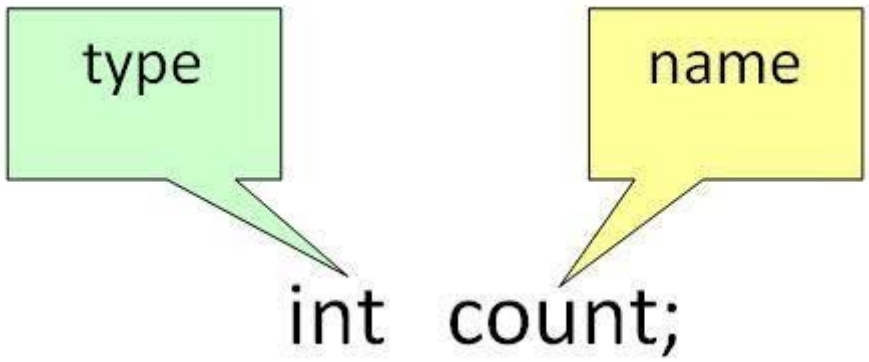
How to declare variables?

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:

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

2. data_name: Name given to the variable.

In this way, a name can only be given to a memory location. It can be assigned values in two ways:

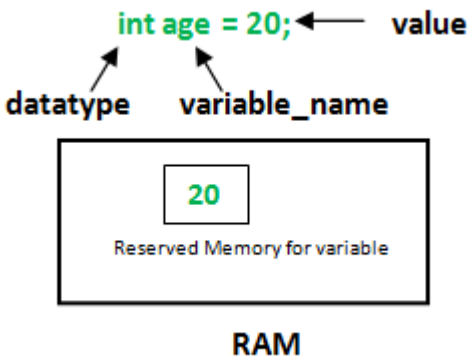
- Variable Initialization
- Assigning value by taking input

How to initialize variables?

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.

Illustrations:

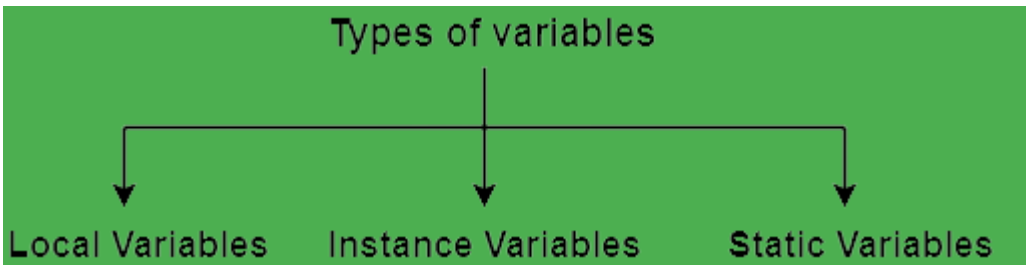



```
float simpleInterest;
// Declaring float variable

int time = 10, speed = 20;
// Declaring and initializing integer variable

char var = 'h';
// Declaring and initializing character variable
```

Types of Variables in Java

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

1. Local Variables
2. Instance Variables
3. 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.

```
/*package whatever //do not write package name here */
// Contributed by Shubham Jain
import java.io.*;

class GFG {
public static void main(String[] args)
{
int var = 10; // Declared a Local Variable
// This variable is local to this main method only
System.out.println("Local Variable: " + var);
}
}
```

Output

```
Local Variable: 10
```

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 0.
- Instance variables can be accessed only by creating objects.

```
/*package whatever //do not write package name here */

import java.io.*;

class GFG {

	public String geek; // Declared Instance Variable

	public GFG()
	{ // Default Constructor

		this.geek = "Ali ahmed"; // initializing Instance Variable
	}
//Main Method
	public static void main(String[] args)
	{

		// Object Creation
		GFG name = new GFG();
		// Displaying O/P
		System.out.println("his  name is: " + name.geek);
	}
}
```

Output

```
his name is: Ali Ahmed
```

3. Static Variables

Static variables are also known as class variables.

- These variables are declared similarly as 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 0.
- If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won't halt the program. The compiler will replace the object name with the class name automatically.
- If we access a static variable without the class name, the compiler will automatically append the class name.

```
/*package whatever //do not write package name here */
```

```
import java.io.*;

class GFG {

public static String geek = "Ali Ahmed";                        //Declared static variable

        public static void main (String[] args) {

        //geek variable can be accessed without object creation
        //Displaying O/P
        //GFG.geek --> using the static variable
                System.out.println("his Name is : "+GFG.geek);
        }
}
```

Output

```
his Name is : Ali Ahmed
```

Differences between the Instance variables and the Static variables

- Each object will have its own copy of an instance variable, whereas we can only have one copy of a static variable per class, irrespective of how many objects we create.
- Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of the instance variable. In the case of a static variable, changes will be reflected in other objects as static variables are common to all objects of a class.
- We can access instance variables through object references, and static variables can be accessed directly using the class name**.**

Syntax: Static and instance variables

```
class GFG
{
    // Static variable
    static int a;

    // Instance variable
    int b;
```

```
}
```

Scope of Variables In Java

Scope of a variable is the part of the program where the variable is accessible. Like C/C++, in Java, all identifiers are lexically (or statically) scoped, i.e.scope of a variable can determined at compile time and independent of function call stack.
Java programs are organized in the form of classes. Every class is part of some package. Java scope rules can be covered under following categories.

Member Variables (Class Level Scope)

These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class. Let's take a look at an example:

```
public class Test
{
   // All variables defined directly inside a class
   // are member variables
   int a;
   private String b;
   void method1() {....}
   int method2() {....}
   char c;
}
```

- We can declare class variables anywhere in class, but outside methods.
- Access specified of member variables doesn't affect scope of them within a class.
- Member variables can be accessed outside a class with following rules

Local Variables (Method Level Scope)

Variables declared inside a method have method level scope and can't be accessed outside the method.

```
public class Test
{
   void method1()
   {
     // Local variable (Method level scope)
     int x;
   }
}
```

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
```

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
```

Note : Local variables don’t exist after method’s execution is over.

Here’s another example of method scope, except this time the variable got passed in as a parameter to the method:

```
class Test
{
   private int x;
   public void setX(int x)
   {
      this.x = x;
   }
}
```

The above code uses **this** keyword to differentiate between the local and class variables.

As an exercise, predict the output of following Java program.

Loop Variables (Block Scope)
A variable declared inside pair of brackets "{" and "}" in a method has scope within the brackets only.

```
public class Test

{
	public static void main(String args[])
	{
		{
			// The variable x has scope within
			// brackets
			int x = 10;
			System.out.println(x);
		}

		// Uncommenting below line would produce
		// error since variable x is out of scope.

		// System.out.println(x);
	}
}
```

Output:

```
10
```

As another example, consider following program with a for loop.

```
class Test
{
	public static void main(String args[])
	{
		for (int x = 0; x < 4; x++)
		{
			System.out.println(x);
		}

		// Will produce error
		System.out.println(x);
	}
}
```

Output:

```
11: error: cannot find symbol
        System.out.println(x);
```

The right way of doing above is,

```
// Above program after correcting the error

class Test
{
	public static void main(String args[])
	{
		int x;
		for (x = 0; x < 4; x++)
		{
			System.out.println(x);
		}

	System.out.println(x);
	}
}
```

Output:

```
0
1
2
3
4
```

Let’s look at tricky example of loop scope. Predict the output of following program. You may be surprised if you are regular C/C++ programmer.

Output :

```
6: error: variable a is already defined in method go(int)
       for (int a = 0; a < 5; a++)
                 ^
1 error
```

As an exercise, predict the output of the following Java program.

```
class Test
{
        public static void main(String args[])
        {
                {
                        int x = 5;
                        {
                                int x = 10;
                                System.out.println(x);
                        }
                }
        }
}
```

From the above knowledge, tell whether the below code will run or not.

```
class Test {

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

```
	{
		for (int i = 1; i <= 10; i++) {
			System.out.println(i);
		}
		int i = 20;
		System.out.println(i);
	}
}
```

Output :

