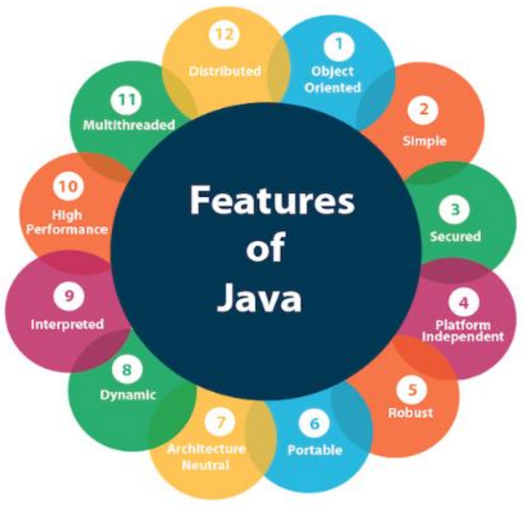
Features of Java

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java *buzzwords*.

A list of most important features of Java language is given below.

1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Robust
6. Architecture neutral
7. Interpreted
8. High Performance
9. Multithreaded
10. Distributed
11. Dynamic



Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Object-oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism

5. Abstraction
6. Encapsulation

Platform Independent

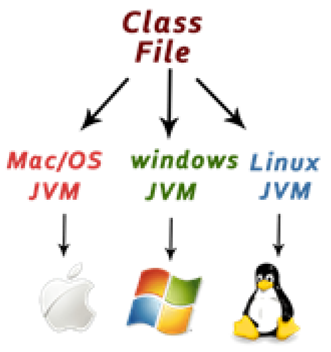
Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

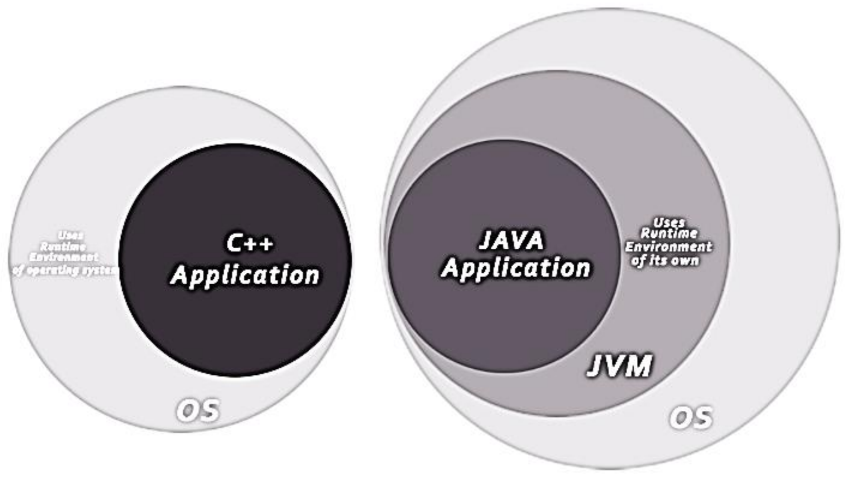
Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).



Secured

Java is best known for its security. With Java, we can develop virus-f

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**


Uses Runtime Environment of operating system
C++ Application
OS
JAVA Application
Uses Runtime Environment of its own
JVM
OS

- **Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.

- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

Robust

Robust simply means strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI (Remote Methode Invocation) and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

RMI : java Remote Method Invocation is used in distributed systems to remotely invoke the methods of java stored object in network .RMI enable two different objects in different Java Virtual Machine to communicate with each other in network by calling methods remotely

EJB : Enterprise Java Beans is server side software component that encapsulate business Logic of an Application

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

- **Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.
- **Class** − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

Objects in Java

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Classes in Java

A class is a blueprint from which individual objects are created.

Following is a sample of a class.

