**Basic terminologies in Java**

When we consider a Java program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what class, object, methods, and instance variables mean.

* **Class:** The class is a blueprint (plan) of the instance of a class (object). It can be defined as a logical template that shares common properties and methods.

Example 1: Blueprint of the house is class.

Example 2: In the real world, Alice is an object of the “Human” class.

* **Object:** Objects have states and behaviors. Example: A dog has states - color, name, and breed as well as behavior such as wagging their tail, barking, and eating. An object is an instance of a class.
* **Methods:** A method is a behavior. A class can contain many methods. It is in methods where the logic are written, data is manipulated and all the actions are executed.
* **Instance Variables :** Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.
* **public static void main(String [] args):** The method main() is the main entry point into a Java program; this is where the processing starts. Also allowed is the signature

**public static void main(String… args){ ..}**

Public Class class\_name {

public static void main(String[] args)

{

System.out.println("welcome to GUVI");

}

}

**Output -**

welcome to GUVI

* **Comments in Java**

There are three types of comments in Java.

* **Single line Comment**

**//** System.out.println("This is an comment.");

* **Multiline Comment**

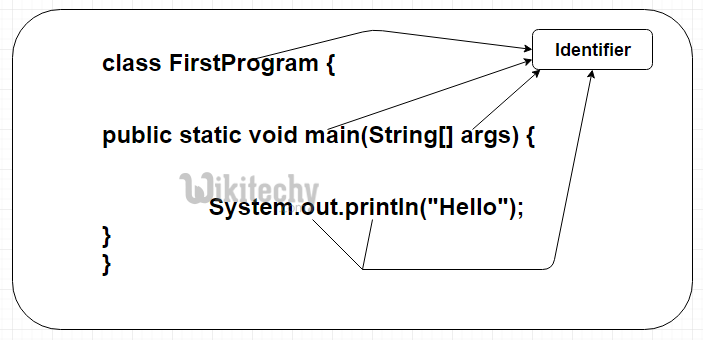
**/\***

System.out.println("This is the first line comment.");

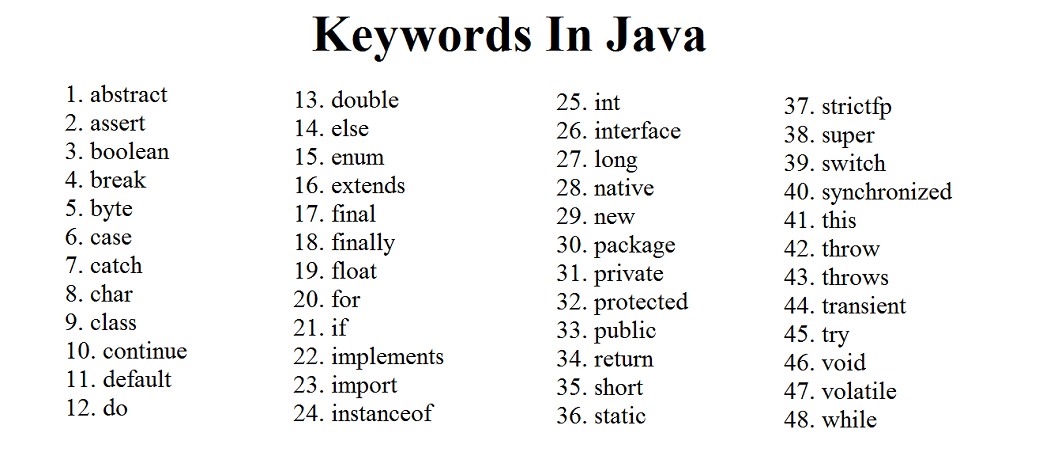
System.out.println("This is the second line comment.");