```
1
2
3
4
5
6
7
8
9
10
20
```

Yes, it will run!
See the program carefully, inner loop will terminate before the outer loop variable is declared.So the inner loop variable is destroyed first and then the new variable of same name has been created.

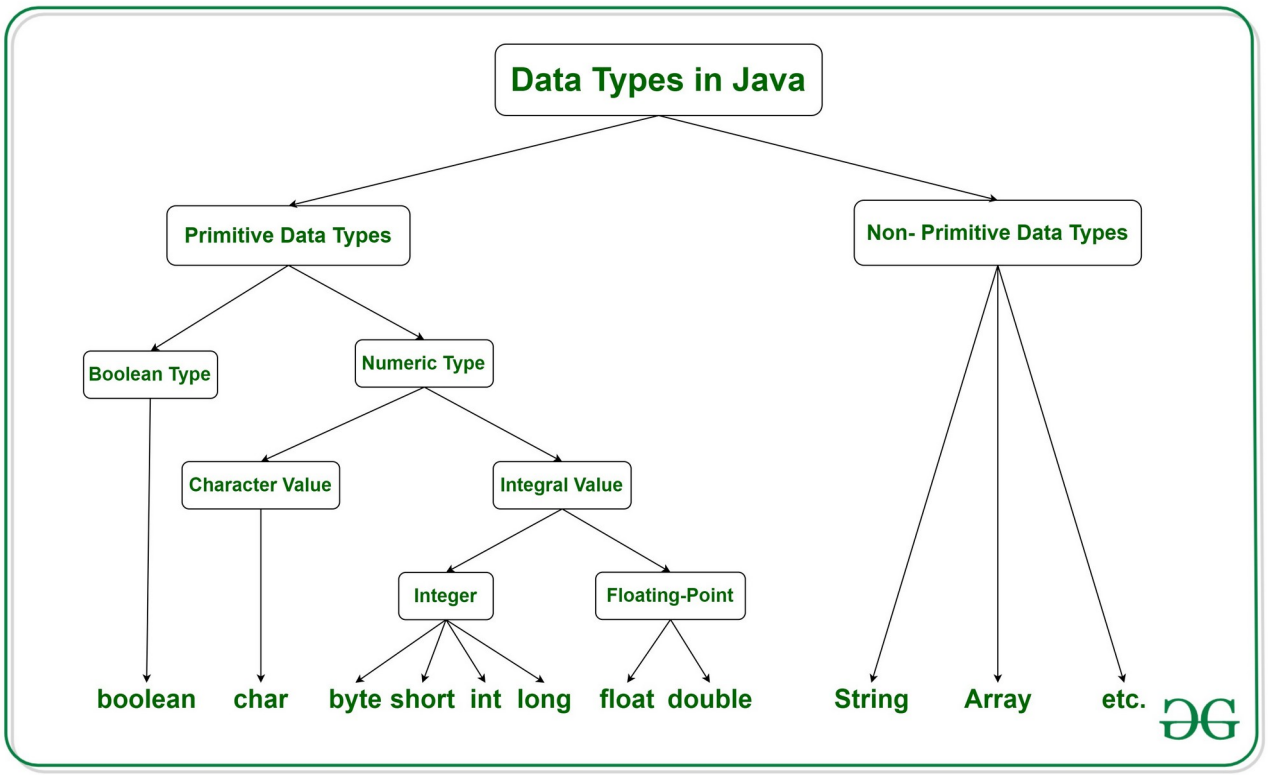
Data types in Java

Data types are 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. Also, let us cover up other important ailments that there are majorly two types of languages that are as follows:

1. First, one is a **Statically typed language** where each variable and expression type is already known at compile time. Once a variable is declared to be of a certain data type, it cannot hold values of other data types.For example C, C++, Java.
2. The other is **Dynamically typed languages.** These languages can receive different data types over time. For example Ruby, Python

Java is **statically typed and also a strongly typed language** because, in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth)

is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types.



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.

Types Of Primitive Data Types

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:

TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE LITERALS	RANGE OF VALUES
boolean	true or false	false	1 bit	true, false	true, false
byte	twos complement integer	0	8 bits	(none)	-128 to 127
char	unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\'','\n',' β'	character representation of ASCII values 0 to 255
short	twos complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos complement integer	0	32 bits	-2, -1, 0, 1, 2	-2,147,483,648 to 2,147,483,647
long	twos complement integer	0	64 bits	-2L, -1L, 0L, 1L, 2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f, -1.23e-100f, .3f, 3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d, -1.23456e-300d, 1e1d	upto 16 decimal digits

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

Type 1: boolean

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

Values: Boolean such as true, false

Default Value: false

Example:

```
// Java Program to Demonstrate Boolean Primitive DataType

// Class

class GFG {

	// Main driver method
	public static void main(String args[])
	{

		//Boolean data type is a data type that has one of two
	//possible values (usually denoted true and false).
		// Setting boolean to false and true initially
		boolean a = false;
		boolean b = true;

		// If condition holds
		if (b == true){

			// Print statement
			System.out.println("Hi Geek");
		}
		// If condition holds
		if(a == false){

		// Print statement
			System.out.println("Hello Geek");
		}
	}
}
```

Output

```
Hi Geek
Hello Geek
```

Type 2: byte

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)

Values: -128 to 127

Default Value: 0

Example:

```
// Java Program to demonstrate Byte Data Type


// Class
class GFG {

	// Main driver method
	public static void main(String args[]) {

		byte a = 126;

		// byte is 8 bit value
		System.out.println(a);

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

		// It overflows here because
		// byte can hold values from -128 to 127
		a++;
		System.out.println(a);

		// Looping back within the range
		a++;
		System.out.println(a);
	}
}
```

Output

```
126
127
-128
-127
```

Type 3: short

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 byte (16 bits)

Values: -32, 768 to 32, 767 (inclusive)

Default Value: 0

Type 4: int

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

Syntax:

```
int intVar;
```

Size: 4 byte (32 bits)

Values: -2, 147, 483, 648 to 2, 147, 483, 647 (inclusive)

> **Note:** The default value is '0'
>
> **Remember:** In Java SE 8 and later, we can use the int data type to represent an unsigned 32-bit integer, which has a value in the range [0, 2^{32}-1]. Use the Integer class to use the int data type as an unsigned integer.

Type 5: long

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.

Syntax:

```
long longVar;
```

Size: 8 byte (64 bits)

Values: {-9, 223, 372, 036, 854, 775, 808} to {9, 223, 372, 036, 854, 775, 807} (inclusive)

Note: The default value is '0'.

Type 6: float

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.

Syntax:

```
float floatVar;
```

Size: 4 byte (32 bits)

Values: upto 7 decimal digits

Note: The default value is '0.0'.

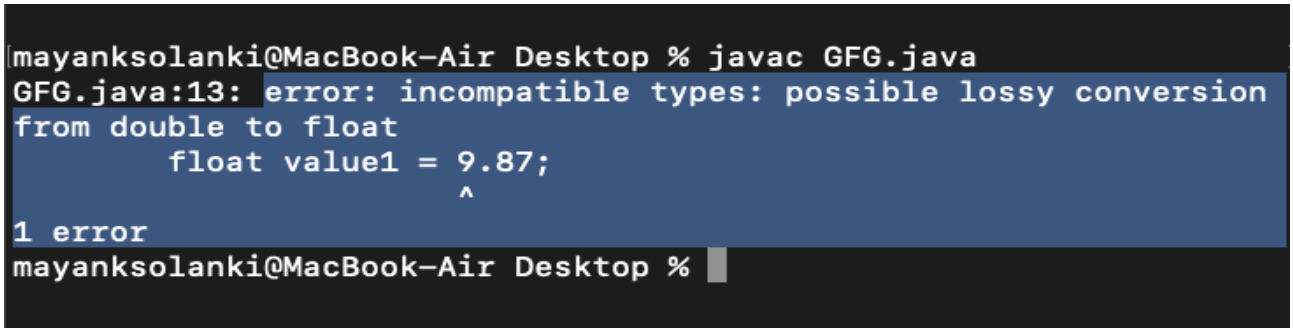
Examples

```
//Java Program to Illustrate Float Primitive Data Type


// Importing required classes
import java.io.*;

// Class
class GFG {

	// Main driver method
	public static void main(String[] args)
	{
		// Declaring and initializing float value

		// float value1 = 9.87;
		// Print statement
		// System.out.println(value1);
		float value2 = 9.87f;
		System.out.println(value2);
	}
}
```

Output

```
9.87
```

If we uncomment lines no 14,15,16 then the output would have been totally different as we would have faced an error.



Type 7: double

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.

Syntax:

```
double doubleVar;
```

Size: 8 bytes or 64 bits

Values: Upto 16 decimal digits

Example:

```
// Java Program to Demonstrate Char Primitive Data Type


// Class
class GFG {

	// Main driver method
	public static void main(String args[])
	{

		// Creating and initializing custom character
		char a = 'G';

		// Integer data type is generally
		// used for numeric values
		int i = 89;

		// use byte and short
		// if memory is a constraint
		byte b = 4;

		// this will give error as number is
		// larger than byte range
		// byte b1 = 7888888955;

		short s = 56;

		// this will give error as number is
		// larger than short range
		// short s1 = 87878787878;

		// by default fraction value
		// is double in java
		double d = 4.355453532;

		// for float use 'f' as suffix as standard
		float f = 4.7333434f;

		//need to hold big range of numbers then we need this data type
		long l = 12121;

		System.out.println("char: " + a);
```

```
                System.out.println("integer: " + i);
                System.out.println("byte: " + b);
                System.out.println("short: " + s);
                System.out.println("float: " + f);
                System.out.println("double: " + d);
                System.out.println("long: " + l);
        }
}
```

Output

```
char: G
integer: 89
byte: 4
short: 56
float: 4.7333436
double: 4.355453532
long: 12121
```

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.

A: 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.

Syntax: Declaring a string

```
<String_Type> <string_variable> = "<sequence_of_string>";
```

Example:

```
// Declare String without using new operator
String s = "GeeksforGeeks";

