**Java -DataTypes**

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java −

* Primitive Data Types
* Reference/Object Data Types



## **Primitive Data Types**

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

### **byte**

* Byte data type is an 8-bit signed two's complement integer
* Minimum value is -128 (-2^7)
* Maximum value is 127 (inclusive)(2^7 -1)
* Default value is 0
* Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
* Example: byte a = 100, byte b = -50

### **short**

* Short data type is a 16-bit signed two's complement integer
* Minimum value is -32,768 (-2^15)
* Maximum value is 32,767 (inclusive) (2^15 -1)
* Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
* Default value is 0.
* Example: short s = 10000, short r = -20000

### **int**

* Int data type is a 32-bit signed two's complement integer.
* Minimum value is - 2,147,483,648 (-2^31)
* Maximum value is 2,147,483,647(inclusive) (2^31 -1)
* Integer is generally used as the default data type for integral values unless there is a concern about memory.
* The default value is 0
* Example: int a = 100000, int b = -200000

### **long**

* Long data type is a 64-bit signed two's complement integer
* Minimum value is -9,223,372,036,854,775,808(-2^63)
* Maximum value is 9,223,372,036,854,775,807 (inclusive)(2^63 -1)
* This type is used when a wider range than int is needed
* Default value is 0L
* Example: long a = 100000L, long b = -200000L

### **float**

* Float data type is a single-precision 32-bit IEEE 754 floating point
* Float is mainly used to save memory in large arrays of floating point numbers
* Default value is 0.0f
* Float data type is never used for precise values such as currency
* Example: float f1 = 234.5f

### **double**

* double data type is a double-precision 64-bit IEEE 754 floating point
* This data type is generally used as the default data type for decimal values, generally the default choice
* Double data type should never be used for precise values such as currency
* Default value is 0.0d
* Example: double d1 = 123.4

### **boolean**

* boolean data type represents one bit of information
* There are only two possible values: true and false
* This data type is used for simple flags that track true/false conditions
* Default value is false
* Example: boolean one = true

### **char**

* char data type is a single 16-bit Unicode character
* Minimum value is '\u0000' (or 0)
* Maximum value is '\uffff' (or 65,535 inclusive)
* Char data type is used to store any character
* Example: char letterA = 'A'

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Default Value** | **Default size** |
| Boolean | False | 1 bit |
| Char | '\u0000' | 2 byte |
| Byte | 0 | 1 byte |
| Short | 0 | 2 byte |
| Int | 0 | 4 byte |
| Long | 0L | 8 byte |
| Float | 0.0f | 4 byte |
| Double | 0.0d | 8 byte |

**Note->** To get 2's complement of binary number is 1's complement of given number plus 1 to the least significant bit (LSB). For example 2's complement of binary number 10010 is (01101) + 1 = 01110.

## **Reference Datatypes**

* Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
* Class objects and various type of array variables come under reference datatype.
* Default value of any reference variable is null.
* A reference variable can be used to refer any object of the declared type or any compatible type.
* Example: Animal animal = new Animal("giraffe");

## **Java Literals**

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example −

byte a = 68;

char a = 'A';

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example −

int decimal = 100;

int octal = 0144;

int hexa = 0x64;

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are −

### **Example**

"Hello World"

"two\nlines"

"\"This is in quotes\""

String and char types of literals can contain any Unicode characters. For example −

char a = '\u0001';

String a = "\u0001";

Java language supports few special escape sequences for String and char literals as well. They are −

|  |  |
| --- | --- |
| **Notation** | **Character represented** |
| \n | Newline (0x0a) |
| \r | Carriage return (0x0d) |
| \f | Formfeed (0x0c) |
| \b | Backspace (0x08) |
| \s | Space (0x20) |
| \t | Tab |
| \" | Double quote |
| \' | Single quote |
| \\ | Backslash |
| \ddd | Octal character (ddd) |
| \uxxxx | Hexadecimal UNICODE character (xxxx) |

**Note->**for more detail of escape sequence please refer:

https://www.geeksforgeeks.org/escape-sequences-in-java/

# Java - Variables

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. Following is the basic form of a variable declaration −

data type variable [ = value][, variable [ = value] ...] ;

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java −

## **Example**

int a, b, c; // Declares three ints, a, b, and c.

int a = 10, b = 10; // Example of initialization

byte B = 22; // initializes a byte type variable B.

double pi = 3.14159; // declares and assigns a value of PI.

char a = 'a'; // the char variable a iis initialized with value 'a'

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java −

* Local variables
* Instance variables
* Class/Static variables

## **Local Variables**

* Local variables are declared in methods, constructors, or blocks.
* Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
* Access modifiers cannot be used for local variables.
* Local variables are visible only within the declared method, constructor, or block.
* Local variables are implemented at stack level internally.
* There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