**\*/**

* **Identifiers in java :**

**** All Java components require names. Names used for classes, variables, and methods are called identifiers.

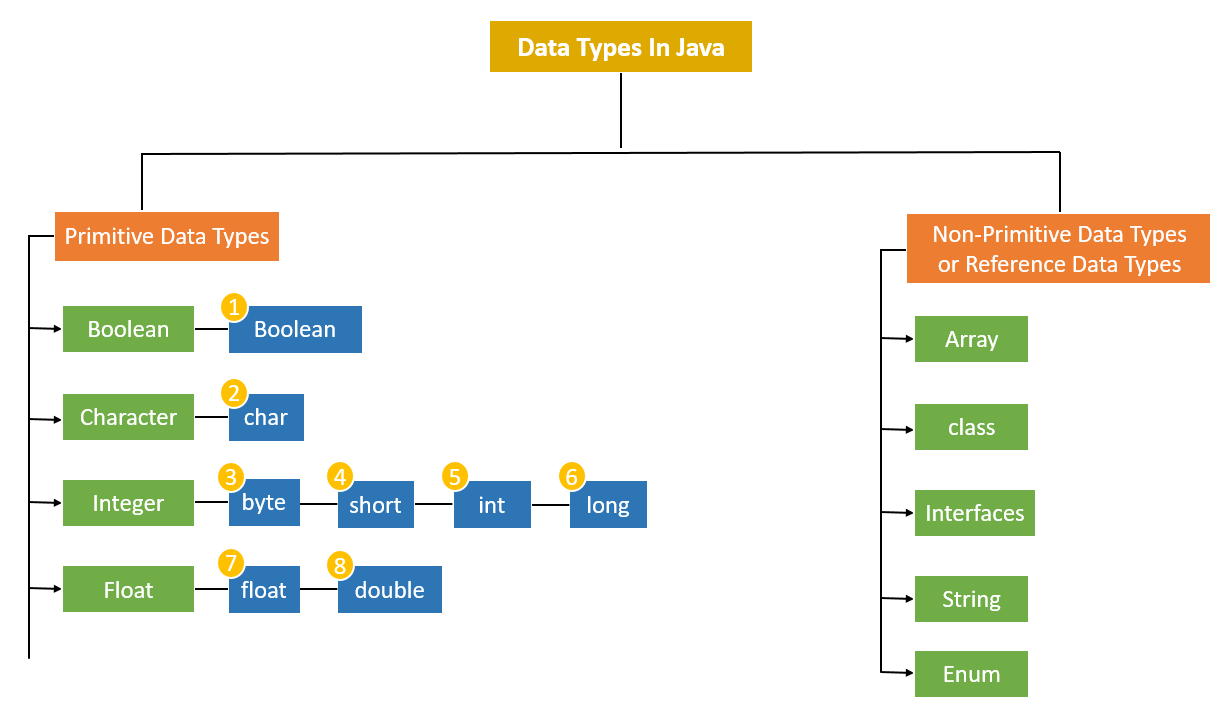
* **Rules to write** **Identifiers :**
* All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (\_).
* After the first character, identifiers can have any combination of characters.
* A key word cannot be used as an identifier.
* We can’t use space in between an identifier.
* Most importantly, identifiers are case sensitive.
* Examples of legal identifiers: age, $salary, \_value, \_\_1\_value.
* Examples of illegal identifiers: 123abc, -salary.
* **Keywords in java :**

Keywords are predefined, reserved words used in Java programming that have special meanings to the compiler. The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

