

Chapter 2

Fundamental Elements of Java

Learning Objectives

After completing this chapter, you will be able to:

- *Understand the concept of identifiers*
- *Understand the concept of keywords*
- *Understand the concept of data types*
- *Understand the escape sequences*
- *Understand the concept of variables*
- *Understand the concept of type conversion*
- *Understand the concept of operators*
- *Understand the concept of command-line arguments*

INTRODUCTION

In this chapter, you will learn about the fundamental elements of Java such as identifiers, keywords, literals, data types, variables, operators, and so on. Identifiers are the names of packages, classes, interfaces, methods, or variables. Literals are notations that represent a fixed value to be stored in variables. 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.

IDENTIFIERS

All components of Java require names. Therefore, identifiers are names of packages, classes, interfaces, methods, or variables. To name an identifier, you must follow certain rules. These rules are as follows:

- Identifier must start with an alphabet (A-Z, a-z) or an underscore(_) or dollar sign (\$) but not with a digit.
- After the first character, an identifier can have any combination of characters which can be an alphabet or digit but no special character except underscore () and dollar sign (\$).
- Java is case sensitive, so the uppercase and lowercase characters are considered individually by the compiler like Cadcim and CADCIM are two different identifiers.
- Java keywords cannot be used as identifiers.

The following variable names are valid in Java:

```
idname_6  
id_name  
_idname
```

The following variable names are invalid in Java:

```
6_idname //Starting with a digit  
idname# //Using a special character (#) such as %, *, #, and so on  
id name //Using space
```

KEYWORDS

Java programming language has some reserved keywords that cannot be used as identifier names because they have special meaning for the compiler. Due to their specific functions in the language, the keywords are highlighted in a different color for easy identification in most integrated development environments of Java.

There are 50 keywords in Java which are listed as follows:

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	final	finally	float
for	goto	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

Some important points regarding Java keywords are given next:

- a. **const** and **goto** are reserved keywords but not used.
- b. All keywords are in lowercase.
- c. **true**, **false**, and **null** are literals, not keywords.

DATA TYPES

Data type describes the size and type of values that can be stored in a variable. In any program, 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. In Java, a data type is divided into three categories:

1. Primitive data types
2. Derived data types
3. User defined data types



Note

Variables will be discussed later in this chapter.

Primitive Data Types

The primitive data types are predefined by the Java programming language. These are the basic data types. They are declaration types and are used to represent single values but not multiple values. Java provides eight primitive data types that are as follows:

- byte
- short
- int
- long
- float
- double
- char
- boolean

These eight primitive data types are grouped into four different categories which are discussed next.



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 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.

Table 2-1 Integer types, their size and ranges

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

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 when working with files or streams in Java. It is used to save space in large arrays. You can create a variable of **byte** type by using the **byte** keyword with the variable name is given next:

```
byte var_name;
```

In this syntax, the variable **var_name** is declared as the **byte** data type.

short

The **short** data type is used rarely in Java. The size of **short** data type is 16-bits (2 bytes) and it ranges from -32,768 to 32,767. You can use it to save large space. You can create a variable of the **short** type as given next:

```
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 (4 bytes) and it ranges from -2,147,483,648 to 2,147,483,647. It is generally used as the default data type for integer values unless there is a concern about memory. You can create a variable of the **int** type as given next:

```
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 (8 bytes) and the range is large enough to hold the large whole numbers. You can create a variable of the **long** type as given next:

```
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.

Table 2-2 Floating-Point types, their size and ranges

Name	Size (in bytes)	Range(Approx.)
float	4	1.40e-45 to 3.40e+38
double	8	4.9e-324 to 1.8e+308

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 (4 bytes) and it ranges from 1.40e-45 to 3.40e+38. You can create a variable of **float** type as given next:

```
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 1.8e+308. You can create a variable of **double** type, as given next:

```
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 single character value that can be represented by alphabets, digits, and special symbols.

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 (1 byte) 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 (2 bytes) 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 65,535 (maximum). You can create a variable of **char** type as given next:

```
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. It can be used for storing true or false values.

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. By default, it returns false. You can create a variable of **boolean** type, as given next:

```
boolean var_name;
```

In this syntax, the variable **var_name** is declared as the **boolean** data type.

Derived Data Types

Derived data types are the data types whose variables can hold more than one value of same type. They are not allowed to store multiple values of different types. They are made by using primitive data types. Example of derived data types are arrays and string.

For example:

```
int a[]={10, 20, 30}; //valid array
int b[]={10,10.5, 'A'}; //invalid array
char carray[]={ 'c', 'a', 'd', 'c', 'i', 'm' };
```

User-defined Data types

User defined data types are the data types whose variables can store multiple values of either same type or different types. These data types are defined by programmers by making use of appropriate features of the language. Some user defined data types are classes and interfaces.



Note
You will learn about the arrays, strings, classes, and interfaces in the later chapters.

ESCAPE SEQUENCES

Escape sequence is a sequence of characters that are used to send a command to a device or a program. These characters are preceded by a backslash (\), which is called an escape character. These characters are not only used for text formatting but they also serve a special purpose. Table 2-3 shows the list of escape sequences used in Java.

Table 2-3 List of escape sequences

Escape Sequence	Description
\t	Insert a tab
\\	Insert a backslash
\'	Insert a single quote
\"	Insert a double quote
\r	Insert a carriage return
\n	Insert a new line
\b	Insert a backspace
\f	Insert a form feed

Example 1

The following program will display the use of escape sequence characters.

```
//Write a program to show the use of escape sequence characters
1  class Escape
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Linefeed      : \nLearning Java");
6          System.out.println("Single Quote  : \'Learning Java\'");
7          System.out.println("Double Quote : \"Learning Java\"");
8          System.out.println("Backslash   : \\Learning Java\\");
9          System.out.println("Horizontal Tab : Learning\tJava");
10         System.out.println("Backspace    : Learning\bJava");
11         System.out.println("Carriage Return: Learning\rJava");
12     }
13 }
```

Explanation

Line 1

class Escape

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

Line 3

public static void main(String arg[])

This line contains the **main()** method which is treated as the starting point of every Java program. The execution of the program starts from this line.