Example

```
public class Dog {
   String breed;
   int age;
   String color;
   void barking() {
   }
   void hungry() {
   }
   void sleeping() {
   }
}
```

A class can contain any of the following variable types.

- **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.Following are some of the important topics that need to be discussed when looking into classes of the Java Language.

Creating an Object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class −

- **Declaration** − A variable declaration with a variable name with an object type.
- **Instantiation** − The 'new' keyword is used to create the object.
- **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object

```
public class Puppy {
   public Puppy(String name) {
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }

   public static void main(String []args) {
      // Following statement would create an object myPuppy
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

Accessing Instance Variables and Methods

Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path −

```
/* First create an object */
ObjectReference = new Constructor();
```

```
/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Example

This example explains how to access instance variables and methods of a class.

Live Demo

```
public class Puppy {
   int puppyAge;

   public Puppy(String name) {
      // This constructor has one parameter, name.
      System.out.println("Name chosen is :" + name );
   }

   public void setAge( int age ) {
      puppyAge = age;
   }

   public int getAge( ) {
      System.out.println("Puppy's age is :" + puppyAge );
      return puppyAge;
   }

   public static void main(String []args) {
      /* Object creation */
      Puppy myPuppy = new Puppy( "tommy" );

      /* Call class method to set puppy's age */
      myPuppy.setAge( 2 );

      /* Call another class method to get puppy's age */
      myPuppy.getAge( );

      /* You can access instance variable as follows as well */
      System.out.println("Variable Value :" + myPuppy.puppyAge );
   }
}
```

If we compile and run the above program, then it will produce the following result −

Output

```
Name chosen is :tommy
Puppy's age is :2
Variable Value :2
```

Java Package

In simple words, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

Import Statements

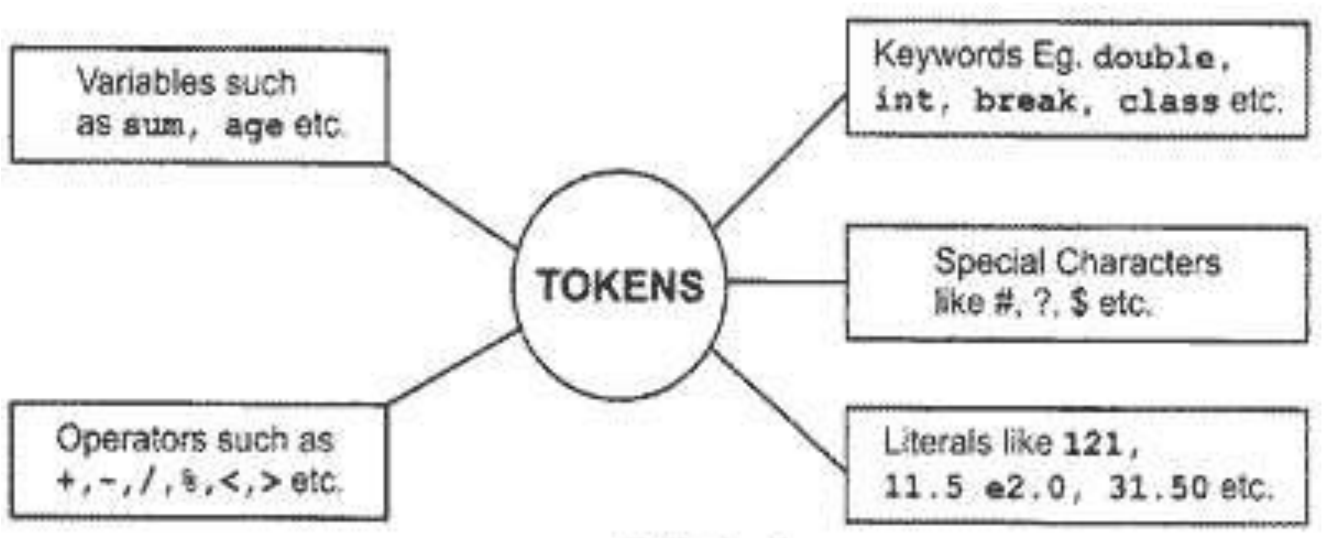
In Java if a fully qualified name, which includes the package and the class name is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example, the following line would ask the compiler to load all the classes available in directory java_installation/java/io −

Java Tokens - What is Java Tokens?