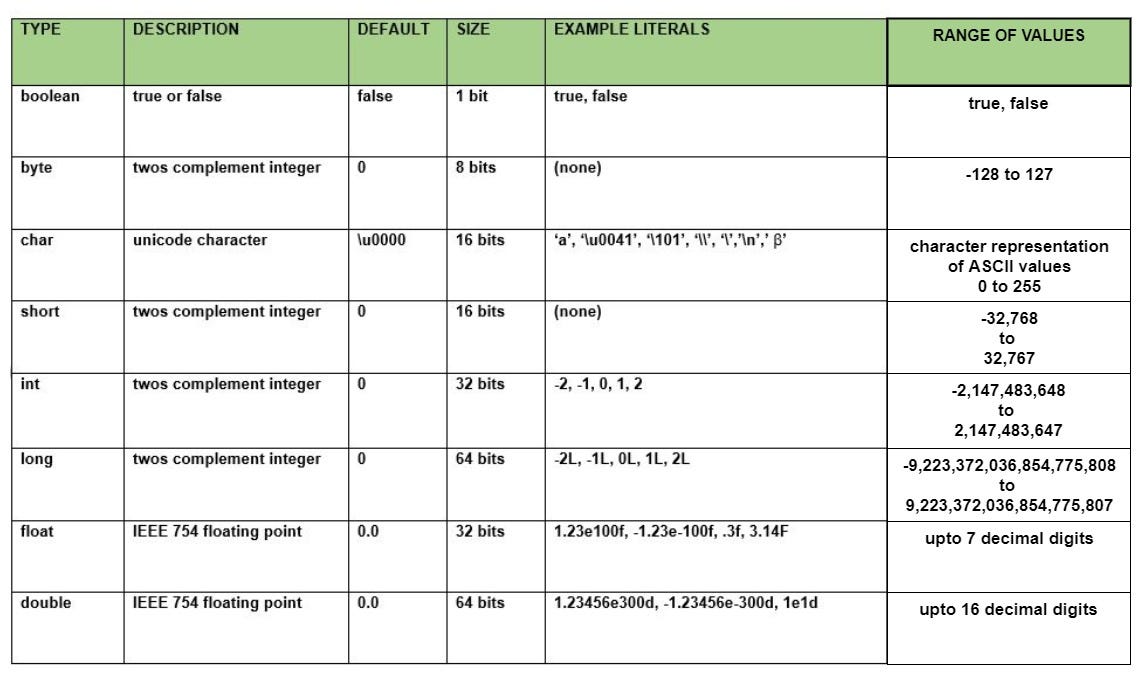
* **What are Data Types in Java?**

Data types in Java are of different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. Java has two categories in which data types are segregated

1. **Primitive Data Type:** such as Boolean, char, int, short, byte, long, float, and double
2. **Non-Primitive Data Type or Object Data type:** such as String, Array, etc.



* **Primitive Data Types in Java**

Primitive data are only single values and have no special capabilities. There are 8 primitive data types. They are depicted below in tabular format below as follows:

Let us discuss and implement each one of the following data types that are as follows:

1. **Boolean Data Type**

Boolean data type represents only one bit of information either **true or false** which is intended to represent the two truth values of logic and Boolean algebra, but the size of the Boolean data type is virtual machine-dependent. Values of type Boolean are not converted implicitly or explicitly (with casts) to any other type. But the programmer can easily write conversion code.

**Syntax**:

**boolean** booleanVar;

**Size**: Virtual machine dependent.

**2. Byte Data Type**

The byte data type is an 8-bit signed two’s complement integer. The byte data type is useful for saving memory in large arrays.

**Syntax:**

**byte** byteVar;

**Size:** 1 byte (8 bits)

**3. Short Data Type**

The short data type is a 16-bit signed two’s complement integer. Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.

**Syntax:**

**short** shortVar;

**Size:** 2 bytes (16 bits)

**4. Integer Data Type**

It is a 32-bit signed two’s complement integer.

**Syntax:**

**int** intVar;

**Size:** 4 bytes ( 32 bits )

**5. Long Data Type**

The range of a long is quite large. The long data type is a 64-bit two’s complement integer and is useful for those occasions where an int type is not large enough to hold the desired value. The size of the Long Datatype is 8 bytes (64 bits).

**Syntax:**

**long** longVar;

**6. Float Data Type**

The float data type is a single-precision 32-bit IEEE 754 floating-point. Use a float (instead of double) if you need to save memory in large arrays of floating-point numbers. The size of the float data type is 4 bytes (32 bits).