Line 5

System.out.println("Linefeed : \nLearning Java");

This line will display the following on the screen:

Linefeed :
Learning Java

Line 6

System.out.println("Single Quote : \'Learning Java\');

This line will display the following on the screen:

Single Quote : 'Learning Java'

Line 7

System.out.println("Double Quote : \"Learning Java\");

This line will display the following on the screen:

Double Quote : "Learning Java"

Line 8

System.out.println("Backslash : \\Learning Java\\");

This line will display the following on the screen:

Backslash : \Learning Java\

Line 9

System.out.println("Horizontal Tab : Learning\tJava");

This line will display the following on the screen:

Horizontal Tab : Learning Java

Line 10

System.out.println("Backspace : Learning\bJava");

This line will display the following on the screen:

Backspace : LearninJava

Line 11

System.out.println("Carriage Return: Learning\rJava");

This line will display the following on the screen:

Javaiaage Return: Learning

The output of Example 1 is displayed in Figure 2-1.


```

C:\Windows\system32\cmd.exe

D:\Java Projects\Ch02>javac Escape.java
D:\Java Projects\Ch02>java Escape
Linefeed      :
Learning Java  :
Single Quote   : 'Learning Java'
Double Quote   : "Learning Java"
Backslash      : \Learning Java\
Horizontal Tab : Learning      Java
Backspace      : LearninJava
Java'sage Return: Learning

D:\Java Projects\Ch02>_

```

Figure 2-1 The output of Example 1

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 when the program is being executed.

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 this 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. To name a variable, you need to follow certain rules. As discussed earlier, identifier is the name used for variables, classes, and so on. The rules to name a variable are same as discussed earlier with identifiers.

For example, you can declare an integer type variable **age**, as follows:

```
int age;
```

When this statement executes, the compiler allocates 4 bytes (size of **int** data type is 4 bytes) of memory to the variable **age**. Now, the variable **age** is treated as a reference to the allocated memory location.

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 this 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 this 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**.

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 this 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.

Example 2

The following program illustrates the concept of dynamic initialization of a variable. The program will calculate the average of three numbers, assign the resultant value to another variable, and display it on the screen.

```
//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 }
```

Explanation

Line 5

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 6

float avg;

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

Line 7

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** are added. After that, the resultant value 57 is divided by 3. Next, the resultant value 19.0 is assigned to the variable **avg** at the execution time.

Line 8

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

This line will display the following on the screen:

The average of three numbers is: 19.0



Note

In line 8, the + sign is used to concatenate the value of the variable **avg** to the given string.

The output of Example 2 is displayed in Figure 2-2.

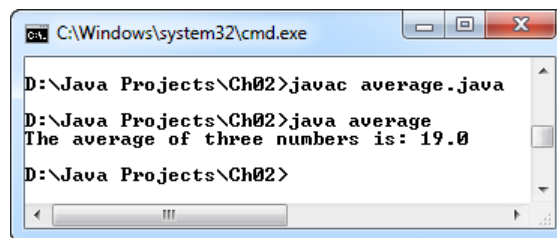


Figure 2-2 The output of Example 2

Types of Variables

There are four types of variables in Java as given next:

- a. Local variables
- b. Instance variables
- c. Class/Static variables
- d. Method Parameter variables

Local Variables

The variables that are declared inside a block of code or within the body of a method (or constructor) are known as local variables.

Constraints followed by local variables are:

- They have local scope; they cannot be accessed outside the block in which they are defined. These variables are accessible only within that particular block.
- Access modifiers cannot be used for local variables.
- There is no default value for the local variables. These variables should be declared and an initial value should be assigned to them before the first use.

For example:

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

In this 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, constructor, or access modifiers 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 and are not declared as static.

For example:

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

In this 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/Static Variables

Class variables are the same as the instance variables except that these variables are declared with the **static** keyword. These variables cannot be local. Regardless of the number of times a class has been instantiated, only one copy of static or class variable is created. You will learn about the **static** data in later chapters.

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 **static** keyword and treated as the class variables.

Method Parameter Variables

Method parameter variables are the variables that are declared in the method declaration signature. Whenever a java method is invoked, a variable is created with the same name as it is declared. Like local variables, it does not have any default value. So, an initial value should be assigned for it, otherwise compiler will give an error.

For example:

```
void demo_method( int a, int b)
{
    -----;
    -----;
}
```

In this example, **demo_method()** is a method and variables **a** and **b** are parameters to this method.

As you know that **public static void main(String[] arg)** is the main method which is the entry point of any program, the variable **arg** is the parameter to this method. The important thing to remember is that parameters are always classified as “variables” not “fields”. This applies to other parameters (such as constructors and exception handlers) that you’ll learn in later chapters.

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 four types of variables discussed earlier 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.

TYPE CONVERSION

Type conversion means converting one data type into another, also known as Type Casting. 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. Implicit conversion (Widening conversion)
- b. Explicit conversion (Narrowing conversion)

Implicit Conversion (Widening Conversion)

The implicit conversion takes place when the destination data type is larger than the source data type and both the data types are compatible. It is also known as automatic conversion. For example, a data element of **short** type is converted into the **int** type. In such cases, Java performs implicit conversion because the **int** data type is larger than the **short** data type and both the data types are compatible. In implicit conversion, no information is lost during the conversion.

Example 3

The following program will convert a data element of **byte** type into the **int** type by using the concept of implicit type conversion 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 }
```

Explanation

Line 5

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 6

```
int dest;
```

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

Line 7

```
dest= src;
```

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

Line 8

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

This line will display the following on the screen:

```
dest = 127
```

The output of Example 3 is displayed in Figure 2-3.

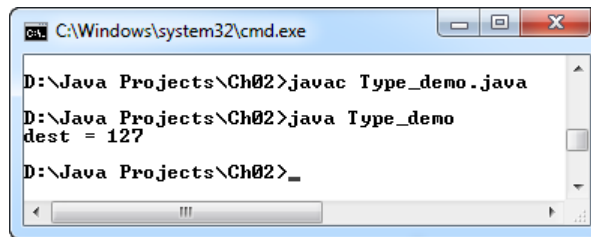


Figure 2-3 The output of Example 3

Explicit Conversion (Narrowing Conversion)

In the previous section, you learned about the 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 this syntax, the **destination_data_type** specifies the data type in which you want to convert the value, which is specified by **value**.



Tip

The thumb rule for explicit conversion is that the same data type should exist on both sides.

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

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

In this 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**.

Example 4

The following program will convert an **int** type into a **byte** type using the concept of explicit type conversion 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 }
```

Explanation

Line 5

byte b;

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

Line 6

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 7

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 8

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

This line will display the following on the screen:

After conversion, value of b is: 44

The output of Example 4 is displayed in Figure 2-4.