Java Tokens:- A java Program is made up of Classes and Methods and in the Methods are the Container of the various Statements And a Statement is made up of Variables, Constants, operators etc .

Tokens are the various Java program elements which are identified by the compiler. A token is the smallest element of a program that is meaningful to the compiler. Tokens supported in Java include keywords, variables, constants, special characters, operations etc.


Variables such as sum, age etc
Keywords Eg. double, int, break, class etc.
TOKENS
Special Characters like #, ?, $ etc.
Operators such as +,-,/,%,<,> etc.
Literals like 121, 11.5 e2.0, 31.50 etc.

Tokens in Java

Tokens are the various Java program elements which are identified by the compiler. A token is the smallest element of a program that is meaningful to the compiler. Tokens supported in Java include keywords, variables, constants, special characters, operations etc.

Token in Java

When you compile a program, the compiler scans the text in your source code and extracts individual tokens. While tokenizing the source file, the compiler recognizes and subsequently removes whitespaces (spaces, tabs, newline and form feeds) and the text enclosed within comments. Now let us consider a program.

```
//Print Hello
Public class Hello
{
```

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

Token	Meaning
Keyword	A variable is a meaningful name of data storage location in computer memory. When using a variable you refer to memory address of computer
Constant	Constants are expressions with a fixed value
Identifier	The term identifier is usually used for variable names
String	Sequence of characters
Special Symbol	Symbols other than the Alphabets and Digits and white-spaces
Operators	A symbol that represents a specific mathematical or non-mathematical action

No	Token Type	Example 1	Example 2
1	Keyword	int	for
2	Constants	height	sum
3	Identifier	-443	43
4	String	"Hello"	"atnyla"
5	Special Symbol	*	@
6	Operators	*	++

The source code contains tokens such as public, class, Hello, {, public, static, void, main, (, String, [], args, {, System, out, println, (, "Hello Java", }, }. The resulting tokens· are compiled into Java bytecodes that is capable of being run from within an interpreted java environment. Token are useful for compiler to detect errors. When tokens are not arranged in a particular sequence, the compiler generates an error message.

Java program is basically a collection of classes. But it have a basic bulding blocks, these basic buildings blocks in Java language which are constructed together to write a java program. This basic bulding blocks are called Token.

Each and every smallest individual units in a Java program are known as C tokens

Simply we can say that java program is also collection of tokens, comments and whitespaces.

In Java Programming tokens are of six types. They are,

Keyword

Keywords are an special part of a language definition. Keywords are predefined, reserved words used in programming that have special meanings to the compiler. These meaning of the keywords has already been described to the java compiler. These meaning cannot be changed. Thus, the keywords cannot be used as variable names because that would try to change the existing meaning of the keyword, which is not allowed. Keywords are part of the syntax and they cannot be used as an identifier. But one can change these words as identifiers by changing one or more letters to upper case, How ever this will be a bad practice so we should avoid that.

Java language has reserved 50 words as keywords

All the Keywords are written in lower-case letters. Since java is case-sensitive

```
int distance ;

int roll_mo ;

float area ;
```

Here, int, float are a keyword that indicates 'distance', 'roll_no', 'area' are a variable of type integer.