**Syntax:**

**float** floatVar;

**7. Double Data Type**

The double data type is a double-precision 64-bit IEEE 754 floating-point. For decimal values, this data type is generally the default choice. The size of the double data type is 8 bytes or 64 bits.

**Syntax:**

**double** doubleVar;

**8. Char Data Type**

The char data type is a single 16-bit Unicode character with the size of 2 bytes (16 bits).

**Syntax:**

**char** charVar;

**Non-Primitive Data Type or Reference Data Types :**

The Reference Data Types will contain a memory address of variable values because the reference types won’t store the variable value directly in memory. They are strings, objects, arrays, etc.

**1. STRINGS**

Strings are defined as an array of characters. The difference between a character array and a string in Java is, that the string is designed to hold a sequence of characters in a single variable whereas, a character array is a collection of separate char-type entities. Unlike C/C++, Java strings are not terminated with a null character.

**Syntax:** Declaring a string

<String\_Type> <string\_variable> = “<sequence\_of\_string>”;

**Example:**

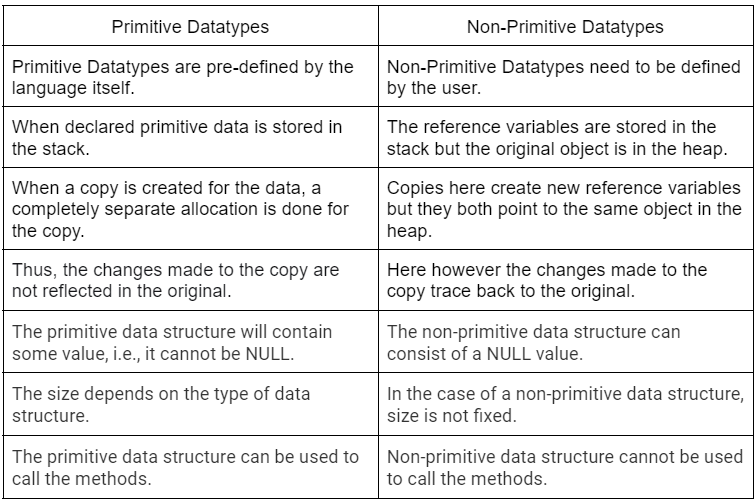
// Declare String without using new operator

String s = "welcome to Guvi";

// Declare String using new operator

String s1 = new String("welcome to guvi");

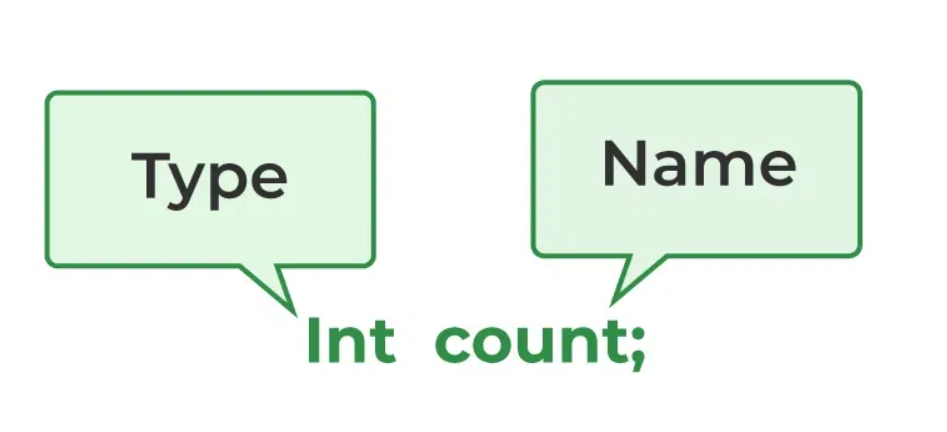
* **Primitive data type vs. non-primitive type in Java**

****

* **Variables in Java**

Java variable is a name given to a memory location. It is the basic unit of storage in a program.