### **Example**

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to only this method.

public class Test {

public void pupAge() {

int age = 0;

age = age + 7;

System.out.println("Puppy age is : " + age);

}

public static void main(String args[]) {

Test test = new Test();

test.pupAge();

}

}

This will produce the following result −

### **Output**

Puppy age is: 7

### **Example**

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

public class Test {

public void pupAge() {

int age;

age = age + 7;

System.out.println("Puppy age is : " + age);

}

public static void main(String args[]) {

Test test = new Test();

test.pupAge();

}

}

This will produce the following error while compiling it −

### **Output**

Test.java:4:variable number might not have been initialized

age = age + 7;

^

1 error

## **Instance Variables**

* Instance variables are declared in a class, but outside a method, constructor or any block.
* When a space is allocated for an object in the heap, a slot for each instance variable value is created.
* Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
* Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
* Instance variables can be declared in class level before or after use.
* Access modifiers can be given for instance variables.
* The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
* Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
* Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. *ObjectReference.VariableName*.

### **Example**

import java.io.\*;

public class Employee {

// this instance variable is visible for any child class.

public String name;

// salary variable is visible in Employee class only.

private double salary;

// The name variable is assigned in the constructor.

public Employee (String empName) {

name = empName;

}

// The salary variable is assigned a value.

public void setSalary(double empSal) {

salary = empSal;

}

// This method prints the employee details.

public void printEmp() {

System.out.println("name : " + name );

System.out.println("salary :" + salary);

}

public static void main(String args[]) {

Employee empOne = new Employee("Ransika");

empOne.setSalary(1000);

empOne.printEmp();

}

}

This will produce the following result −

### **Output**

name : Ransika

salary :1000.0

## **Class/Static Variables**

* Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
* There would only be one copy of each class variable per class, regardless of how many objects are created from it.
* Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.
* Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
* Static variables are created when the program starts and destroyed when the program stops.
* Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
* Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
* Static variables can be accessed by calling with the class name *ClassName.VariableName*.
* When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

### **Example**

import java.io.\*;

public class Employee {

// salary variable is a private static variable

private static double salary;

// DEPARTMENT is a constant

public static final String DEPARTMENT = "Development ";

public static void main(String args[])

{

salary = 1000;

System.out.println(DEPARTMENT + "average salary:" + salary);

}

}

This will produce the following result −

### **Output**

Development average salary:1000

**Note** − If the variables are accessed from an outside class, the constant should be accessed as Employee.DEPARTMENT

# What do you mean by a dynamic initialization of variables?

## **If any variable is not assigned with value at compile-time and assigned at run time is called dynamic initialization of a variable**. Basically, this is achieved through constructors, setter methods, normal methods and builtin api methods which returns a value or object.

## **Java Program to variables dynamic initialization**

Let us see the few examples of how to initialize variables at run time.  
 **Creating a class with the only constructor.**

class Rectangle {

private double length;

private double width;

public Rectangle(double length, double width) {

this.length = length;

this.width = width;

System.out.println("Length is:"+length);

System.out.println("Width is:"+width);

}

public static void main(String[] args) {

// passing the values at runtime.

Rectangle rectangle1 = new Rectangle(10,16);

//Rectangle rectangle2 = new Rectangle(5.5, 7.5);

//Rectangle rectangle3 = new Rectangle(2.9, 10);

}

}

in the above program, length and width are initialized at runtime with different values for each Rectangle object.

### **Setter methods based dynamic initialization:**

Let us do now with setter methods. Now add the following to the Rectangle class.

class Rectangle {

double length;

double width;

public double getLength() {

return length;

}

public void setLength(double length) {

this.length = length;

}

public double getWidth() {

return width;

}

public void setWidth(double width) {

this.width = width;

}

public void printdata()

{

System.out.println("Length is:"+length);

System.out.println("Width is:"+width);

}

public static void main(String[] args) {

Rectangle rectangle4 = new Rectangle();

rectangle4.setLength(20);

rectangle4.setWidth(30);

rectangle4.printdata();

}

}

Now, values can be set at runtime using setter methods.

# Scope and lifetime of variables in Java?

### **Instance Variables**

A variable which is declared inside a class and outside all the methods and blocks is an instance variable. The general scope of an instance variable is throughout the class except in static methods. The lifetime of an instance variable is until the object stays in memory.

### **Class Variables**

A variable which is declared inside a class, outside all the blocks and is marked static is known as a class variable. The general scope of a class variable is throughout the class and the lifetime of a class variable is until the end of the program or as long as the class is loaded in memory.

### **Local Variables**

All other variables which are not instance and class variables are treated as local variables including the parameters in a method. Scope of a local variable is within the block in which it is declared and the lifetime of a local variable is until the control leaves the block in which it is declared.