// Declare String using new operator
String s1 = new String("GeeksforGeeks");
```

B: Class

A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default access. Refer to access specifiers for classes or interfaces in Java
2. **Class name:** The name should begin with an initial letter (capitalized by convention).
3. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.

4. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body is surrounded by braces, { }.

C: Object

It is a basic unit of Object-Oriented Programming and represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

D: Interface

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, nobody).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- A Java library example is Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

E: Array

An array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. The following are some important points about Java arrays.

- In Java, all arrays are dynamically allocated. (discussed below)
- Since arrays are objects in Java, we can find their length using member length. This is different from C/C++ where we find length using size.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each has an index beginning from 0.
- Java array can also be used as a static field, a local variable, or a method parameter.
- The **size** of an array must be specified by an int value and not long or short.
- The direct superclass of an array type is Object.

Operators in Java

1. Arithmetic Operators: They are used to perform simple arithmetic operations on primitive data types.

- *** :** Multiplication
- **/ :** Division
- **% :** Modulo
- **+ :** Addition
- **– :** Subtraction

2. Unary Operators: Unary operators need only one operand. They are used to increment, decrement or negate a value.

- **– : Unary minus**, used for negating the values.
- **+ : Unary plus** indicates the positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is the byte, char, or short. This is called unary numeric promotion.
- **++ : Increment operator**, used for incrementing the value by 1. There are two varieties of increment operators.
    - **Post-Increment:** Value is first used for computing the result and then incremented.
    - **Pre-Increment:** Value is incremented first, and then the result is computed.
- **— : Decrement operator**, used for decrementing the value by 1. There are two varieties of decrement operators.
    - **Post-decrement:** Value is first used for computing the result and then decremented.
    - **Pre-Decrement:** Value is decremented first, and then the result is computed.
- **! : Logical not operator**, used for inverting a boolean value.

3. Assignment Operator: ‘=’ Assignment operator is used to assigning a value to any variable. It has a right to left associativity, i.e. value given on the right-hand side of the operator is assigned to the variable on the left, and therefore right-hand side value must be declared before using it or should be a constant.

The general format of the assignment operator is:

```
variable = value;
```

In many cases, the assignment operator can be combined with other operators to build a shorter version of the statement called a **Compound Statement**. For example, instead of a = a+5, we can write a += 5.

- +=, for adding left operand with right operand and then assigning it to the variable on the left.
- -=, for subtracting right operand from left operand and then assigning it to the variable on the left.
- *=, for multiplying left operand with right operand and then assigning it to the variable on the left.
- /=, for dividing left operand by right operand and then assigning it to the variable on the left.
- %=, for assigning modulo of left operand by right operand and then assigning it to the variable on the left.

4. Relational Operators: These operators are used to check for relations like equality, greater than, and less than. They return boolean results after the comparison and are extensively used in looping statements as well as conditional if-else statements. The general format is,

```
variable relation_operator value
```

- Some of the relational operators are-
    - **==, Equal to** returns true if the left-hand side is equal to the right-hand side.
    - **!=, Not Equal to** returns true if the left-hand side is not equal to the right-hand side.
    - **<, less than:** returns true if the left-hand side is less than the right-hand side.
    - **<=, less than or equal to** returns true if the left-hand side is less than or equal to the right-hand side.
    - **>, Greater than:** returns true if the left-hand side is greater than the right-hand side.
    - **>=, Greater than or equal to** returns true if the left-hand side is greater than or equal to the right-hand side.

5. Logical Operators: These operators are used to perform “logical AND” and “logical OR” operations, i.e., a function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e., it has a short-

circuiting effect. Used extensively to test for several conditions for making a decision. Java also has “Logical NOT”, which returns true when the condition is false and vice-versa

Conditional operators are:

- **&&, Logical AND:** returns true when both conditions are true.
- **||, Logical OR:** returns true if at least one condition is true.
- **!, Logical NOT:** returns true when a condition is false and vice-versa

6. Ternary operator: Ternary operator is a shorthand version of the if-else statement. It has three operands and hence the name ternary.

```
 Here is the syntax for a ternary operator in Java:

variable = (expression) ? expressionIsTrue : expressionIsFalse;
```

The above statement means that if the condition evaluates to true, then execute the statements after the ‘?’ else

execute the statements after the ‘:’.

```
// Java program to illustrate

// max of three numbers using
// ternary operator.

 public class myapp {

    public static void main(String[] args) {
     int val1 = 10;
int val2 = 20;

int max = (val1 >= val2 )? val1 : val2;

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

}}
```

Examples

```
public class EvaluateAge {

        public static void main(String[] args) {
                int age = 22;

                String result = (age >= 16) ? "This user is over 16." : "This
user is under 16."
                System.out.println(result);
        }
}
```

1. Precedence and Associativity: There is often confusion when it comes to hybrid equations which are equations having multiple operators. The problem is which part to solve first. There is a golden rule to follow in these situations. If the operators have different precedence, solve the higher precedence first. If they have the same precedence, solve according to associativity, that is, either from right to left or from left to right. The explanation of the below program is well written in comments within the program itself.

```
public class operators {
        public static void main(String[] args)
        {
        int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;

        // precedence rules for arithmetic operators.
        // (* = / = %) > (+ = -)
        // prints a+(b/d)
        System.out.println("a+b/d = " + (a + b / d));

        // if same precedence then associative
        // rules are followed.
        // e/f -> b*d -> a+(b*d) -> a+(b*d)-(e/f)
        System.out.println("a+b*d-e/f = "
                                        + (a + b * d - e / f));
        }
}
```

Output

```
a+b/d = 20
a+b*d-e/f = 219
```

3. Using + over (): When using + operator inside ***system.out.println()*** make sure to do addition using parenthesis. If we write something before doing addition, then string addition takes place, that is, associativity of addition is left to right, and hence integers are added to a string first producing a string, and string objects concatenate when using +. Therefore it can create unwanted results.

```
public class operators {
	public static void main(String[] args)
	{

	int x = 5, y = 8;

	// concatenates x and y as
	// first x is added to "concatenation (x+y) = "
	// producing "concatenation (x+y) = 5"
	// and then 8 is further concatenated.
	System.out.println("Concatenation (x+y)= " + x + y);

	// addition of x and y
	System.out.println("Addition (x+y) = " + (x + y));
	}
}
```

Output

```
Concatenation (x+y)= 58
Addition (x+y) = 13
```

Java Arithmetic Operators with Examples

```
// Java code to illustrate Addition operator

import java.io.*;

class Addition {
	public static void main(String[] args)
	{
	// initializing variables
	int num1 = 10, num2 = 20, sum = 0;

	// Displaying num1 and num2
	System.out.println("num1 = " + num1);
	System.out.println("num2 = " + num2);

	// adding num1 and num2
	sum = num1 + num2;
	System.out.println("The sum = " + sum);
	}
}
```

Output

```
num1 = 10
num2 = 20
The sum = 30
```

2. Subtraction(-): This operator is a binary operator and is used to subtract two operands.

Syntax:

```
num1 - num2
```

Example:

```
num1 = 20, num2 = 10
sub = num1 - num2 = 10
```

```
// Java code to illustrate Subtraction operator


import java.io.*;

class Subtraction {
	public static void main(String[] args)
	{
		// initializing variables
		int num1 = 20, num2 = 10, sub = 0;

		// Displaying num1 and num2
		System.out.println("num1 = " + num1);
		System.out.println("num2 = " + num2);

		// subtracting num1 and num2
		sub = num1 - num2;
		System.out.println("Subtraction = " + sub);
	}
}
```

Output

```
num1 = 20
num2 = 10
Subtraction = 10
```

Java Assignment Operators with Examples

Types of Assignment Operators in Java

The Assignment Operator is generally of two types. They are:

1. Simple Assignment Operator: The Simple Assignment Operator is used with the “=” sign where the left side consists of the operand and the right side consists of a value. The value of the right side must be of the same data type that has been defined on the left side.

2. Compound Assignment Operator: The Compound Operator is used where +,-,*, and / is used along with the = operator.

Let’s look at each of the assignment operators and how they operate:

1. (=) operator:

This is the most straightforward assignment operator, which is used to assign the value on the right to the variable on the left. This is the basic definition of an assignment operator and how it functions.

Syntax:

```
num1 = num2;
```

Example:

```
a = 10;
ch = 'y';
```

```
// Java code to illustrate "=" operator


import java.io.*;

class Assignment {
	public static void main(String[] args)
	{
		// Declaring variables
		int num;
		String name;

		// Assigning values
		num = 10;
		name = "GeeksforGeeks";

		// Displaying the assigned values
		System.out.println("num is assigned: " + num);
		System.out.println("name is assigned: " + name);
	}
}
```

Output

```
num is assigned: 10
name is assigned: GeeksforGeeks
```

2. (+=) operator:

This operator is a compound of ‘+’ and ‘=’ operators. It operates by adding the current value of the variable on the left to the value on the right and then assigning the result to the operand on the left.

Syntax:

```
num1 += num2;
```

Example:

```
a += 10

This means,
a = a + 10
```

```
// Java code to illustrate "+="


import java.io.*;

class Assignment {
	public static void main(String[] args)
	{

		// Declaring variables
		int num1 = 10, num2 = 20;

		System.out.println("num1 = " + num1);
		System.out.println("num2 = " + num2);

		// Adding & Assigning values
		num1 += num2;

		// Displaying the assigned values
		System.out.println("num1 = " + num1);
	}
}
```

Output

```
num1 = 10
num2 = 20
num1 = 30
```

3. (-=) operator:

This operator is a compound of ‘-‘ and ‘=’ operators. It operates by subtracting the variable’s value on the right from the current value of the variable on the left and then assigning the result to the operand on the left.

Syntax:

```
num1 -= num2;
```

Example:

```
a -= 10

This means,
a = a - 10
```

```
// Java code to illustrate "-="

import java.io.*;

class Assignment {
        public static void main(String[] args)
        {

        // Declaring variables
        int num1 = 10, num2 = 20;