* The value stored in a variable can be changed during program execution.
* Variables in Java are only a name given to a memory location. All the operations done on the variable affect that memory location.
* In Java, all variables must be declared before use.
* **How to Declare Variables in Java?**

We can declare variables in Java as pictorially depicted below as a visual aid.

From the image, it can be easily perceived that while declaring a variable, we need to take care of two things that are:

**datatype:** Type of data that can be stored in this variable.

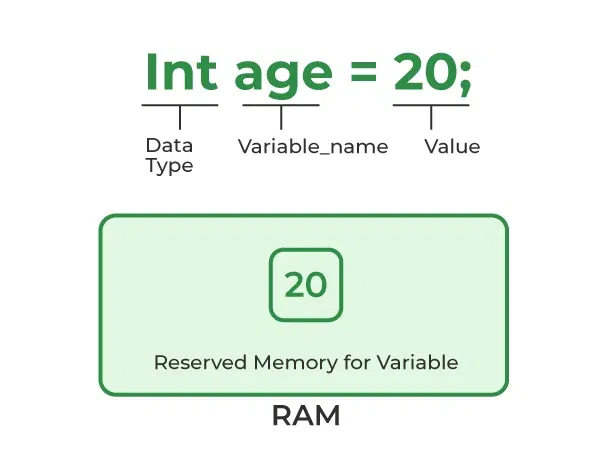
**data\_name**: Name was given to the variable.

* **How to Initialize Variables in Java?**

It can be perceived with the help of 3 components that are as follows:

**datatype:** Type of data that can be stored in this variable.

**variable\_name:** Name given to the variable.

**value:** It is the initial value stored in the variable.

* **Types of Variables in Java**

Now let us discuss different types of variables which are listed as follows:

* Local Variables
* Instance Variables
* Static Variables

**1. Local Variables**

A variable defined within a block or method or constructor is called a local variable.

* These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function.
* The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.
* Initialization of the local variable is mandatory before using it in the defined scope.

Below is the implementation of the above approach:

Class guvi {

public static void main(String[] args)

{

// Declared a Local Variable

**int var = 10;**

// This variable is local to this main method only

System.out.println("Local Variable: " + **var**);

}

}

**2. Instance Variables**

Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.

* As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
* Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
* Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable.
* Instance variables can be accessed only by creating objects.
* We initialize instance variables using constructors while creating an object. We can also use instance blocks to initialize the instance variables.

// Java Program to demonstrate

class guvi {

// Declared instance variable

String student = "Shubham Jain";

public static void main(String[] args)

{

Guvi Zen=new Guvi();

// geek variable can be accessed by creating object

System.out.print(Zen. student); }

}

**3. Static Variables**

Static variables are also known as class variables.

* These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor, or block.
* Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
* Static variables are created at the start of program execution and destroyed automatically when execution ends.
* Initialization of a static variable is not mandatory. Its default value is dependent on the data type of variable

// Java Program to demonstrate

class guvi {

// Declared instance variable

Public static String student = "Shubham Jain";

public static void main(String[] args)

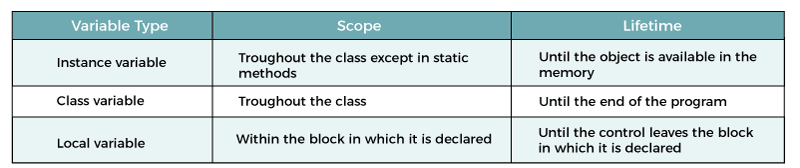
{

// student variable can be accessed without object

// static variable

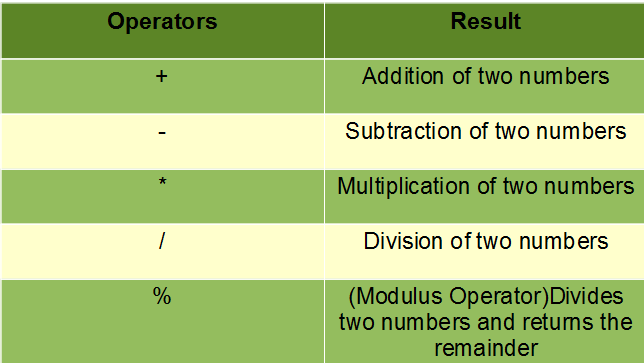
System.out.print(guvi. student); }

}

* **Scope of variables**
* **Operators In java :**

Operators in Java are the symbols used for performing specific operations in Java. Operators make tasks like addition, multiplication, etc. which look easy although the implementation of these tasks is quite complex.