# What is type conversion in java?

Java provides various datatypes to store various data values. It provides 7 primitive datatypes (stores single values) as listed below −

* boolean − Stores 1-bit value representing true or, false.
* byte − Stores twos compliment integer up to 8 bits.
* char − Stores a Unicode character value up to 16 bits.
* short − Stores an integer value upto 16 bits.
* int − Stores an integer value upto 32 bits.
* long − Stores an integer value upto 64 bits.
* float − Stores a floating point value upto 32bits.
* double − Stores a floating point value up to 64 bits.

## **Type Casting/type conversion**

Converting one primitive datatype into another is known as type casting (type conversion) in Java. You can cast the primitive datatypes in two ways namely, Widening and, Narrowing.

**Widening** − Converting a lower datatype to a higher datatype is known as widening. In this case the casting/conversion is done automatically therefore, it is known as implicit type casting. In this case both datatypes should be compatible with each other.



## **Example**

public class WideningExample {

   public static void main(String args[]){

      char ch = 'C';

      int i = ch;

      System.out.println(i);

   }

}

## **Output**

Integer value of the given character: 67

**Narrowing** − Converting a higher datatype to a lower datatype is known as narrowing. In this case the casting/conversion is not done automatically, you need to convert explicitly using the cast operator “( )” explicitly. Therefore, it is known as explicit type casting. In this case both datatypes need not be compatible with each other.



## **Example**

import java.util.Scanner;

public class NarrowingExample {

   public static void main(String args[]){

      Scanner sc = new Scanner(System.in);

      System.out.println("Enter an integer value: ");

      int i = sc.nextInt();

      char ch = (char) i;

      System.out.println("Character value of the given integer: "+ch);

   }

}

## **Output**

Enter an integer value:

67

Character value of the given integer: C

# Java - Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups –

* Unary Operators
* Arithmetic Operators
* Shift Operators
* Relational Operators
* Bitwise Operators
* Logical Operators
* Ternary Operator
* Assignment Operators

### **Java Unary Operator**

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:p 10s

* incrementing/decrementing a value by one
* negating an expression
* inverting the value of a boolean

### **Java Unary Operator Example: ++ and --**

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** x=10;

System.out.println(x++);//10 (11)

System.out.println(++x);//12

System.out.println(x--);//12 (11)

System.out.println(--x);//10

}}

**Output:**

10

12

12

10

### **Java Unary Operator Example 2: ++ and --**

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

**int** a=10;

**int** b=10;

System.out.println(a++ + ++a);//10+12=22

System.out.println(b++ + b++);//10+11=21

}}

**Output:**

22

21

## **The Arithmetic Operators**

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators −

Assume integer variable A holds 10 and variable B holds 20, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator. | A + B will give 30 |
| - (Subtraction) | Subtracts right-hand operand from left-hand operand. | A - B will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator. | A \* B will give 200 |
| / (Division) | Divides left-hand operand by right-hand operand. | B / A will give 2 |
| % (Modulus) | Divides left-hand operand by right-hand operand and returns remainder. | B % A will give 0 |
| ++ (Increment) | Increases the value of operand by 1. | B++ gives 21 |
| -- (Decrement) | Decreases the value of operand by 1. | B-- gives 19 |

## **Example**

public class Test {

public static void main(String args[]) {

int a = 10;

int b = 20;

int c = 25;

int d = 25;

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

System.out.println("a - b = " + (a - b) );

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

System.out.println("b / a = " + (b / a) );

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

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

System.out.println("a++ = " + (a++) );

System.out.println("a-- = " + (a--) );

// Check the difference in d++ and ++d

System.out.println("d++ = " + (d++) );

System.out.println("++d = " + (++d) );

}

}

This will produce the following result −

## **Output**

a + b = 30

a - b = -10

a \* b = 200

b / a = 2

b % a = 0

c % a = 5

a++ = 10

b-- = 11

d++ = 25

++d = 27

### **Java Left Shift Operator**

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

### **Java Left Shift Operator Example**

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

System.out.println(10<<2);//10\*2^2=10\*4=40

System.out.println(10<<3);//10\*2^3=10\*8=80

System.out.println(20<<2);//20\*2^2=20\*4=80

System.out.println(15<<4);//15\*2^4=15\*16=240

}}

**Output:**

40

80

80

240

### **Java Right Shift Operator**

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

### **Java Right Shift Operator Example**

**public** OperatorExample{

**public** **static** **void** main(String args[]){

System.out.println(10>>2);//10/2^2=10/4=2

System.out.println(20>>2);//20/2^2=20/4=5

System.out.println(20>>3);//20/2^3=20/8=2

}}

**Output:**

2

5

2

### **Java Shift Operator Example: >> vs >>>**

**public** **class** OperatorExample{

**public** **static** **void** main(String args[]){

  //For positive number, >> and >>> works same

    System.out.println(20>>2);