Java Language Keywords

Here is a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs. The keywords const and goto are reserved, even though they are not currently used. true, false, and null might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

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

Identifiers

Identifiers are the names of variables, methods, classes, packages and interfaces

Identifier must follow some rules. Here are the rules:

- All identifiers must start with either a letter(a to z or A to Z) or currency character($) or an underscore.
- They must not begin with a digit
- After the first character, an identifier can have any combination of characters.
- A Java **keywords** cannot be used as an identifier.
- Identifiers in Java are case sensitive, foo and Foo are two different identifiers.
- They can be any length

Each variable has a name by which it is identified in the program. It's a good idea to give your variables mnemonic names that are closely related to the values they hold. Variable names can include any alphabetic character or digit and the underscore _. The main restriction on the names you can give your variables is that they cannot contain any white space. You cannot begin a variable name with a number. It is important to note that as in C but not as in Fortran or Basic, all variable names are case-sensitive. MyVariable is not the same as myVariable. There is no limit to the length of a Java variable name. The following are legal variable names:

	MYVARIABLE	_9pins
	x	andros
	i	ανδρος
MyVariable	_myvariable	OReilly
myvariable	$myvariable	

- This_is_an_insanely_long_variable_name_that_just_keeps_going_and_going_and_going_and_well_you_get_the_idea_The_line_breaks_arent_really_part_of_the_variable_name_Its_just_that_this_variable_name_is_so_ridiculously_long_that_it_won't_fit_on_the_page_I_cant_imagine_why_you_would_need_such_a_long_variable_name_but_if_you_do_you_can_have_it

The following are not legal variable names:

- My Variable // Contains a space
- 9pins // Begins with a digit
- a+c // The plus sign is not an alphanumeric character
- testing1-2-3 // The hyphen is not an alphanumeric character
- O'Reilly // Apostrophe is not an alphanumeric character
- OReilly_&_Associates // ampersand is not an alphanumeric character

If you want to begin a variable name with a digit, prefix the name you'd like to have (e.g. 8number) with an underscore, e.g. _8number. You can also use the underscore to act like a space in long variable names.

classes and interfaces start with a leading uppercase, the second and subsequent words are marked with a leading uppercase letters.

Examples:

```
Employee
Area
Animal
MotorCycle
FurnitureLength
etc.
```

public methods and instance variables starts with a leading lowercase letters.

Examples:

```
suma
area
addition
multiplication
etc.
```

When more than one words are in a name, the second and subsequent words are marked with a leading uppercase letters.

Examples:

```
areaCircle
buisnessName
averageResult
etc.
```

Private and local variables use only lowercase combined with underscore.

Examples:

```
volume
area_circle
distance_moon
etc.
```

Variables that represents constant values use all uppercse letters and underscore between words

Examples:

```
TOTAL_MARKS
GROSS_SALARY
PI_VALUE
etc.
```

These are not rules but these are conventions.

Constants or Literals

Literals in java are sequence of characters (digits, letters and other characters) that represent constant values to be stored in variables. Java language specifies five major type of literals. They are below :

Constant	**Type of Value Stored**
Integer Literals	Literals which stores integer value
Floating Literals	Literals which stores float value
Character Literals	Literals which stores character value
String Literals	Literals which stores string value
Boolean Literals	Literals which stores true or false

Learn More about literals or Constant

String

In C and C++, strings are nothing but an array of characters ended with a null character (‘\0’).This null character indicates the end of the string. Strings are always enclosed in double quotes. Whereas, a character is enclosed in single quotes in C and C++.**Declarations for String:**

- char string[20] = {‘g’, ’e’, ‘e’, ‘k’, ‘s’, ‘f’, ‘o’, ‘r’, ‘g’, ’e’, ‘e’, ‘k’, ‘s’, ‘\0’};
- char string[20] = "atnyla";
- char string [] = "atnyla";

In java, string is basically an object that represents sequence of char values. An array of characters works same as java string. For example:

1. char[] ch={'a','t','n','y','l','a'};
2. String s=new String(ch);
3. is same as:

1. String s="atnyla";

Java String class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

Learn More about Strings

Special symbol

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.[] () {}, ; * = #

,	<	>	.	_
(	)	;	$	:
%	[	]	#	?
'	&	{	}	"
^	!	*	/	|
-	\	~	+	

- **Brackets[]:** Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.
- **Parentheses():** These special symbols are used to indicate function calls and function parameters.
- **Braces{}:** These opening and ending curly braces mark the start and end of a block of code containing more than one executable statement.
- **semicolon (;):** It is used to separate more than one statements like in for loop is separates initialization, condition, and increment.
- **comma (,):** It is an operator that essentially invokes something called an initialization list.
- **an asterisk (*):** It is used for mutiplication.
- **assignment operator (=):** It is used to assign values.
- **preprocessor(#):** The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.

Learn More about Special symbol

Operators

An operator is a symbol that takes one or more arguments and operates on them to produce a result. Operators are of many types and are considered in operator

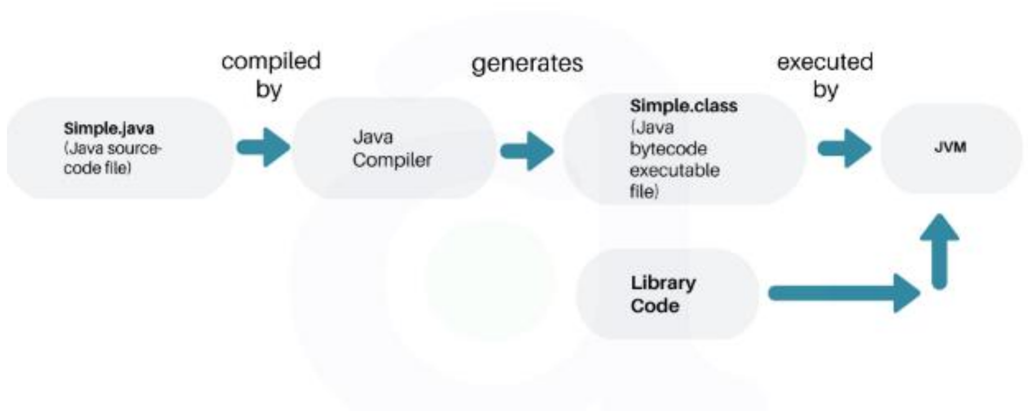
There are many types of operators in java which are given below:

- Unary Operator,
- Arithmetic Operator,
- shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

compilation and Execution process

You have to create your program and compile it before it can be executed.You can use any text editor or IDE to create and edit a Java source-code file, source file is officially called a compilation unit. This process is repetitive, as shown in below.For most computer languages, the name of the file that holds the source code to a program is immaterial. However, this is not the case with Java. The first thing that you must learn about Java is that the name you give to a source file is very important. For this example, the name of the source file should be Simple.java.

If your program has compile errors, you have to modify the program to fix them, then recompile it. If your program has runtime errors or does not produce the correct result, you have to modify the program, recompile it, and execute it again.The Java language is a high-level language, but Java bytecode is a low-level language. The bytecode is similar to machine instructions but is architecture neutral and can run on any platform that has a Java Virtual Machine (JVM).Rather than a physical machine, the virtual machine is a program that interprets Java bytecode. This is one of Java’s primary advantages: Java bytecode can run on a variety of hardware platforms and operating systems. Java source code is compiled into Java bytecode and Java bytecode is interpreted by the JVM. Your Java code may use the code in the Java library. The JVM exe- cutes your code along with the code in the library.


compiled by
generates
executed by
Simple.java (Java source-code file)
Java Compiler
Simple.class (Java bytecode executable file)
JVM
Library Code

To execute a Java program is to run the program's bytecode. You can execute the bytecode on any platform with a JVM, which is an interpreter. It translates the individual instructions in the bytecode into the target machine language code one at a time rather than the whole pro- gram as a single unit. Each step is executed immediately after it is translated.

Java Variables and Data Types with EXAMPLE

What is a Variable?