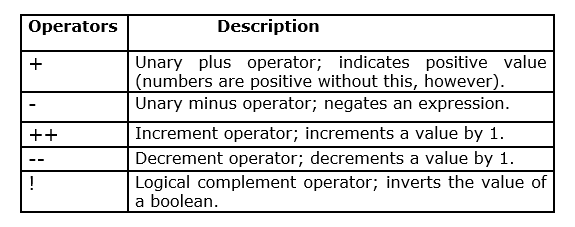
**Types of Operators in Java :**

1. **Arithmetic Operators**These operators involve the mathematical operators that can be used to perform

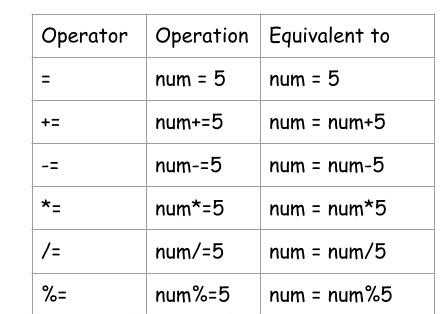
various simple or advanced arithmetic operations on the primitive data types referred to as the operands. These operators consist of various unary and binary operators that can be applied on a single or two operands.

1. **Unary Operators in Java**

Java unary operators are the types that need only one operand to perform any operation like increment, decrement, negation, etc. It consists of various arithmetic, logical, and other operators that operate on a single operand. Let’s look at the various unary operators in detail and see how they operate.

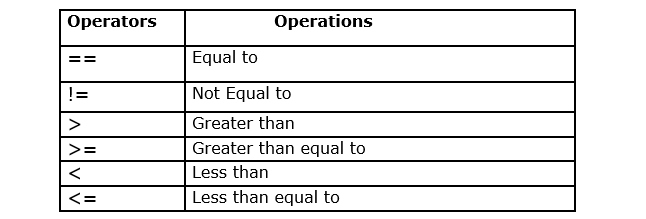


1. **Assignment Operators**

These operators are used to assign values to a variable. The left side operand of the assignment operator is a variable, and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type of the operand on the left side. Otherwise, the compiler will raise an error. This means that the assignment operators have right to left associativity.