        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2);

        // Subtracting & Assigning values
        num1 -= num2;

        // Displaying the assigned values
        System.out.println("num1 = " + num1);
        }
}
```

Output

```
num1 = 10
num2 = 20
num1 = -10
```

4. (*=) operator:

This operator is a compound of '*' and '=' operators. It operates by multiplying the current value of the variable on the left to the value on the right and then assigning the result to the operand on the left.

Syntax:

```
num1 *= num2;
```

Example:

```
a *= 10
This means,
a = a * 10
```

```
// Java code to illustrate "*="

import java.io.*;

class Assignment {
        public static void main(String[] args)
        {

        // Declaring variables
        int num1 = 10, num2 = 20;

        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2);

        // Multiplying & Assigning values
        num1 *= num2;

        // Displaying the assigned values
        System.out.println("num1 = " + num1);
        }
}
```

5. (%=) operator:

This operator is a compound of ‘%’ and ‘=’ operators. It operates by dividing the current value of the variable on the left by the value on the right and then assigning the remainder to the operand on the left.

Syntax:

```
num1 %= num2;
```

Example:

```
a %= 3

This means,
a = a % 3
```

```
/ Java code to illustrate "%="

import java.io.*;

class Assignment {
    public static void main(String[] args)
    {

        // Declaring variables
        int num1 = 5, num2 = 3;

        System.out.println("num1 = " + num1);
        System.out.println("num2 = " + num2);

        // Moduling & Assigning values
        num1 %= num2;

        // Displaying the assigned values
        System.out.println("num1 = " + num1);
    }
}
```

Output

```
num1 = 5
num2 = 3
num1 = 2
```

Java Unary Operator with Examples

Operator 1: Unary minus(-)

This operator can be used to convert a positive value to a negative one.

Syntax:

```
~(operand)
```

Illustration:

```
a = -10
```

Example:

```
// Java Program to Illustrate Unary - Operator

// Importing required classes
import java.io.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Declaring a custom variable
        int n1 = 20;

        // Printing the above variable
        System.out.println("Number = " + n1);

        // Performing unary operation
        n1 = -n1;

        // Printing the above result number
        // after unary operation
        System.out.println("Result = " + n1);
    }

}
```

Output

```
Number = 20
Result = -20
```

Operator 2: ‘NOT’ Operator(!)

This is used to convert true to false or vice versa. Basically, it reverses the logical state of an operand.

Syntax:

```
!(operand)
```

Illustration:

```
cond = !true;
// cond < false
```

Example:

```
// Java Program to Illustrate Unary NOT Operator

// Importing required classes
import java.io.*;

// Main class
class GFG {

	// Main driver method
	public static void main(String[] args)
	{
		// Initializing variables
		boolean cond = true;
		int a = 10, b = 1;

		// Displaying values stored in above variables
		System.out.println("Cond is: " + cond);
		System.out.println("Var1 = " + a);
		System.out.println("Var2 = " + b);

		// Displaying values stored in above variables
		// after applying unary NOT operator
		System.out.println("Now cond is: " + !cond);
		System.out.println("!(a < b) = " + !(a < b));
		System.out.println("!(a > b) = " + !(a > b));
	}
}
```

Output:

```
Cond is: true
Var1 = 10
Var2 = 1
Now cond is: false
!(a < b) = true
!(a > b) = false
```

Operator 3: Increment(++ & --)

It is used to increment the value of an integer. It can be used in two separate ways:

Example 1

```
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
}}
```

Example 2

```
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



}}
```

Java Relational Operators with Examples

Operator 1: ‘Equal to’ operator (==)

This operator is used to check whether the two given operands are equal or not. The operator returns true if the operand at the left-hand side is equal to the right-hand side, else false.

Syntax:

```
var1 == var2
```

Illustration:

```
var1 = "GeeksforGeeks"
var2 = 20
var1 == var2 results in false
```

Example:

```
// Java Program to Illustrate equal to Operator

// Importing I/O classes
import java.io.*;

// Main class
class GFG {

	// Main driver method
	public static void main(String[] args)
	{
		// Initializing variables
		int var1 = 5, var2 = 10, var3 = 5;

		// Displaying var1, var2, var3
		System.out.println("Var1 = " + var1);
		System.out.println("Var2 = " + var2);
		System.out.println("Var3 = " + var3);

		// Comparing var1 and var2 and
		// printing corresponding boolean value
		System.out.println("var1 == var2: "
						+ (var1 == var2));

		// Comparing var1 and var3 and
		// printing corresponding boolean value
		System.out.println("var1 == var3: "
						+ (var1 == var3));
	}
}
```

Output

```
Var1 = 5
Var2 = 10
Var3 = 5
var1 == var2: false
var1 == var3: true
```

Operator 2: ‘Not equal to’ Operator(!=)

This operator is used to check whether the two given operands are equal or not. It functions opposite to that of the equal-to-operator. It returns true if the operand at the left-hand side is not equal to the right-hand side, else false.

Syntax:

```
var1 != var2
```

Illustration:

```
var1= "GeeksforGeeks"

var2 = 20

var1 != var2 results in true
```

Example:

```
// Java Program to Illustrate No- equal-to Operator

// Importing I/O classes
import java.io.*;

// Main class
class GFG {

	// Main driver method
	public static void main(String[] args)
	{
		// Initializing variables
		int var1 = 5, var2 = 10, var3 = 5;

		// Displaying var1, var2, var3
		System.out.println("Var1 = " + var1);
		System.out.println("Var2 = " + var2);
		System.out.println("Var3 = " + var3);

		// Comparing var1 and var2 and
		// printing corresponding boolean value
		System.out.println("var1 == var2: "
                                                + (var1 != var2));
```

```
                // Comparing var1 and var3 and
                // printing corresponding boolean value
                System.out.println("var1 == var3: "
                                                + (var1 != var3));
        }
}
```

Output

```
Var1 = 5
Var2 = 10
Var3 = 5
var1 == var2: true
var1 == var3: false
```

Operator 3: 'Greater than' operator(>)

This checks whether the first operand is greater than the second operand or not. The operator returns true when the operand at the left-hand side is greater than the right-hand side.

Syntax:

```
var1 > var2
```

Illustration:

```
var1 = 30
var2 = 20

var1 > var2 results in true
```

Example:

```
// Java code to Illustrate Greater than operator


// Importing I/O classes
import java.io.*;

// Main class
class GFG {

        // Main driver method
        public static void main(String[] args)
        {
                // Initializing variables
                int var1 = 30, var2 = 20, var3 = 5;
```

```
                // Displaying var1, var2, var3
                System.out.println("Var1 = " + var1);
                System.out.println("Var2 = " + var2);
                System.out.println("Var3 = " + var3);

                // Comparing var1 and var2 and
                // printing corresponding boolean value
                System.out.println("var1 > var2: " + (var1 > var2));

                // Comparing var1 and var3 and
                // printing corresponding boolean value
                System.out.println("var3 > var1: "
                                                    + (var3 >= var1));
        }
}
```

Java Logical Operators with Examples

test for several conditions for making a decision.

1. **AND Operator** (**&&**) – if(a && b) [if true execute else don’t]
2. **OR Operator** (||) – if(a || b) [if one of them is true execute else don’t]
3. **NOT Operator** (**!**) – !(a<b) [returns false if a is smaller than b]

Example For Logical Operator in Java

Here is an example depicting all the operators where the values of variables a, b, and c are kept the same for all the situations.

```
a = 10, b = 20, c = 30

For AND operator:

Condition 1: c > a
Condition 2: c > b

Output: True [Both Conditions are true]

For OR Operator:

Condition 1: c > a
Condition 2: c > b

Output: True [One of the Condition if true]

For NOT Operator:
Condition 1: c > a
Condition 2: c > b
```

Output: False [Because the result was true and NOT operator did it's opposite]

1. Logical 'AND' Operator (&&)

This operator returns true when both the conditions under consideration are satisfied or are true. If even one of the two yields false, the operator results false. In Simple terms, ***cond1 && cond2 returns true when both cond1 and cond2 are true (i.e. non-zero***).

Syntax:

```
condition1 && condition2
```

Illustration:

```
a = 10, b = 20, c = 20

condition1: a < b
condition2: b == c


if(condition1 && condition2)
d = a + b + c

// Since both the conditions are true
d = 50.
```

Example

```
// Java code to illustrate
// logical AND operator

import java.io.*;

class Logical {
	public static void main(String[] args)
	{
		// initializing variables
		int a = 10, b = 20, c = 20, d = 0;

		// Displaying a, b, c
		System.out.println("Var1 = " + a);
		System.out.println("Var2 = " + b);
		System.out.println("Var3 = " + c);

		// using logical AND to verify
		// two constraints
		if ((a < b) && (b == c)) {
			d = a + b + c;
			System.out.println("The sum is: " + d);
		}
		else
			System.out.println("False conditions");
	}
```

```
}
```

Output:

```
Var1 = 10
Var2 = 20
Var3 = 20
The sum is: 50
```

Now in the below example, we can see the short-circuiting effect. Here when the execution reaches to if statement, the first condition inside the if statement is false and so the second condition is never checked. Thus the ++b(pre-increment of b) never happens and b remains unchanged.

Example:

```
import java.io.*;

class shortCircuiting {
	public static void main(String[] args)
	{

		// initializing variables
		int a = 10, b = 20, c = 15;