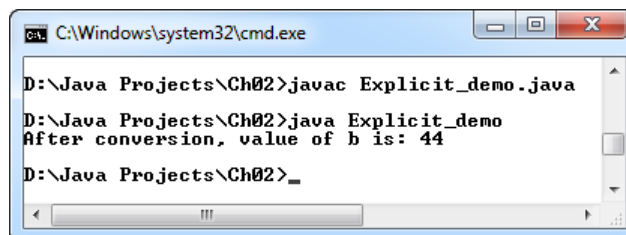


Figure 2-4 The output of Example 4



Note
A boolean value cannot be assigned to any other data type. Boolean is incompatible for conversion. Boolean value can be assigned only to another boolean.

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 different categories, which are as follows:

- a. Unary operator
- b. Arithmetic operators
- c. Bitwise operators
- d. Relational operators
- e. Logical operators
- f. Assignment operators
- g. Miscellaneous operators

Unary Operators

Operators that require only one operand are known as unary operators. Table 2-4 lists all unary operators that are used in Java.

Table 2-4 Unary operators with their syntax

Operator	Description	Syntax
+	Unary plus operator; indicates positive value (numbers are positive without this)	var1 = +var2
-	Unary minus operator; negates an expression	var1 = -var2
++	Increment operator; increments a value by 1	var1 = var2++, var1 = ++var2
--	Decrement operator; decrements a value by 1	var1 = var2--, var1 = --var2
!	Unary Compliment operator; inverts the value of a boolean	!var1

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; refer to syntax shown in Table 2-4.

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

- a. Postfix notation
- b. 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++;      //increment
var1--;      //decrement
```

In this syntax, the increment and decrement operators (`++` and `--`) are used after the operand **var1**. It will increase and decrease 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 this 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;
--var1;
```

In these syntaxes, the increment and decrement operator (`++` and `--`) is used before the operand **var1**. It will increase or decrease 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 this 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;
```

Example 5

The following program will perform all the unary operations and display the resultant values on the screen.

//Write a program to perform various unary operations

```
1  class UnaryOp_Demo
2  {
3      public static void main(String[] arg)
4      {
5          int result,res=+10;
6          System.out.println("Unary plus Operator result is " +res);
7          res = -res;
8          System.out.println("Unary Minus Operator result is " +res);
9          result=res++;
10         System.out.println("Post-increment result is " +result);
11         result=++res;
12         System.out.println("Pre-increment result is " +result);
13         result=res--;
14         System.out.println("Post-decrement result is " +result);
15         result=--res;
16         System.out.println("Pre-decrement result is " +result);
17         boolean success = false;
18         System.out.println("Result without compliment operator is "
19                             +success);
19         System.out.println("Result with compliment operator is "+!success);
20     }
21 }
```

Explanation

Line 5

int result,res= +10;

In this line, **result** and **res** are declared as an integer type variables and **+10** is assigned as the initial value to variable **res**. **+** is a unary operator which indicates positive value. If you do not use unary plus operator, still it indicates the positive value.

Line 6

System.out.println("Unary plus Operator result is " +res);

This line will display the following on the screen:

Unary plus Operator result is 10

Line 7

```
res = -res;
```

In this line, **res** is the variable and its value is updated to negative value by using **unary minus** operator (-).

Line 8

```
System.out.println("Unary Minus Operator result is " + res);
```

This line will display the following on the screen:

Unary Minus Operator result is -10

Line 9

```
result=res++;
```

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

Line 10

```
System.out.println("Post-increment result is " + result);
```

This line will display the following on the screen:

Post-increment result is -10

Line 11

```
result=++res;
```

In this line, first the value (-9) of the variable **res** is incremented by 1. Next, it is assigned to the variable **result**.



Note

*In Line 10, the output is -10. But in Line 11, the initial value of **res** variable is -9 because in post increment, first the value is assigned to the variable then incremented by 1. So, Line 10 shows the assigned value as output and incremented value (-9) is stored in the memory.*

Line 12

```
System.out.println("Pre-increment result is " + result);
```

This line will display the following on the screen:

Pre-increment result is -8

Line 13

```
result=res--;
```

In this line, first the value (-8) of the variable **res** is assigned to the variable **result**. Next, it is decremented by 1.

Line 14

```
System.out.println("Post-decrement result is " + result);
```

This line will display the following on the screen:

Post-decrement result is -8

Line 15

```
result=--res;
```

In this line, first the value (-9) of the variable **res** is decremented by 1. Next, it is assigned to the variable **result**.

Line 16

```
System.out.println("Pre-decrement result is " +result);
```

This line will display the following on the screen:

Pre-decrement result is -10

Line 17

```
boolean success = false;
```

In this line, success is declared as a boolean type variable and **false** is assigned as an initial value to it.

Line 18

```
System.out.println("Result without compliment operator is " +success);
```

This line will display the following on the screen:

Result without compliment operator is false

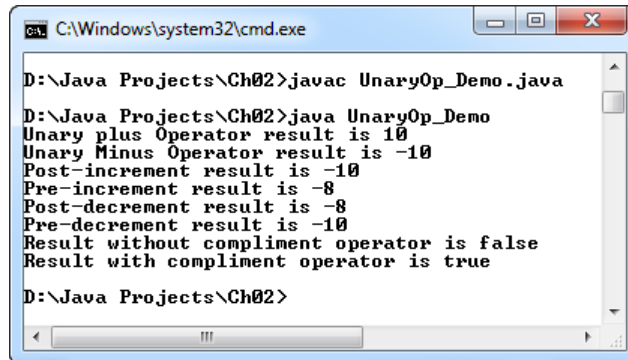
Line 19

```
System.out.println("Result with compliment operator is " +!success);
```

This line will display the following on the screen:

Result with compliment operator is true

The output of Example 5 is displayed in Figure 2-5.