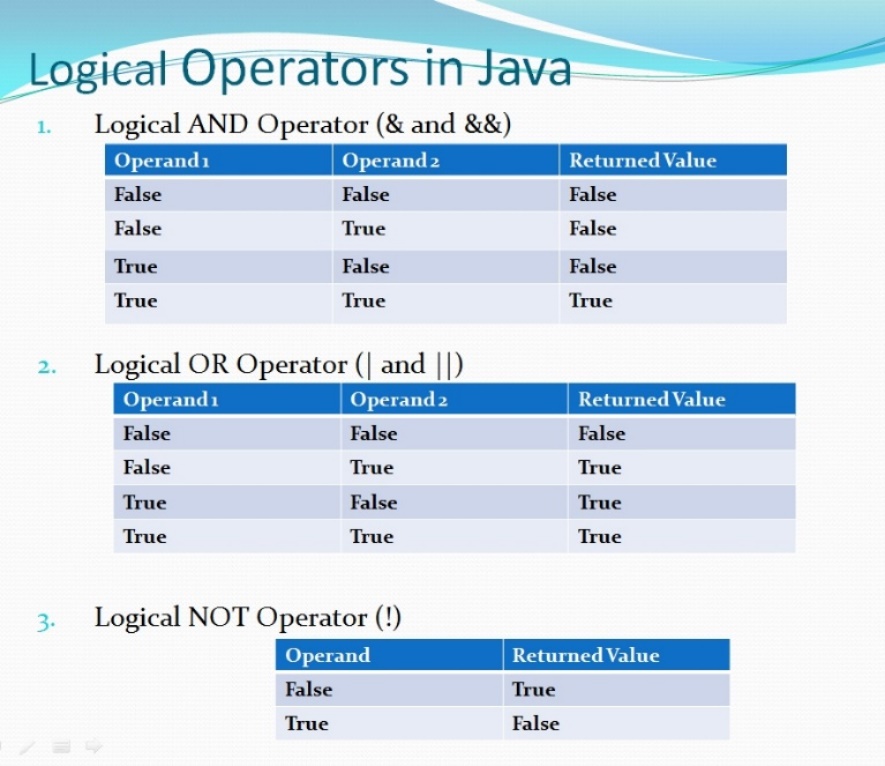
1. **Relational Operators**

Java Relational Operators are a bunch of binary operators used to check for relations between two operands, including equality, greater than, less than, etc. They return a boolean result after the comparison and are extensively used in looping statements as well as conditional if-else statements and so on.



1. **Logical Operators**

Logical operators are used to performing logical “AND”, “OR” and “NOT” operations, i.e. the function similar to AND gate and OR gate in digital electronics. They are used to combine two or more conditions/constraints or to complement the evaluation of the original condition under particular consideration. One thing to keep in mind is, while using AND operator, the second condition is not evaluated if the first one is false. Whereas while using the OR operator, the second condition is not evaluated if the first one is true