A variable can be thought of as a container which holds value for you, during the life of a Java program. Every variable is assigned a **data type** which designates the type and quantity of value it can hold.

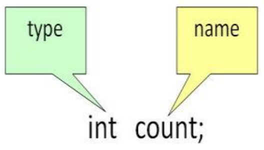
In order to use a variable in a program you to need to perform 2 steps

1. Variable Declaration
2. Variable Initialization

- Variable Declaration
- Variable Initialization
- Types of variables
- Data Types in Java
- Type Conversion & Type Casting

Variable Declaration:

To declare a variable, you must specify the data type & give the variable a unique name.



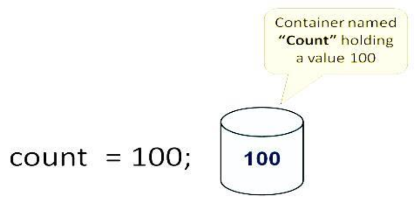
Examples of other Valid Declarations are

```
int a,b,c;

float pi;
```

```
double d;

char a;
```

Variable Initialization:

To initialize a variable, you must assign it a valid value.



Example of other Valid Initializations are

```
pi =3.14f;

do =20.22d;

a='v';
```

You can combine variable declaration and initialization.

```
int count = 100;
```

Example :

```
int a=2,b=4,c=6;

float pi=3.14f;

double do=20.22d;

char a='v';
```

Types of variables

In Java, there are three types of variables:

1. Local Variables
2. Instance Variables
3. Static Variables

1) Local Variables

Local Variables are a variable that are declared inside the body of a method.

2) Instance Variables

Instance variables are defined without the STATIC keyword .They are defined Outside a method declaration. They are Object specific and are known as instance variables.

3) Static Variables

Static variables are initialized only once, at the start of the program execution. These variables should be initialized first, before the initialization of any instance variables.

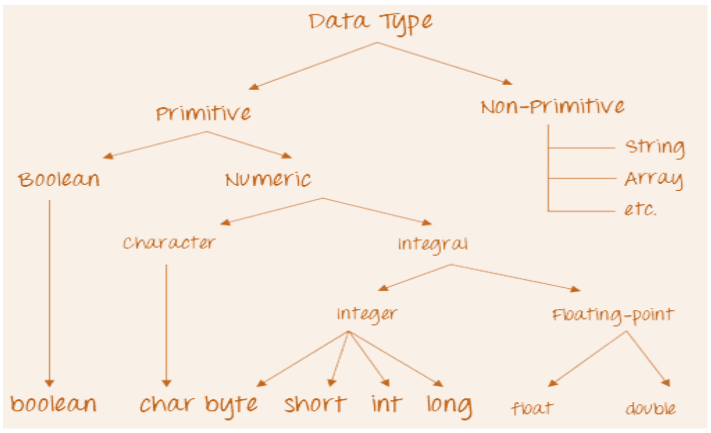
Example: Types of Variables in Java

```
class Mathexample {
   int data = 99; //instance variable
   static int a = 1; //static variable
   void method() {
      int b = 90; //local variable
   }
}
```

Data Types in Java

Data types classify the different values to be stored in the variable. In java, there are two types of data types:

1. Primitive Data Types
2. Non-primitive Data Types

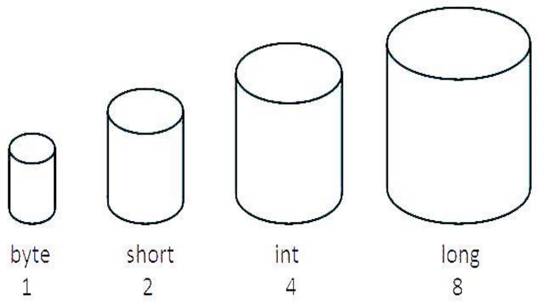


Primitive Data Types

Primitive Data Types are predefined and available within the Java language. Primitive values do not share state with other primitive values.