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

		// Using logical AND
		// Short-Circuiting effect as the first condition is
		// false so the second condition is never reached
		// and so ++b(pre increment) doesn't take place and
		// value of b remains unchanged
		if ((a > c) && (++b > c)) {
			System.out.println("Inside if block");
		}

		// displaying b
		System.out.println("Value of b : " + b);
	}
}
```

Output:

```
Value of b : 20
Value of b : 20
```

2. Logical ‘OR’ Operator (||)

This operator returns true when one of the two conditions under consideration is satisfied or is true. If even one of the two yields true, the operator results true. To make the result false, both the constraints need to return false.

Syntax:

```
condition1 || condition2
```

Example:

```
a = 10, b = 20, c = 20

condition1: a < b
condition2: b > c

if(condition1 || condition2)
d = a + b + c

// Since one of the condition is true

d = 50.
```

```
// Java code to illustrate

// logical OR operator

import java.io.*;

class Logical {
	public static void main(String[] args)
	{
		// initializing variables
		int a = 10, b = 1, c = 10, d = 30;

		// Displaying a, b, c
		System.out.println("Var1 = " + a);
		System.out.println("Var2 = " + b);
		System.out.println("Var3 = " + c);
		System.out.println("Var4 = " + d);

		// using logical OR to verify
		// two constraints
		if (a > b || c == d)
			System.out.println("One or both + the conditions are true");
		else
			System.out.println("Both the + conditions are false");
	}
}
```

Output:

```
Var1 = 10
Var2 = 1
Var3 = 10
Var4 = 30
One or both the conditions are true
```

3. Logical ‘NOT’ Operator (!)

Unlike the previous two, this is a unary operator and returns true when the condition under consideration is not satisfied or is a false condition. Basically, if the condition is false, the operation returns true and when the condition is true, the operation returns false.

Syntax:

```
!(condition)
```

Example:

```
a = 10, b = 20

!(a<b) // returns false
!(a>b) // returns true

Example
```

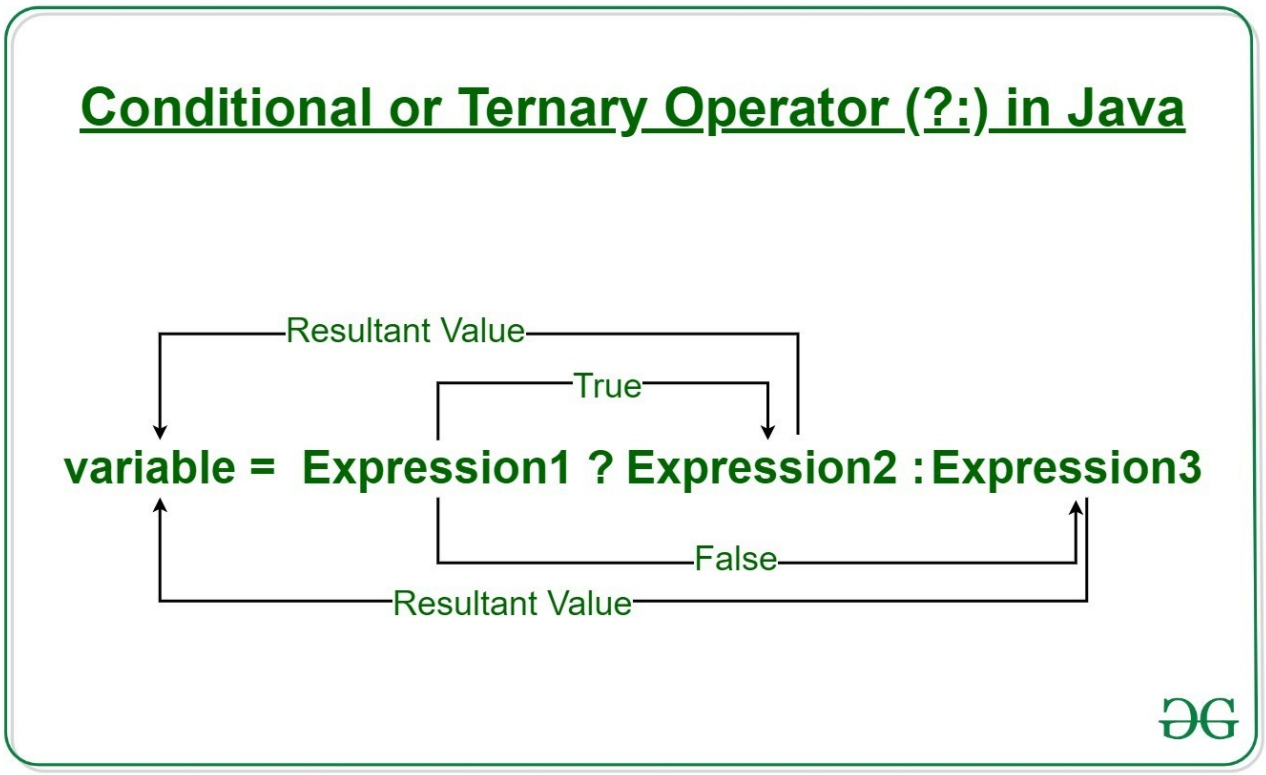
```
// Java code to illustrate

// logical NOT operator
import java.io.*;

class Logical {
	public static void main(String[] args)
	{
		// initializing variables
		int a = 10, b = 1;

		// Displaying a, b, c
		System.out.println("Var1 = " + a);
		System.out.println("Var2 = " + b);

		// Using logical NOT operator
		System.out.println("!(a < b) = " + !(a < b));
		System.out.println("!(a > b) = " + !(a > b));
	}
}
```

Output:

```
Var1 = 10
Var2 = 1
!(a < b) = true
!(a > b) = false
```

Java Ternary Operator with Examples

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. Although it follows the same algorithm as of if-else statement, the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.



Syntax:

```
variable = Expression1 ? Expression2: Expression3
```

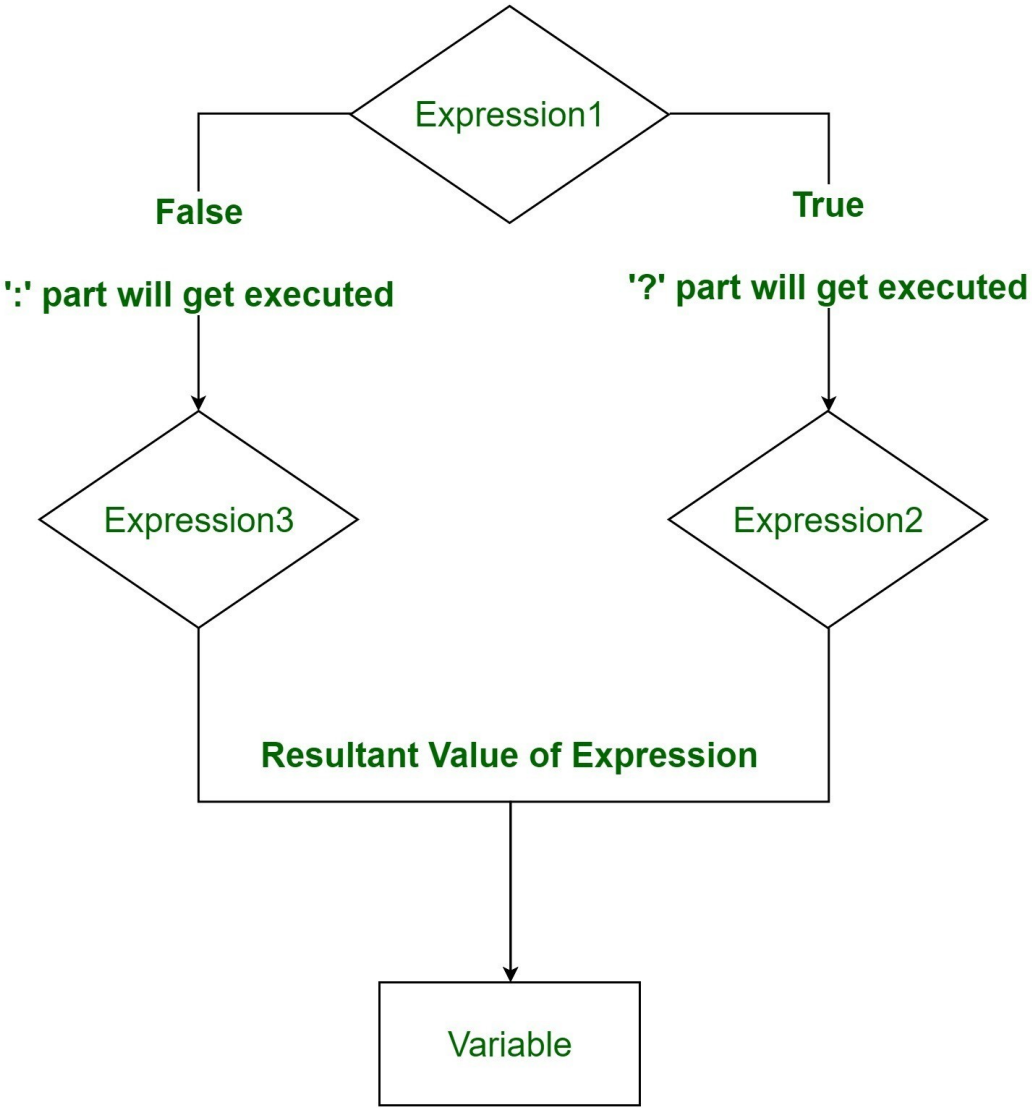
If operates similarly to that of the if-else statement as in *Exression2* is executed if *Expression1* is true else *Expression3* is executed.