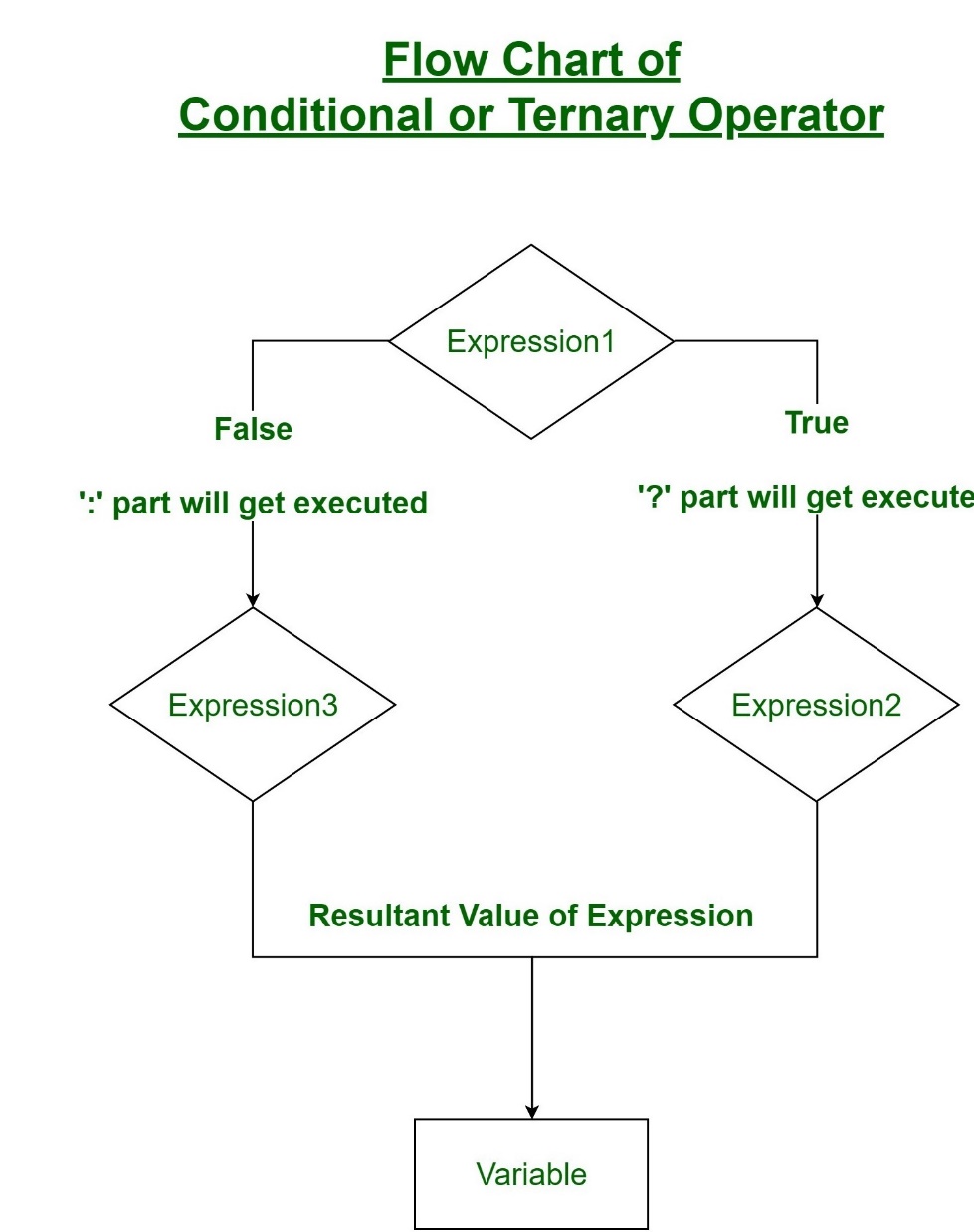


1. **Ternary Operator**

Java ternary operator is the only conditional operator that takes three operands. It’s a one-liner replacement for the if-then-else statement and is used a lot in Java programming. We can use the ternary operator in place of if-else conditions or even switch conditions using nested ternary operators.

**Syntax:**

**variable = Expression1 ? Expression2 : Expression3**



* **Type Casting in Java :**

In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion is performed by the programmer. In this section, we will discuss type casting and its types with proper examples.

* **Type casting**

Converting a value from one data type to another data type is known as type casting.

**Types of Type Casting**

There are two types of type casting:

1. Widening Type Casting
2. Narrowing Type Casting

* **Widening Type Casting**

Converting a lower data type into a higher one is called widening type casting. It is also known as implicit conversion or casting down. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

* Both data types must be compatible with each other.
* The target type must be larger than the source type.

**byte -> short -> char -> int -> long -> float -> double**

For example, the conversion between numeric data types to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

**public** **class** WideningTypeCastingExample

{

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

{

**int** x = 7;

//automatically converts the integer type into long type

**long** y = x;

//automatically converts the long type into float type

**float** z = y;

System.out.println("Before conversion, int value "+x);

System.out.println("After conversion, long value "+y);

System.out.println("After conversion, float value "+z);

}

}

**Output:**

Before conversion, the value is: 7

After conversion, the long value is: 7

After conversion, the float value is: 7.0

* **Narrowing Type Casting**

Converting a higher data type into a lower one is called narrowing type casting. It is also known as explicit conversion or casting up. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

**double -> float -> long -> int -> char -> short -> byte**

Let's see an example of narrowing type casting.

In the following example, we performed the narrowing type casting two times. First, we have converted the double type into a long data type after that long data type is converted into an int type.

**public** **class** NarrowingTypeCastingExample

{

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

{

**double** d = 166.66;

//converting double data type into long data type

**long** l = (**long**)d;

//converting long data type into int data type

**int** i = (**int**)l;

System.out.println("Before conversion: "+d);

//fractional part lost

System.out.println("After conversion into long type: "+l);

//fractional part lost

System.out.println("After conversion into int type: "+i);

}

}

**Output:**

Before conversion: 166.66

After conversion into long type: 166

After conversion into int type: 166