```
C:\Windows\system32\cmd.exe

D:\Java Projects\Ch02>javac UnaryOp_Demo.java
D:\Java Projects\Ch02>java UnaryOp_Demo
Unary plus Operator result is 10
Unary Minus Operator result is -10
Post-increment result is -10
Pre-increment result is -8
Post-decrement result is -8
Pre-decrement result is -10
Result without compliment operator is false
Result with compliment operator is true

D:\Java Projects\Ch02>
```

Figure 2-5 The output of Example 5

Arithmetic Operators

Operators that are used in mathematical expressions are known as arithmetic operators. Table 2-5 lists all arithmetic operators that are used in Java.

Table 2-5 Arithmetic Operators with their syntax

Operator	Description	Syntax
+	Addition	var1=var2 + var3
-	Subtraction	var1=var2 - var3
*	Multiplication	var1=var2 * var3
/	Division	var1=var2 / var3
%	Modulus Operator; gives remainder	var1=var2 % var3

Example 6

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

/ Write a program to perform various arithmetic operations on two numbers using the arithmetic operators: */*

```
1  class Arith_operators
2  {
3      public static void main(String arg[])
4      {
5          int val1=30, val2=10;
6          int sum= val1+val2;
7          int sub= val1-val2;
8          int mul= val1*2;
9          int div= val1/val2;
10         int mod=mul%7;
11         System.out.println("Value 1 = " +val1);
12         System.out.println("Value 2 = " +val2);
13         System.out.println("Addition = " +sum);
14         System.out.println("Subtraction = " +sub);
15         System.out.println("Multiplication = " +mul);
16         System.out.println("Division = " +div);
17         System.out.println("Modulus = " +mod);
18     }
19 }
```

Explanation

Line 5

int val1=30, val2=10;

In this line, **val1** and **val2** are declared as integer type variables and their initial values are assigned as 30 and 10, respectively.

Line 6

```
int sum= val1+val2;
```

In this line, the value (30) of the variable **val1** is added to the value (10) of the variable **val2** and the resultant value (40) is assigned to the integer variable **sum**.

Line 7

```
int sub= val1-val2;
```

In this line, the value (10) of the variable **val2** is subtracted from the value (30) of the variable **val1** and the resultant value (20) is assigned to the integer variable **sub**.

Line 8

```
int mul= val1*2;
```

In this line, the value (30) of the variable **val1** is multiplied by 2 and the resultant value (60) is assigned to the integer variable **mul**.

Line 9

```
int div= val1/val2;
```

In this line, the value (30) of the variable **val1** is divided by the value (10) of the variable **val2** and the resultant value (3), which represents the quotient, is assigned to the integer variable **div**.

Line 10

```
int mod=mul%7;
```

In this line, the value (60) of the variable **mul** is divided by 7 and the resultant value (4), which represents the remainder is assigned to the integer variable **mod**. Therefore, **mul%7** returns the remainder value 4.

Line 11

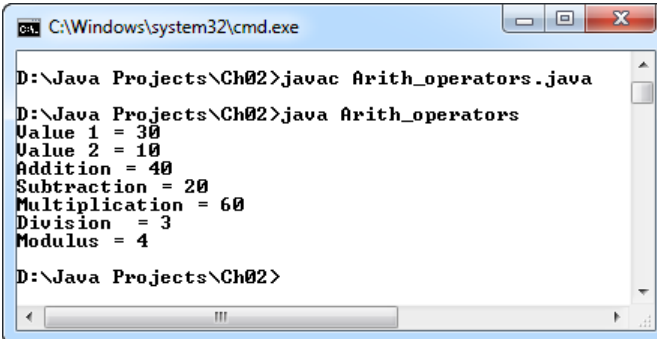
```
System.out.println("Value 1 =" +val1);
```

This line will display the following on the screen:

Value 1=30

The working of lines 12 to 17 is similar to that of line 11.

The output of Example 6 is displayed in Figure 2-6.



```
C:\Windows\system32\cmd.exe

D:\Java Projects\Ch02>javac Arith_operators.java
D:\Java Projects\Ch02>java Arith_operators
Value 1 = 30
Value 2 = 10
Addition = 40
Subtraction = 20
Multiplication = 60
Division = 3
Modulus = 4

D:\Java Projects\Ch02>
```

Figure 2-6 The output of Example 6

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 2-6 shows a list of bitwise operators.

Table 2-6 Bitwise Operators

Operator	Operation
~	Bitwise Compliment
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
>>	Right Shift
<<	Left Shift
>>>	Zero Fill Right Shift

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

The Bitwise Compliment (~) Operator

The bitwise Compliment (~) 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 unary NOT operator. The syntax for using the compliment (~) operator is as follows:

```
~ value or expression;
```

For example:

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

In this 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 2-7 represents all (**~**, **&**, **|**, and **^**) bitwise logical operators.

Table 2-7 Bitwise Logical Operators

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

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

The Right Shift (>>) Operator

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 this 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;
```

This example operates in the following way:

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

When the >> operator operates on the given bits for the first time, the right most bit, bit 1, is lost and all other bits shifts 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
```

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 this 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;
```

This example operates in the following way:

00010001 //Bits representing the value 17

When the << operator operates on the given bits for the first time, the leftmost bit, bit 0 is lost and all other bits shifts 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

The Zero Fill Right Shift (>>>) Operator

The zero fill right shift (>>>) operator is used to move all the bits of an operand to the right direction and shifted values are filled up with zeros. It is also known as Unsigned right shift operator. 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 this 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 = 10;  
int b = a>>>2;
```

This example operates in the following way:

00001010 //Bits representing the value 10

When the >>> operator operates on the given bits for the first time, the rightmost bit (bit 1), is lost and all other bits shift to the right by filling the left most bit with zero. The bit pattern which is produced after the first step is as follows:

00000101 //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:

00000010 //Bits representing the value 4



Note

Difference between the right shift (>>) and the zero fill right shift (>>>) operator is the sign extension. The zero fill right shift operator ">>>" shifts a zero into the leftmost position, whereas in right shift (>>), the leftmost position depends on sign extension.

Example 7

The following program will perform all the bitwise operations and display the resultant values on the screen.