```
if(Expression1)
{
    variable = Expression2;
}
else
{
    variable = Expression3;
}
```

Example:

```
num1 = 10;
num2 = 20;

res=(num1>num2) ? (num1+num2):(num1-num2)

Since num1<num2,
the second operation is performed
res = num1-num2 = -10
```

Flow Chart of
Conditional or Ternary Operator



Example 1:

```
// Java program to find largest among two
// numbers using ternary operator

import java.io.*;

class Ternary {
	public static void main(String[] args)
	{

	// variable declaration
	int n1 = 5, n2 = 10, max;

	System.out.println("First num: " + n1);
	System.out.println("Second num: " + n2);

	// Largest among n1 and n2
	max = (n1 > n2) ? n1 : n2;

	// Print the largest number
	System.out.println("Maximum is = " + max);
	}
}
```

Output

```
First num: 5
Second num: 10
Maximum is = 10
```

Example 2:

```
// Java code to illustrate ternary operator

import java.io.*;

class Ternary {
        public static void main(String[] args)
        {

        // variable declaration
        int n1 = 5, n2 = 10, res;

        System.out.println("First num: " + n1);
        System.out.println("Second num: " + n2);

        // Performing ternary operation
        res = (n1 > n2) ? (n1 + n2) : (n1 - n2);

        // Print the largest number
        System.out.println("Result = " + res);
        }
}
```

Output

```
First num: 5
Second num: 10
Result = -5
```

Decision Making in Java (if, if-else, switch, break, continue, jump)

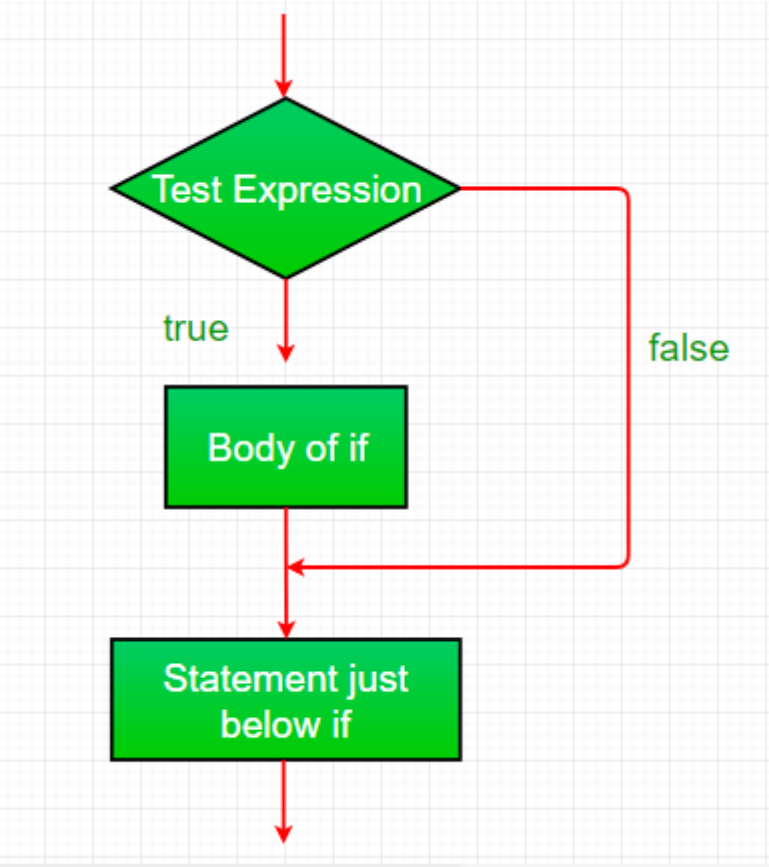
Decision Making in programming is similar to decision-making in real life. In programming also face some situations where we want a certain block of code to be executed when some condition is fulfilled.

A programming language uses control statements to control the flow of execution of a program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.

1. if: if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
{
   // Statements to execute if
   // condition is true
}
```



example

```
// Java program to illustrate If statement

class IfDemo {
        public static void main(String args[])
        {
                int i = 10;

                if (i > 15)
                        System.out.println("10 is less than 15");

                // This statement will be executed
                // as if considers one statement by default
                System.out.println("I am Not in if");
        }
}
```

2. if-else: The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax:

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```

Example

```
// Java program to illustrate if-else statement
class IfElseDemo {
   public static void main(String args[])
   {
      int i = 10;

      if (i < 15)
         System.out.println("i is smaller than 15");
      else
         System.out.println("i is greater than 15");
   }
}
```

3. nested-if: A nested if is an if statement that is the target of another if or else. Nested if statements mean an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
{
   // Executes when condition1 is true
   if (condition2)
   {
      // Executes when condition2 is true
   }
}
```

Example :

```
// Java program to illustrate nested-if statement

class NestedIfDemo {
	public static void main(String args[])
	{
		int i = 10;

		if (i == 10 || i<15) {
			// First if statement
			if (i < 15)
				System.out.println("i is smaller than 15");

			// Nested - if statement
			// Will only be executed if statement above
			// it is true
			if (i < 12)
				System.out.println(
					"i is smaller than 12 too");
		} else{
				System.out.println("i is greater than 15");
		}
	}
}
```

Output

```
i is smaller than 15
i is smaller than 12 too
```

Example

```
// Java program to illustrate if-else-if ladder

class ifelseifDemo {
	public static void main(String args[])
	{
		int i = 20;

		if (i == 10)
			System.out.println("i is 10");
		else if (i == 15)
			System.out.println("i is 15");
		else if (i == 20)
			System.out.println("i is 20");
		else
			System.out.println("i is not present");
	}
}
```

5. switch-case: The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

Example

```
public class Main {
  public static void main(String[] args) {
    int day = 4;
    switch (day) {
      case 1:
        System.out.println("Monday");
        break;
      case 2:
        System.out.println("Tuesday");
        break;
      case 3:
        System.out.println("Wednesday");
        break;
      case 4:
        System.out.println("Thursday");
        break;
      case 5:
        System.out.println("Friday");
        break;
      case 6:
        System.out.println("Saturday");
        break;
      case 7:
        System.out.println("Sunday");
        break;
    }
```

```
 }

}
```

Example 2

```
int day = 4;
switch (day) {
  case 6:
    System.out.println("Today is Saturday");
    break;
  case 7:
    System.out.println("Today is Sunday");
    break;
  default:
    System.out.println("Looking forward to the Weekend");
}
// Outputs "Looking forward to the Weekend"
```

Java if statement with Examples

Example 1

```
// Java program to illustrate If statement

class IfDemo {
	public static void main(String args[])
	{
		int i = 10;

		if (i < 15)
			System.out.println("10 is less than 15");

		// This statement will be executed
		// as if considers one statement by default
		System.out.println("Outside if-block");
	}
}
```

Output:

```
10 is less than 15
Outside if-block
```

Example 2:

```
// Java program to illustrate If statement

class IfDemo {
	public static void main(String args[])
	{
		String str = "GeeksforGeeks";
		int i = 4;

		// if block
		if (i == 4) {
			i++;
			System.out.println(str);
		}

		// Executed by default
```

```
                System.out.println("i = " + i);
        }
}
```

Java if-else statement with Examples

Example 1:

```
// Java program to illustrate if-else statement

class IfElseDemo {
        public static void main(String args[])
        {
                int i = 20;

                if (i < 15)
                        System.out.println("i is smaller than 15");
                else
                        System.out.println("i is greater than 15");

                System.out.println("Outside if-else block");
        }
}
```

Example 2

```
// Java program to illustrate if-else-if ladder

import java.io.*;

class GFG {
        public static void main(String[] args)
        {
                // initializing expression
                int i = 20;

                // condition 1
                if (i == 10)
                        System.out.println("i is 10\n");

                // condition 2
                else if (i == 15)
                        System.out.println("i is 15\n");

                // condition 3
                else if (i == 20)
                        System.out.println("i is 20\n");
```

```
                else
                        System.out.println("i is not present\n");

