Session 02:

JAVA Variables, Data Types, and Operators

# Basic Datatypes

# Variable Types

# Basic Operators

# Basic 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**

* Example: byte a = 100, byte b = -50

### **short**

* Example: short s = 10000, short r = -20000

### **int**

* Example: int a = 100000, int b = -200000

### **long**

* Example: long a = 100000L, long b = -200000L

### **float**

* Example: float f1 = 234.5f

### **double**

* Example: double d1 = 123.4

### **boolean**

* Example: boolean one = true

### **char**

* Example: char letterA = 'A'

## 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) |

# Variable Types

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

# Basic Operators

## 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 −

[Show Examples](https://www.tutorialspoint.com/java/java_arithmatic_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

## The Relational Operators

There are following relational operators supported by Java language.

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

[Show Examples](https://www.tutorialspoint.com/java/java_relational_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **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. | (A <= B) is true. |

## 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 −

[Show Examples](https://www.tutorialspoint.com/java/java_bitwise_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

## The Logical Operators

The following table lists the logical operators −

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

[Show Examples](https://www.tutorialspoint.com/java/java_logical_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

## The Assignment Operators

Following are the assignment operators supported by Java language −

[Show Examples](https://www.tutorialspoint.com/java/java_assignment_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **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 |

## Miscellaneous Operators

There are few other operators supported by Java Language.

### **Conditional 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

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example −

**Example**

class Vehicle {}

public class Car extends Vehicle {

public static void main(String args[]) {

Vehicle a = new Car();

boolean result = a instanceof Car;

System.out.println( result );

}

}

This will produce the following result −

**Output**

true

## Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example, x = 7 + 3 \* 2; here x is assigned 13, not 20 because operator \* has higher precedence than +, so it first gets multiplied with 3 \* 2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | expression++ expression-- | Left to right |
| Unary | ++expression –-expression +expression –expression ~ ! | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> >>> | Left to right |
| Relational | < > <= >= instanceof | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %= ^= |= <<= >>= >>>= | Right to left |