```
//Write a program to perform various bitwise operations
1  class BitwiseOp_demo
2  {
3      public static void main(String args[])
4      {
5          int a = 15, b = 10, c = 0;
6          c = a & b;          /* 10 = 0000 1010 */
7          System.out.println("a & b = " + c );
8          c = a | b;          /* 15 = 0000 1111 */
9          System.out.println("a | b = " + c );
10         c = a ^ b;          /* 5 = 0000 0101 */
11         System.out.println("a ^ b = " + c );
12         c = ~a;             /* -16 = 1111 0000 */
13         System.out.println("~a = " + c );
14         c = a << 2;          /* 60 = 0011 1100 */
15         System.out.println("a << 2 = " + c );
16         c = a >> 2;          /* 3 = 0000 0011 */
17         System.out.println("a >> 2 = " + c );
18         c = a >>> 2;         /* 3 = 0000 1111 */
19         System.out.println("a >>> 2 = " + c );
20     }
21 }
```

Explanation

Line 5

int a = 15, b = 10, c = 0;

In this line, **a**, **b** and **c** are declared as integer type variables and 15, 10 and 0 are assigned as initial value to them.

Line 6

c = a & b;

In this line, **&** is the bitwise AND operator between **a** and **b** variables which is performing the bitwise AND operation and the resultant value is assigned to the variable **c**.

Line 7

```
System.out.println("a & b = " + c);
```

This line will display the following on the screen:

a & b = 10

Line 8

```
c = a | b;
```

In this line, | is the bitwise OR operator between **a** and **b** variables which is performing the bitwise OR operation and the resultant value is assigned to the variable **c**.

Line 9

```
System.out.println("a | b = " + c);
```

This line will display the following on the screen:

a | b = 15

Line 10

```
c = a ^ b;
```

In this line, ^ is the bitwise XOR operator between **a** and **b** variables which is performing the bitwise XOR operation and the resultant value is assigned to the variable **c**.

Line 11

```
System.out.println("a ^ b = " + c);
```

This line will display the following on the screen:

a ^ b = 5

Line 12

```
c = ~a;
```

In this line, ~ is the bitwise complement operator which is performing complement of variable **a** and the resultant value is assigned to the variable **c**.

Line 13

```
System.out.println("~a = " + c);
```

This line will display the following on the screen:

~a = -16

Line 14

```
c = a << 2;
```

In this line, << is the bitwise left shift operator which moves all the bits of variable **a** to the left direction by 2 bits. In this process, the leftmost bits are lost and the resultant value is assigned to the variable **c**.

Line 15

```
System.out.println("a << 2 = " + c);
```

This line will display the following on the screen:

a << 2 = 60

Line 16

```
c = a >> 2;
```

In this line, `>>` is the bitwise right shift operator which moves all the bits of variable **a** to the right direction by 2 bits. In this process, the right most bits are lost, and the resultant value is assigned to the variable **c**.

Line 17

```
System.out.println("a >> 2 = " + c);
```

This line will display the following on the screen:

```
a >> 2 = 3
```

Line 18

```
c = a >>> 2;
```

In this line, `>>>` is the bitwise zero fill right shift operator which moves all the bits of variable **a** to the right direction by 2 bits. The shifted values are filled by zero and the resultant value is assigned to the variable **c**.

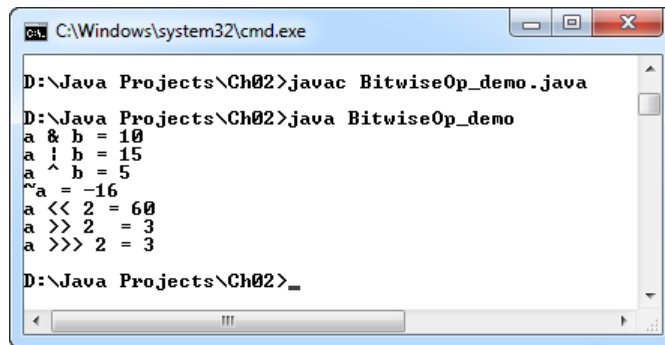
Line 19

```
System.out.println("a >>> 2 = " + c);
```

This line will display the following on the screen:

```
a >>> 2 = 3
```

The output of Example 7 is displayed in Figure 2-7.



```

C:\Windows\system32\cmd.exe

D:\Java Projects\Ch02>javac BitwiseOp_demo.java
D:\Java Projects\Ch02>java BitwiseOp_demo
a & b = 10
a | b = 15
a ^ b = 5
~a = -16
a << 2 = 60
a >> 2 = 3
a >>> 2 = 3
D:\Java Projects\Ch02>_

```

Figure 2-7 The output of Example 7

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 2-8 shows the list of relational operators and their syntax.

Table 2-8 Relational operators and their syntax

Operator	Operation	Syntax
==	Equal to	var1==var2
!=	Not equal to	var1!=var2
>	Greater than	var1>var2
<	Less than	var1<var2
>=	Greater than equal to	var1>=var2
<=	Less than equal to	var1<=var2

In the syntax shown in Table 2-8, the relational operators are used to check the relation between the two variables, **var1** and **var2**. If the values of variable **var1** and **var2** satisfy the condition then the outcome of this operation is true. Otherwise, it is false.

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 another set 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 2-9 shows the list of logical operators.

Table 2-9 The Logical Operators

Operator	Operation
&&	Logical Short-circuiting AND Operator
	Logical Short-circuiting OR Operator
!	Logical NOT Operator

Among these operators, the working of Logical NOT (!) operator is the same as the unary Complement Operator which is used for inverting a **boolean** value from **false** to **true** and vice-versa.

For example:

```
!(x == y)
```

In this example, the **==** (equal to) operator is used to check the equality between the two variables, **x** and **y**. If the outcome of the relational expression (**x == y**) is true then this outcome is inverted by logical not operator. Therefore, the final outcome is false.

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

The short-circuit **&&** and **||** operators are mostly used in 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 2-10 shows the working of short-circuit **&&** and **||** operators.

Table 2-10 Logical Operators

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

In Table 2-10, 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.

The Assignment (=) Operators

Assignment operators are used to assign value to a variable. These operators can be categorized as follows:

- Simple assignment operator
- Compound assignment operator

Simple Assignment Operator

Simple assignment operator is denoted by the single equal (=) symbol and is used to assign a value to a variable. The = operator has already been discussed in the previous examples of this chapter. 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. You need to assign a variable on the left side and the value to be assigned to this variable is always placed on the right side of the operator. The value assigned on the right 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 this syntax, the same value represented by **value** is assigned to all the three variables **var1**, **var2**, and **var3**. The assignment operator is evaluated from right to left; therefore **var1 = var2 = var3 = 0**; would assign 0 to var3, then var3 to var2, then var2 to var1.

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. Compound assignment operators are also known as Short hand assignment operators. Table 2-11 shows the list of compound assignment operators with their syntax.

Table 2-11 Compound assignment operators and their syntax

Operator	Description	Syntax	Equivalent Expression
+=	It adds right operand to the left operand and assigns the result to the left operand.	<code>var1+=var2;</code>	<code>var1=var1+var2;</code>
-=	It subtracts right operand from the left operand and assigns the result to the left operand.	<code>var1-=var2;</code>	<code>var1=var1-var2;</code>
=	It multiplies right operand to the left operand and assigns the result to the left operand.	<code>var1=var2;</code>	<code>var1=var1*var2;</code>
/=	It divides left operand with the right operand and assigns the result to the left operand.	<code>var1/=var2;</code>	<code>var1=var1/var2;</code>
%=	It takes modulus using left operand and right operand, and assigns the result to the left operand.	<code>var1%=var2;</code>	<code>var1=var1%var2;</code>
&=	Bitwise AND assignment operator	<code>var1&=var2;</code>	<code>var1=var1&var2;</code>
 =	Bitwise OR assignment operator	<code>var1 =var2;</code>	<code>var1=var1 var2;</code>
^=	Bitwise XOR assignment operator	<code>var1^=var2;</code>	<code>var1=var1^var2;</code>
<<=	Bitwise left shift assignment operator	<code>var1<<=2;</code>	<code>var1=var1<<2;</code>
>>=	Bitwise right shift assignment operator	<code>var1>>=2;</code>	<code>var1=var1>>2;</code>
>>>=	Bitwise zero right shift assignment operator	<code>var1>>>=2;</code>	<code>var1=var1>>>2;</code>

In the syntax as shown in Table 2-11, first the given operation is performed on the variable **var1** and **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 ;
```

Example 8

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

//Write a program to perform assignment operations

```
1  class Assign_demo
2  {
3      public static void main(String args[])
4      {
5          int var = 10, result = 0;
6          result += var ;//10
7          System.out.println("result += var : " + result );
8          result *= var ;//100
9          System.out.println("result *= var : " + result );
10         result -= var ;//90
11         System.out.println("result -= var : " + result );
12         result /= var ;//9
13         System.out.println("result /= var : " + result );
14         result %= var ;//9
15         System.out.println("result %= var : " + result );
16         result ^= var ;//3
17         System.out.println("result ^= var = " + result );
18         result |= var ;//11
19         System.out.println("result |= var = " + result );
20         result &= var ;//10
21         System.out.println("result &= var = " + result );
22         result <=<= 2 ;//40
23         System.out.println("result <=<= 2 = " + result );
24         result >>= 2 ;//10
25         System.out.println("result >>= 2 = " + result );
26         result >>>= 3 ;//1
27         System.out.println("result >>>= 3 = " + result );
28     }
29 }
```

Explanation

Line 5

```
int var = 10, result = 0;
```

In this line, **var** and **result** are declared as integer type variables and 10 and 0 are assigned as initial value to them.

Line 6

```
result += var ;
```

In this line, first the value (0) of the variable **result** is added to the value (10) of the variable **var**. Next, the resultant value is assigned back to the variable **result**.

Line 7

```
System.out.println("result += var : " + result );
```

This line will display the following on the screen:

```
result += var : 10
```

Line 8

```
result *= var ;
```

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

Line 9

```
System.out.println("result *= var : " + result );
```

This line will display the following on the screen:

```
result *= var : 100
```

Line 10

```
result -= var ;
```

In this line, first the value (10) of the variable **var** is subtracted from the value (100) of the variable **result**. Next, the resultant value is assigned back to the variable **result**.

Line 11

```
System.out.println("result -= var : " + result );
```

This line will display the following on the screen:

```
result -= var : 90
```

Line 12

```
result /= var ;
```

In this line, first the value (90) of the variable **result** is divided by the value (10) of the variable **var**. Next, the resultant (quotient) value is assigned back to the variable **result**.

Line 13

```
System.out.println("result /= var : " + result );
```

This line will display the following on the screen:

```
result /= var : 9
```

Line 14

```
result %= var ;
```

In this line, first the value (9) of the variable **result** is divided by the value (10) of the variable **var**. Next, the remainder value is assigned back to the variable **result**.

Line 15

```
System.out.println("result %= var : " + result );
```

This line will display the following on the screen:

```
result %= var : 9
```

Line 16

```
result ^= var ;
```

In this line, first the bitwise XOR operation is performed between the variables **result** and **var** whose values are 9 and 10, respectively. Next, the resultant value is assigned back to the variable **result**.

Line 17

```
System.out.println("result ^= var : " + result );
```

This line will display the following on the screen:

```
result ^= var : 3
```

Line 18

```
result |= var ;
```

In this line, first the bitwise OR operation is done between the variables **result** and **var** whose values are 3 and 10, respectively. Next, the resultant value is assigned back to the variable **result**.