                System.out.println("Outside if-else-if");
        }
}
```

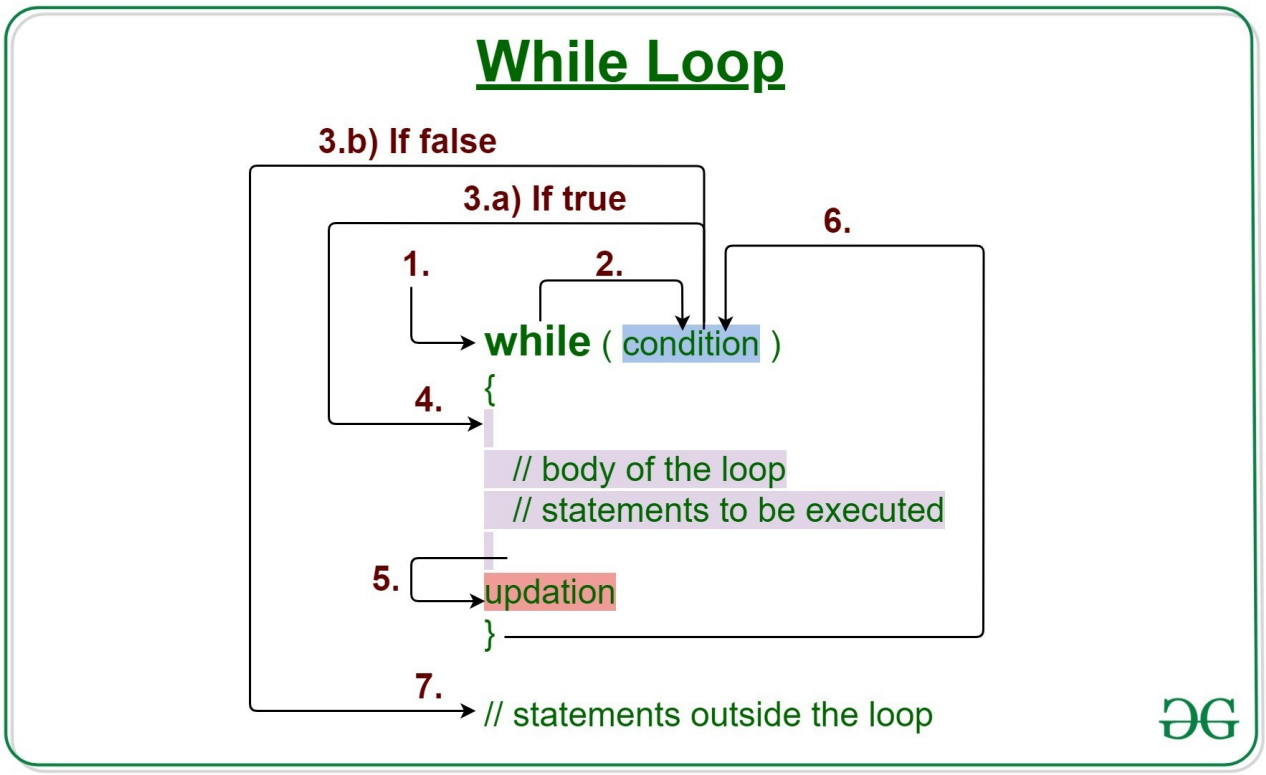
Example 3

```
// Java program to illustrate if-else-if ladder

import java.io.*;

class GFG {
        public static void main(String[] args)
        {

                // initializing expression
                int i = 20;

                // condition 1
                if (i < 10)
                        System.out.println("i is less than 10\n");

                // condition 2
                else if (i < 15)
                        System.out.println("i is less than 15\n");

                // condition 3
                else if (i < 20)
                        System.out.println("i is less than 20\n");

                else
                        System.out.println("i is greater than "
                                                        + "or equal to 20\n");

                System.out.println("Outside if-else-if");
        }
}
```

Loops in Java

Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true. Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

Java while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement. While loop in Java comes into use when we need to repeatedly execute a block of statements. The while loop is considered as a repeating if statement. If the number of iterations is not fixed, it is recommended to use the while loop.



Syntax:

```
while (test_expression)
{
   // statements

  update_expression;
}
```

The various **parts of the While loop** are:

1. Test Expression: In this expression, we have to test the condition. If the condition evaluates to true then we will execute the body of the loop and go to update expression. Otherwise, we will exit from the while loop.

Example:

```
i <= 10
```

2. Update Expression: After executing the loop body, this expression increments/decrements the loop variable by some value.

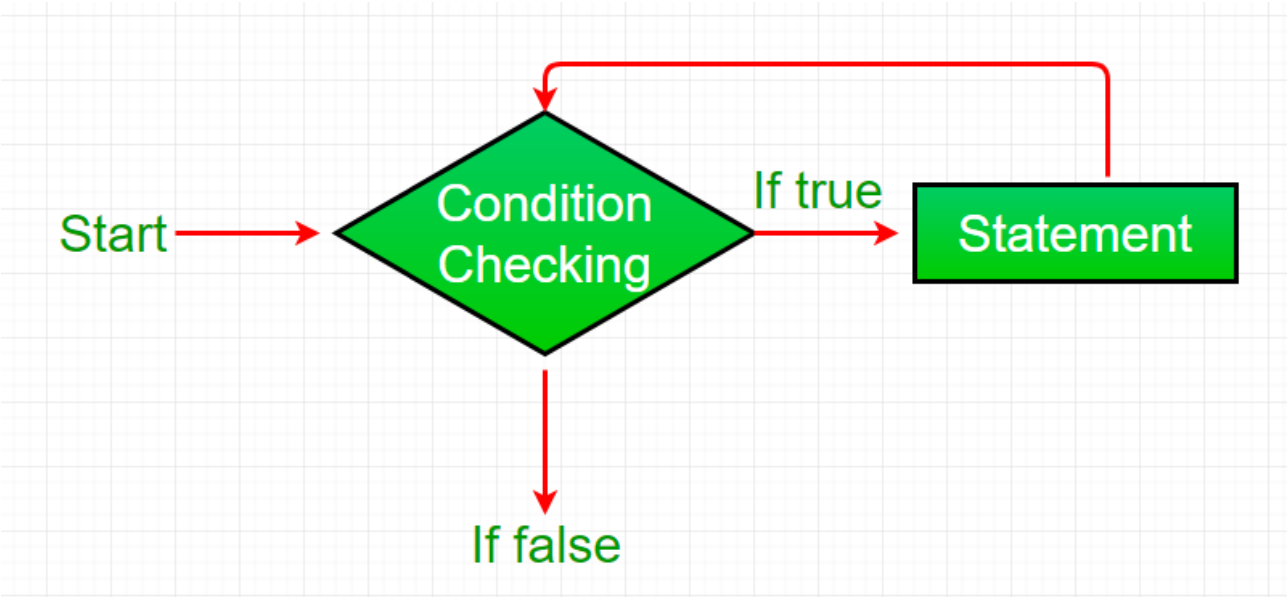
Example:

```
i++;
```

How Does a While loop execute?

1. Control falls into the while loop.
2. The flow jumps to Condition
3. Condition is tested.
    - If Condition yields true, the flow goes into the Body.
    - If Condition yields false, the flow goes outside the loop
4. The statements inside the body of the loop get executed.
5. Updation takes place.
6. Control flows back to Step 2.
7. The while loop has ended and the flow has gone outside.

Flowchart For while loop (Control Flow):



Example 1: This program will try to print “Hello World” 5 times.

```
// Java program to illustrate while loop.

class whileLoopDemo {

	public static void main(String args[])

	{

		// initialization expression

		int i = 1;



		// test expression

		while (i < 6) {

			System.out.println("Hello World");



			// update expression

			i++;

		}

	}

}
```

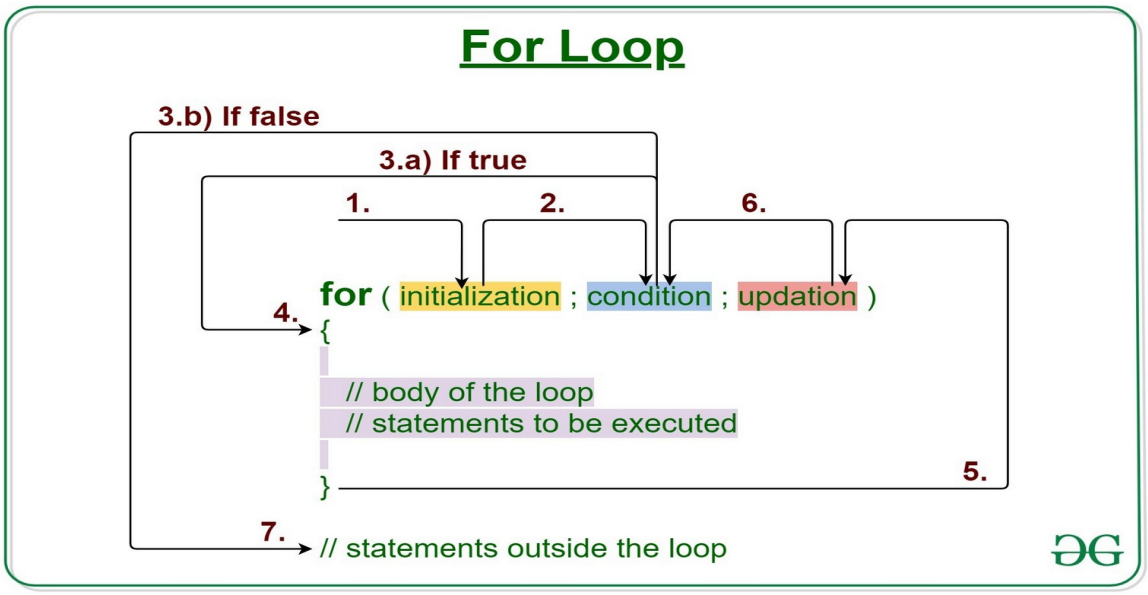
Example 2: This program will find the summation of numbers from 1 to 10.