    System.out.println(20>>>2);

    //For negative number, >>> changes parity bit (MSB) to 0

    System.out.println(-20>>2);

    System.out.println(-20>>>2);

}}

**Output:**

5

5

-5

1073741819

## **The Relational Operators**

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == (equal to) | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != (not equal to) | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > (greater than) | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < (less than) | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= (greater than or equal to) | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= (less than or equal to) | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. |  |

## **Example**

public class Test {

public static void main(String args[]) {

int a = 10;

int b = 20;

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

System.out.println("a != b = " + (a != b) );

System.out.println("a > b = " + (a > b) );

System.out.println("a < b = " + (a < b) );

System.out.println("b >= a = " + (b >= a) );

System.out.println("b <= a = " + (b <= a) );

}

}

This will produce the following result −

## **Output**

a == b = false

a != b = true

a > b = false

a < b = true

b >= a = true

b <= a = false

## **The Bitwise Operators**

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011

The following table lists the bitwise operators −

Assume integer variable A holds 60 and variable B holds 13 then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & (bitwise and) | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| | (bitwise or) | Binary OR Operator copies a bit if it exists in either operand. | (A | B) will give 61 which is 0011 1101 |
| ^ (bitwise XOR) | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ (bitwise compliment) | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << (left shift) | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> (right shift) | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 1111 |
| >>> (zero fill right shift) | Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. | A >>>2 will give 15 which is 0000 1111 |

## **Example**

public class Test {

public static void main(String args[]) {

int a = 60; /\* 60 = 0011 1100 \*/

int b = 13; /\* 13 = 0000 1101 \*/

int c = 0;

c = a & b; /\* 12 = 0000 1100 \*/

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

c = a | b; /\* 61 = 0011 1101 \*/

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

c = a ^ b; /\* 49 = 0011 0001 \*/

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

c = ~a; /\*-61 = 1100 0011 \*/

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

c = a << 2; /\* 240 = 1111 0000 \*/

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

c = a >> 2; /\* 15 = 1111 \*/

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

c = a >>> 2; /\* 15 = 0000 1111 \*/

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

}

}

This will produce the following result −

## **Output**

a & b = 12

a | b = 61

a ^ b = 49

~a = -61

a << 2 = 240

a >> 2 = 15

a >>> 2 = 15

## **The Logical Operators**

The following table lists the logical operators −

Assume Boolean variables A holds true and variable B holds false, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && (logical and) | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false |
| || (logical or) | Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. | (A || B) is true |
| ! (logical not) | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true |

## **Example**

public class Test {

public static void main(String args[]) {

boolean a = true;

boolean b = false;

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

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

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

}

}

This will produce the following result −

## **Output**

a && b = false

a || b = true

!(a && b) = true

### **Ternary Operator ( ? : )**

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as −

variable x = (expression) ? value if true : value if false

Following is an example −

**Example**

public class Test {

public static void main(String args[]) {

int a, b;

a = 10;

b = (a == 1) ? 20: 30;

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

b = (a == 10) ? 20: 30;

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

}

}

This will produce the following result −

**Output**

Value of b is : 30

Value of b is : 20

### **instanceof Operator**

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as −

( Object reference variable ) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example −

**Example**

public class Test {

public static void main(String args[]) {

String name = "James";

// following will return true since name is type of String

boolean result = name instanceof String;

System.out.println( result );

}

}

This will produce the following result −

**Output**

true

## **The Assignment Operators**

Following are the assignment operators supported by Java language −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator. Assigns values from right side operands to left side operand. | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand. | C -= A is equivalent to C = C – A |
| \*= | Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand. | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand. | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand. | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator. | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator. | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator. | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator. | C ^= 2 is same as C = C ^ 2 |
| |= | bitwise inclusive OR and assignment operator. | C |= 2 is same as C = C | 2 |

## **Example**

public class Test {

public static void main(String args[]) {

int a = 10;

int b = 20;

int c = 0;

c = a + b;

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

c += a ;

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

c -= a ;

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

c \*= a ;

System.out.println("c \*= a = " + c );

a = 10;

c = 15;

c /= a ;

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

a = 10;

c = 15;

c %= a ;

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

c <<= 2 ;

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

c >>= 2 ;

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

c >>= 2 ;

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

c &= a ;

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

c ^= a ;

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

c |= a ;

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

}

}

This will produce the following result −

## **Output**

c = a + b = 30

c += a = 40

c -= a = 30

c \*= a = 300

c /= a = 1

c %= a = 5

c <<= 2 = 20

c >>= 2 = 5

c >>= 2 = 1

c &= a = 0

c ^= a = 10

c |= a = 10