Line 19

```
System.out.println("result |= var : " + result );
```

This line will display the following on the screen:

```
result |= var : 11
```

Line 20

```
result &= var ;
```

In this line, first the bitwise AND operation is done between the variables **result** and **var** whose values are 3 and 10, respectively. Next, the resultant value is assigned back to the variable **result**.

Line 21

```
System.out.println("result &= var : " + result );
```

This line will display the following on the screen:

```
result &= var : 10
```

Line 22

```
result <<= 2 ;
```

In this line, first the bitwise left shift operation is performed on the variable **result** whose value is 10. It moves all the bits of variable **result** to the left direction by 2 bits and the left most bits are lost. Next, the resultant value is assigned back to the variable **result**.

Line 23

```
System.out.println("result <= 2 : " + result );
```

This line will display the following on the screen:

```
result <= 2 : 40
```

Line 24

```
result >>= 2 ;
```

In this line, first the bitwise right shift operation is performed on the variable **result** whose value is 40. It moves all the bits of variable **result** to the right direction by 2 bits and the right most bits are lost. Next, the resultant value is assigned back to the variable **result**.

Line 25

```
System.out.println("result >>= 2 : " + result );
```

This line will display the following on the screen:

```
result >>= 2 : 10
```

Line 26

```
result >>>= 2 ;
```

In this line, first the bitwise zero fill right shift operation is performed on the variable **result** whose value is 10. It moves all the bits of variable **result** to the right direction by 2 bits and the shifted values are filled by zero. Next, the resultant value is assigned back to the variable **result**.

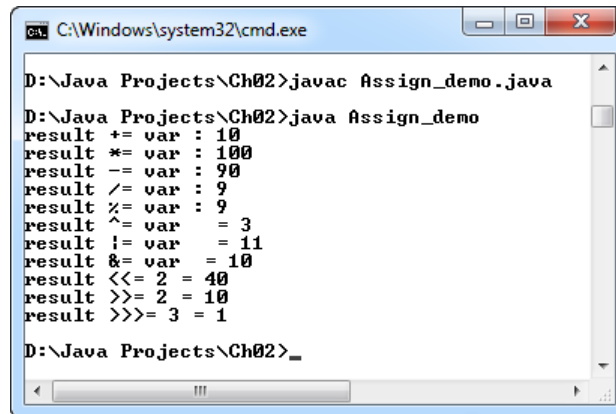
Line 27

```
System.out.println("result >>>= 2 : " + result );
```

This line will display the following on the screen:

```
result >>>= 2 : 1
```

The output of Example 8 is displayed in Figure 2-8.



```
C:\Windows\system32\cmd.exe

D:\Java Projects\Ch02>javac Assign_demo.java

D:\Java Projects\Ch02>java Assign_demo
result += var : 10
result *= var : 100
result -= var : 90
result /= var : 9
result %= var : 9
result ^= var : 3
result |= var : 11
result &= var : 10
result <<= 2 = 40
result >>= 2 = 10
result >>>= 3 = 1

D:\Java Projects\Ch02>_
```

Figure 2-8 The output of Example 8

The ? : Operator

The **? :** operator is also known as the ternary operator because it works on three operands. The first operand is a boolean expression. If the expression is true then it returns second operand, else it returns third operand. It is a conditional operator that provides a shorter syntax for the **if-else** statement (discussed in the later chapters). The syntax for using the **? :** operator is as follows:

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

In this syntax, if the condition specified by **conditional_expression** results in true, the **statement 1** is executed. Otherwise, the **statement 2** is executed.

For example:

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

In this 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**.

Example 9

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

```
//Write a program to find the greater number
1  class Ternary_demo
2  {
3      public static void main(String arg[])
4      {
5          int a=20, b=11, c;
6          c= a>b ? a : b;
7          System.out.println("The greater value is: " +c);
8      }
9  }
```

Explanation

Line 6

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** is assigned to the integer variable **c**.

Line 7

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

This line will display the following on the screen:

The greater value is: 20

The output of Example 9 is displayed in Figure 2-9.

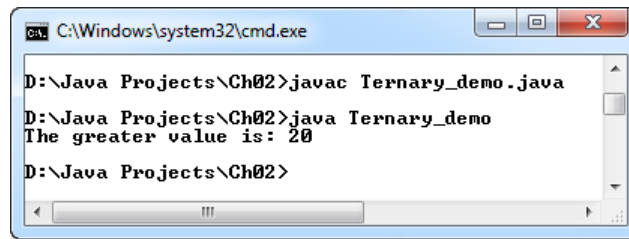


Figure 2-9 The output of Example 9

The instanceof Operator

The **instanceof** operator is used to check whether the object is an instance of the specified type (class or subclass or interface) at runtime. It is also known as type comparison operator because it compares the instance with type. If object is of the specified type, then the **instanceof** operator evaluates to true. Otherwise, the result is false. If you apply the **instanceof** operator with any variable that has a null value, it returns false.

The syntax for the **instanceof** operator is as follows:

```
object_name instanceof class_name
```

Example 10

The following program will check whether the object is an instance of the class by using the **instanceof** operator and display the resultant value on the screen.

//Write a program to check whether the object is an instance of the class

```
1  class Instanceof_demo
2  {
3      public static void main(String args[])
4      {
5          Instanceof_demo id=new Instanceof_demo();
6          boolean i=id instanceof Instanceof_demo;
7          System.out.println( "value:" +i);
8      }
9  }
```

Explanation

Line 5

Instanceof_demo id=new Instanceof_demo();

In this line, an **id** object of **instanceof_demo** class is created.

Line 6

boolean i=id instanceof Instanceof_demo;

In this line, **instanceof** operator is used to check whether the **id** object is instance of **Instanceof_demo** class.

Line 7

```
System.out.println("value:" + i);
```

This line will display the following on the screen:

value: true

The output of Example 10 is displayed in Figure 2-10.

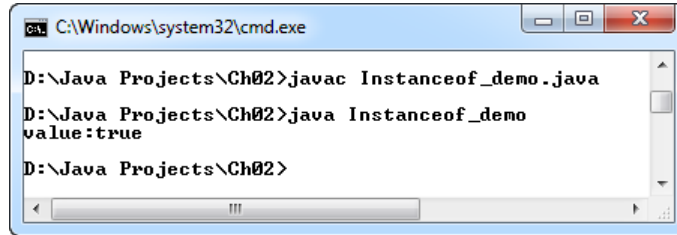


Figure 2-10 The output of Example 10

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.