```
 // Java program to illustrate while loop
```

```
class whileLoopDemo {

	public static void main(String args[])

	{

		int x = 1, sum = 0;



		// Exit when x becomes greater than 4

		while (x <= 10) {

			// summing up x

			sum = sum + x;



			// Increment the value of x for

			// next iteration

			x++;

		}

		System.out.println("Summation: " + sum);

	}

}
```

Java For loop with Examples

Loops in Java come into use when we need to repeatedly execute a block of statements. **Java for loop** provides a concise way of writing the loop structure. The for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.



Syntax:

```
for (initialization expr; test expr; update exp)
{
     // body of the loop
     // statements we want to execute
}
```

The various **parts of the For loop** are:

1. **Initialization Expression:** In this expression, we have to initialize the loop counter to some value.

Example:

```
int i=1;
```

2. **Test Expression:** In this expression, we have to test the condition. If the condition evaluates to true then, we will execute the body of the loop and go to update expression. Otherwise, we will exit from the for loop.

Example:

```
i <= 10
```

3. **Update Expression**: After executing the loop body, this expression increments/decrements the loop variable by some value.

Example:

```
i++;
```

How does a For loop execute?

1. Control falls into the for loop. Initialization is done
2. The flow jumps to Condition
3. Condition is tested.
    1. If Condition yields true, the flow goes into the Body
    2. If Condition yields false, the flow goes outside the loop
4. The statements inside the body of the loop get executed.
5. The flow goes to the Updation
6. Updation takes place and the flow goes to Step 3 again
7. The for loop has ended and the flow has gone outside.

Example 1: This program will print 1 to 10

```
/*package whatever //do not write package name here */

// Java program to write a code in for loop from 1 to 10



class GFG {

	public static void main(String[] args)

	{

		for (int i = 1; i <= 10; i++) {

			System.out.println(i);

		}

	}

}
```

Example 2: This program will try to print “Hello World” 5 times.

```
// Java program to illustrate for loop

class forLoopDemo {

	public static void main(String args[])

	{

		// Writing a for loop

		// to print Hello World 5 times

		for (int i = 1; i <= 5; i++)

			System.out.println("Hello World");

	}

}
```

Example 3: The following program prints the sum of x ranging from 1 to 20.

```
// Java program to illustrate for loop.

class forLoopDemo {

	public static void main(String args[])

	{

		int sum = 0;



		// for loop begins

		// and runs till x <= 20

		for (int x = 1; x <= 20; x++) {

			sum = sum + x;
```

```
                }

                System.out.println("Sum: " + sum);

        }

}
```

Enhanced For Loop or Java For-Each loop

Java also includes another version of for loop introduced in Java 5. Enhanced for loop provides a simpler way to iterate through the elements of a collection or array. It is inflexible and should be used only when there is a need to iterate through the elements in a sequential manner without knowing the index of the currently processed element.

> **Note:** The object/variable is immutable when enhanced for loop is used i.e it ensures that the values in the array can not be modified, so it can be said as a read-only loop where you can't update the values as opposed to other loops where values can be modified.

Syntax:

```
for (T element:Collection obj/array)
{
    // loop body
    // statement(s)
}
```

Let's take an example to demonstrate how enhanced for loop can be used to simplify the work. Suppose there is an array of names and we want to print all the names in that array. Let's see the difference between these two examples by this simple implementation:

```
// Java program to illustrate enhanced for loop



public class enhancedforloop {



        public static void main(String args[])

        {

                String array[] = { "Ron", "Harry", "Hermoine" };



                // enhanced for loop

                for (String x : array) {

                        System.out.println(x);

                }



                /* for loop for same function

                for (int i = 0; i < array.length; i++)

                {

                        System.out.println(array[i]);

                }

                */

        }

}
```

Output

```
Ron
Harry
Hermoine
```

Time Complexity: O(1), **Auxiliary Space :** O(1)

Recommendation: Use this form of statement instead of the general form whenever possible. (as per JAVA doc.)

JAVA Infinite for loop:

This is an infinite loop as the condition would never return false. The initialization step is setting up the value of variable i to 1, since we are incrementing the value of i, it would always be greater than 1 so it would never return false. This would eventually lead to the infinite loop condition.

Example:

```
// Java infinite loop

class GFG {

	public static void main(String args[])

	{

		for (int i = 1; i >= 1; i++) {

			System.out.println("Infinite Loop " + i);

		}

	}

}
```

There is another method for calling the Infinite for loop

If you use two **semicolons ;;** in the for loop, it will be infinitive for loop.

Syntax:

```
for(;;){
//code to be executed
}
```

Example:

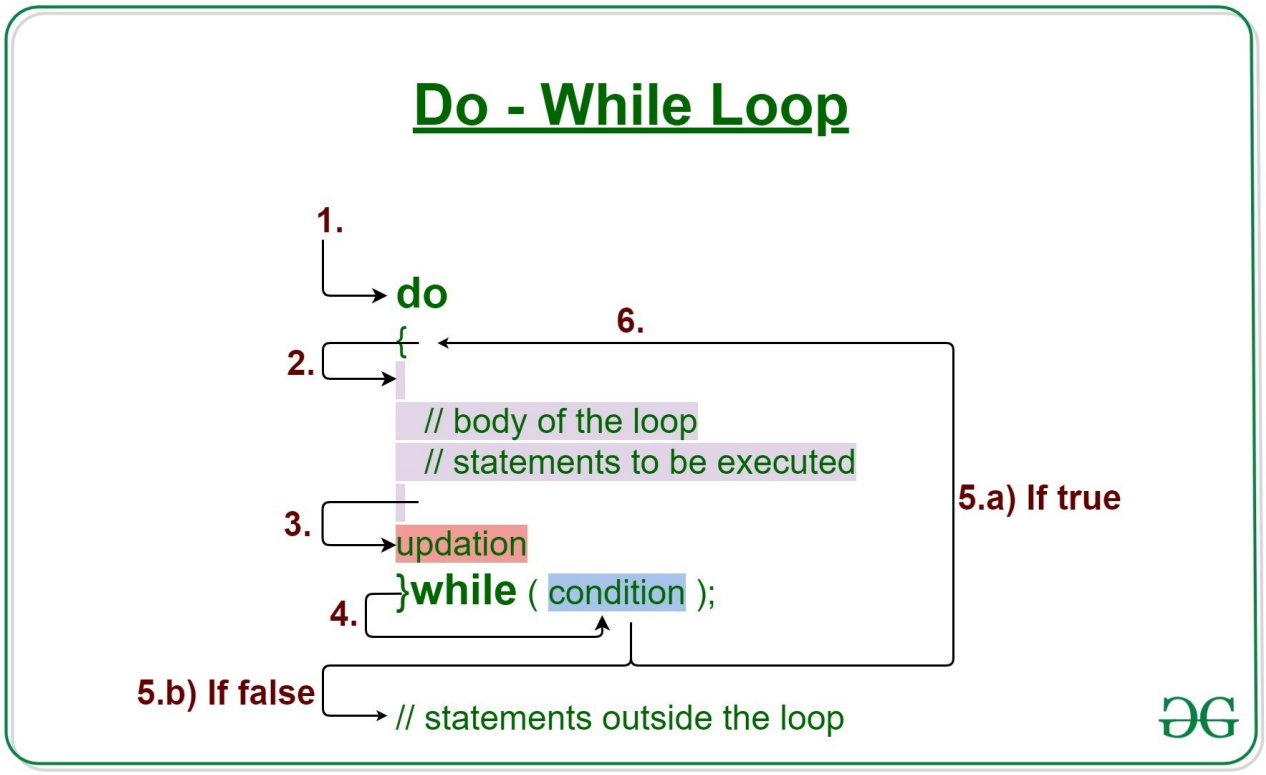
```
public class GFG {



        public static void main(String[] args)

        {

                for (;;) {

                        System.out.println("infinitive loop");

                }

        }

}
```

Java do-while loop with Examples

Loops in Java come into use when we need to repeatedly execute a block of statements. Java **do-while loop** is an **Exit control loop**. Therefore, unlike ***for*** or ***while*** loop, a do-while check for the condition after executing the statements of the loop body.

Syntax:

```
do
{
    // Loop Body
    Update_expression
}

// Condition check
while (test_expression);
```




```
// Java Program to Illusterate One Time Iteration

// Inside do-while Loop

// When Condition IS Not Satisfied

// Class

class GFG {

	// Main driver method

	public static void main(String[] args)

	{

		// initial counter variable

		int i = 0;

		do {
```

```
                    // Body of loop that will execute minimum
                    // 1 time for sure no matter what
                    System.out.println("Print statement");
                    i++;
             }
             // Checking condition
             // Note: It is being checked after
             // minimum 1 iteration
             while (i < 0);
      }
}
```

Example 1: This program will try to print “Hello World” 5 times.

```
// Java Program to Illustrate Do-while Loop
// Class
class GFG {
      // Main driver method
      public static void main(String args[])
      {
             // Declaring and initialization expression
             int i = 1;
             // Do-while loop
             do {
                    // Body od do-while loop
                    // Print statement
                    System.out.println("Hello World");
                    // Update expression
```

```
                  i++;
            }
            // Test expression
            while (i < 6);
      }
}
```

Example 2

```
// Java Program to Illustrate Do-while Loop
// Class
class GFG {
      // Main driver method
      public static void main(String args[])
      {
            // Declaring and initializing integer values
            int x = 21, sum = 0;
            // Do-while loop
            do {
                  // Execution statements(Body of loop)
                  // Here, the line will be printed even
                  // if the condition is false
                  sum += x;
                  x--;
            }
            // Now checking condition
            while (x > 10);
            // Summing up
            System.out.println("Summation: " + sum);
      }
}
```