expressions (statements that contain a relational operator) at a time and the outcome of these operators is a **boolean** value, either **true** or **false**. Table 2-8 shows a list of the logical operators.



Operator	Operation
&	Boolean logical AND
	Boolean logical OR
^	Boolean logical XOR
&&	Short-circuit AND
	Short-circuit OR
!	Unary NOT

*Table 2-8 List of logical operators*

Among these operators, the working of the **&** (logical AND), **|** (logical OR), and **^** (logical XOR) operators is the same as the bitwise operators, except that they operate on the **boolean** values. However, the **!** (unary NOT) operator is used for inverting a **boolean** value from **false** to **true** and vice-versa.

### Logical Short-Circuit AND (&&) and OR (||) Operators

The short-circuit **&&** and **||** operators are mostly used in such control flow statements, in which the final outcome is based on the outcome of two or more than two conditions. The **&&** operator returns **true**, if the outcome of all operands is **true**. Otherwise, **false**. However, the **||** operator returns **true**, if the outcome of any one of the operands is **true**. Otherwise, it is **false**. Table 2-9 shows the working of short-circuit **&&** and **||** operators.

X	Y	X&&Y	X  Y
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

*Table 2-9 Short-circuit AND and OR operators*

In the above table, you can observe that the **&&** operator returns **true**, only when both the operands are **true**. Otherwise, it returns **false**. Whereas, the **||** operator returns **true**, even if any one or both the operands are **true**. These operators are also known as the short-circuit operators because when these operators are used, only the left-hand operand is evaluated. Based on the result of that single operand, the final outcome will be produced. You will learn more about the working of these operators in the next chapter.

## OTHER IMPORTANT OPERATORS

Some of the other important operators are discussed below:

### The Assignment (=) Operator

You have already learned about the = operator in the previous examples of this chapter. This operator is used to assign a value to a variable. The single equal (=) symbol is used for the assignment operator. The syntax for using the assignment operator is as follows:

```
variable_name = value;
```

In this syntax, the **value** on the right of the assignment operator (=) is assigned to the variable **variable\_name** on the left. The left value should always be a variable. The right value can be a variable, a constant, or the result of an operation.

You can also use the assignment operator (=) for multiple assignments. Its syntax is as follows:

```
var1 = var2 = var3 = value;
```

In the above syntax, the same value represented by **value** is assigned to all the three variables **var1**, **var2**, and **var3**.

### The ? : Operator

The ? : operator is also known as the ternary operator. The working of ? : operator is the same as that of the **if-else** control statement. The syntax for using the ? : operator is as follows:

```
conditional_expression ? statement 1 : statement 2
```

In the above syntax, if the condition specified by **conditional\_expression** results in **true**, the **statement 1** will be executed. Otherwise, the **statement 2** will be executed.

For example:

```
int c = a!=0 ? a : b;
```

In the above example, first the conditional expression **a!=0** (value of the variable **a** is not equal to 0) is evaluated. If it results in **true**, the value of the variable **a** will be assigned to the integer variable **c**. Otherwise, the value of the variable **b** will be assigned to the integer variable **c**.

The following example illustrates the use of ternary ( ? : ) operator:

### Example 9

Write a program to find greater of the two given numbers using the ternary ( ? : ) operator.

The following program will find the greater of the two given numbers, assign the resultant value to another variable, and also display the resultant value on the screen.

```
//Write a program to find the greater number
class Ternary_demo
{
    public static void main(String arg[])
    {
        int a=20, b=11, c;
        c= a>b ? a : b;
        System.out.println("The greater value is: " +c);
    }
}
```

### Explanation

Line 7

**c= a>b ? a : b;**

In this line, the ? : operator is used. First, the conditional expression **a>b** is evaluated. The expression results in **true** because the value 20 of the variable **a** is greater than the value 11 of the variable **b**. Now, the resultant value 20 of the variable **a** will be assigned to the integer variable **c**.

Line 8

**System.out.println("The greater value is: " +c);**

This line will display the following message on the screen:

The greater value is: 20

The output of the program will be displayed on the screen as follows:



```
C:\D:\WINDOWS\system32\CMD.exe
D:\JavaProjects\Ch_02>javac Ternary_demo.java
D:\JavaProjects\Ch_02>java Ternary_demo
The greater value is: 20
D:\JavaProjects\Ch_02>_
```

## Operator Precedence

The operator precedence determines the order of execution of operators by the compiler. An operator with a high precedence is executed before an operator with a low precedence.

For example:

```
x=a+b*c
```

The multiplication operator (\*) has a higher precedence than the addition (+) and the assignment operators (=). So, in the above example, first the value of the variable **b** is multiplied by the value of the variable **c**, and then the resultant value is added to the variable **a** (because the addition operator has a higher precedence than the assignment operator). Next, the resultant value of the expression **a+b\*c** is assigned to the variable **x**.

A list of Java operators arranged from the highest to the lowest precedence is given in the Table 2-10.

Precedence	Operators
1	( )      []      .
2	++      --      ~      !
3	*      /      %
4	+      -
5	>>      >>>      <<
6	>      >=      <      <=
7	==      !=
8	&
9	^
10	
11	&&
12	
13	? :
14	=      op=

**Table 2-10** List of operators and their precedence

**Note**

*In the above table, the highest precedence is represented by 1 and the lowest precedence is represented by 14. And, the operators given in the same line have the same precedence.*

**Self-Evaluation Test**

Answer the following questions and then compare them to those given at the end of this chapter:

1. A \_\_\_\_\_ is a named storage location, where the data can be stored.
2. Variables declared inside a class but outside the method are known as the \_\_\_\_\_ variables.
3. The \_\_\_\_\_ conversion is used to convert a larger data type into a smaller one.
4. A \_\_\_\_\_ operator returns the remainder after the division of two numbers.
5. The \_\_\_\_\_ operator is used to increase the value of its operand by one.
6. In Java, the size of all data types is clearly defined. (T/F)
7. A variable can start with a digit. (T/F)
8. In Java, the + sign is used for concatenation. (T/F)
9. In Java, a class variable is declared with keyword **static**. (T/F)
10. The % operator returns the quotient after the division of two numbers. (T/F)

**Review Questions**

Answer the following questions:

1. Differentiate between the local and instance variables.
2. Explain incompatible or explicit type conversion with the help of a suitable example.
3. Explain the working of the % operator with the help of a suitable example.
4. Explain the working of the prefix increment operator with the help of a suitable example.
5. Explain ? : operator with a suitable example.

**Find errors in the following program statements:**

```
6. class Demo
{
    public static main void(String args[])
    {
        System.out.println("Hello Java");
    }
}
```

```
7. class Variable_demo
{
    public static void main(String args[])
    {
        int a = 10 , b=19;
        c=a+b;
        System.out.println(c);
    }
}
```

```
8. class Type_convert
{
    public static void main(String args[])
    {
        byte a;
        int b = 200;
        a = b;
        -----;
        -----;
    }
}
```

```
9. Class Syntax
{
    public static void main(String args[])
    {
        System.out.println("Error");
    }
}
```

```
10. class Ternary
{
    public static void main(String args[])
    {
        int x= 10, y=10, c;
        c= x==y ? x : y;
```

```
        -----;  
        -----;  
    }  
}
```

## Exercises

### Exercise 1

Write a program to calculate the area of a circle having radius 10.5 units.

### Exercise 2

Write a program to shift the value 200 to the right by two positions using the (shift right) >> operator.

**Answers to Self-Evaluation Test**

1. variable, 2. instance, 3. explicit, 4. %, 5. ++, 6. T, 7. F, 8. T, 9. T, 10. F