All binary operators except the assignment operators are evaluated from left to right. When operators of equal precedence appear in the same expression then they are evaluated from left to right whereas assignment operators are evaluated from right to left.

A list of Java operators arranged from the highest to the lowest precedence is given in the Table 2-12.



Note

In the operator precedence 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.

For example:

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

The multiplication operator (*) has a higher precedence than the addition (+) and the assignment operators (=). Therefore, in the given 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**.

Table 2-12 Operator Precedence

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

COMMAND-LINE ARGUMENTS

You must have noted that in all examples explained in this book, no information was passed to the program during the time of its execution. But sometimes, you need to pass some information to a program during the time of its execution. This can be done by using the command-line arguments. You can simply pass these arguments by appending them after the name of the program during the time of its execution. The command-line arguments that are passed during the execution time are stored in the String type array **arg[]** of the **main()** method.

For example:

```
D:\Java Projects\Ch02>java demo How are you
```

In this example, **demo** is the program name and **How are you** are the command-line arguments. Here, the first argument **How** is stored at **arg[0]**, **are** is stored at **arg[1]**, and so on.

Example 11

The following program illustrates the use of the command-line arguments. The program will display all the command-line arguments entered by a user.

//Write a program to display the command-line arguments

```

1  class Commandline_demo
2  {
3      public static void main(String arg[])
4      {
5          System.out.println("First argument is: " +arg[0]);
6          System.out.println("Second argument is: " +arg[1]);
7      }
8  }

```

Explanation

In this example, the arguments are passed by the user during the time of program's execution at 0th and 1st position of array **arg[]**.

Now, execute the program using the following statement:

```
java Commandline_demo Hello User
```

The output of Example 11 is displayed in Figure 2-11.

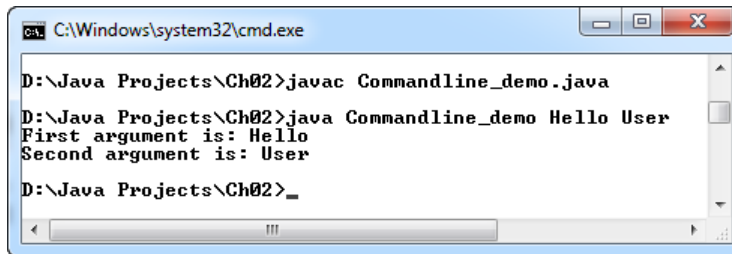


Figure 2-11 The output of Example 11

As you know, you can pass only string type array through command line arguments because array **arg[]** is of string type. If you want to pass values other than string then you will have to convert it into other types. You can use different ways for these conversions which are discussed next.

String to int

We can convert **String** to **int** in Java by using **Integer.parseInt()** method. Whenever users want to do any mathematical operation, they need numbers. But when they pass numbers, JVM treats them as **String** type. In such cases, users have to convert these values from **String** to **int**. To do so, they will use **Integer.parseInt()** method.

For example:

```
int x=Integer.parseInt(arg[0]);
```

In this example, **parseInt** is the method of the **Integer** class and is used to read numeric values from the command-line arguments. Next, the resultant value will be assigned to the integer type variable **x**.

String to long

We can convert **String** to **long** in Java by using the **Long.parseLong()** method. Whenever users want to do any mathematical operations on long numbers, they need to convert **String** to **long** by using **Long.parseLong()** method.

For example:

```
long y=Long.parseLong(arg[0]);
```

In this example, **parseLong** is the method of the **Long** class and is used to read long numeric values from the command-line arguments. Next, the resultant value will be assigned to the long type variable **y**.

String to float

We can convert **String** to **float** in Java by using **Float.parseFloat()** method. Whenever users want to do any mathematical operations on float numbers, they need to convert **String** to **float** by using the **Float.parseFloat()** method.

For example:

```
float z=Float.parseFloat(arg[0]);
```

In this example, **parseFloat** is the method of the **Float** class and is used to read float numeric values from the command-line arguments. Next, the resultant value will be assigned to the float type variable **z**.

Example 12

The following program illustrates the use of **String** to **int** conversion through command-line arguments. The program will calculate the sum of two integer values entered by the user and display the resultant value on the screen.

//Write a program to calculate the sum of two integer numbers entered by user.

```
1  class Command_demo
2  {
3      public static void main(String arg[])
4      {
5          int a,b,c;
6          a= Integer.parseInt(arg[0]);
7          b= Integer.parseInt(arg[1]);
8          c=a+b;
9          System.out.println("Addition = " +c);
10     }
11 }
```

Explanation

Lines 6 and 7

```
int a = Integer.parseInt(arg[0]);
int b = Integer.parseInt(arg[1]);
```

In these lines, **parseInt** is the method of the **Integer** class and is used to read numeric values from the command-line arguments. Next, the resultant values will be assigned to the integer type variables **a** and **b**, respectively.

The output of Example 12 is displayed in Figure 2-12.

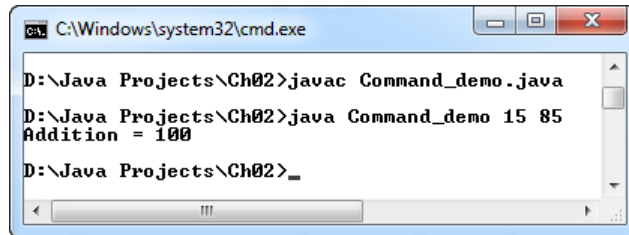


Figure 2-12 The output of Example 12

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 primitive 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 **static** keyword. (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 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.
6. Explain **instanceof** operator.
7. Differentiate between primitive and user defined data types.
8. Find errors in the following program statements:

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

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

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

```

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

(e) 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 shift the value 200 to the right by two positions using the (shift right) >> operator.

Exercise 2

Write a program to calculate the area of a circle whose radius is entered by the user.

Answers to Self-Evaluation Test

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