There are 8 primitive types: byte, short, int, long, char, float, double, and boolean **Integer data types**

```
byte (1 byte)
short (2 bytes)
int (4 bytes)
long (8 bytes)
```



Floating Data Type

float (4 bytes)

double (8 bytes)

Textual Data Type

char (2 bytes)

Logical

boolean (1 byte) (true/false)

Java Data Types		
Data Type	**Standard Default Value**	**Default size**
byte	0	1 byte
short	0	2 bytes
int	0	4 bytes
long	0L	8 bytes
float	0.0f	4 bytes
double	0.0d	8 bytes
boolean	false	1 bit
char	'\u0000'	2 bytes

Points to Remember:

- All numeric data types are signed(+/-).
- The size of data types remain the same on all platforms (standardized)
- char data type in Java is 2 bytes because it uses **UNICODE** character set. By virtue of it, Java supports internationalization. UNICODE is a character set which covers all known scripts and language in the world

Java Variable Type Conversion & Type Casting

A variable of one type can receive the value of another type. Here there are 2 cases -

Case 1) Variable of smaller capacity is be assigned to another variable of bigger capacity.

double d;   Int i=10;   d=i;

This process is Automatic, and non-explicit is known as ***Conversion***

Case 2) Variable of larger capacity is be assigned to another variable of smaller capacity

    double d =10;   int i;   i=(int) d;

In such cases, you have to explicitly specify the **type cast operator. This process is known as *Type Casting*.**

In case, you do not specify a type cast operator; the compiler gives an error. Since this rule is enforced by the compiler, it makes the programmer aware that the conversion he is about to do may cause some loss in data and prevents **accidental losses.**

Example: To Understand Type Casting

Step 1) Copy the following code into an editor.

```
class Demo {
 public static void main(String args[]) {
  byte x;
  int a = 270;
  double b = 128.128;
  System.out.println("int converted to byte");
  x = (byte) a;
  System.out.println("a and x " + a + " " + x);
  System.out.println("double converted to int");
  a = (int) b;
  System.out.println("b and a " + b + " " + a);
  System.out.println("\ndouble converted to byte");
  x = (byte)b;
  System.out.println("b and x " + b + " " + x);
 }
}
```

Step 2) Save, Compile & Run the code.

Output:

```
int converted to byte
a and x 270 14
double converted to int
b and a 128.128 128

double converted to byte
b and x 128.128 -128
```

Java Variables - Dynamic Initialization

Initialization is the process of providing value to a variable at declaration time. A variable is initialized once in its life time. Any attempt of setting a variable's value after its declaration is called assignment. To use a local variable you have to either initialize or assign it before the variable is first used. But for class members, the compulsion is not so strict. If you don't initialize them then compiler takes care of the initialization process and set class members to default values.

Java allows its programmers to initialize a variable at run time also. Initializing a variable at run time is called dynamic initialization. The following piece of code (DynamicInitializationDemo.java) demonstrates it.

```
/* DynamicInitializationDemo.java */
public class DynamicInitializationDemo
{
    public static void main(String[] args)
    {
        //dynSqrt will be initialized when Math.sqrt
        //will be executed at run time
        double dynSqrt = Math.sqrt (16);
        System.out.println("sqrt of 16 is : " + dynSqrt);
    }
}

OUTPUT
======
sqrt of 16 is : 4.0
```

Scope of a Variable in Java

The scope of a variable specifies the region of the source program where that variable is known, accessible and can be used. In Java, the declared variable has a definite scope. When a variable is defined within a class, its scope determines whether it can be used only within the defined class or outside of the class also.

Local variables can be used only within the block in which they are defined. The scope of instance variables covers the entire class, so they can be used by any of the methods within that class.

Variables must be declared before usage within a scope. Variables that are declared in an outer scope can also be used in an inner scope. The unique way in which Java checks for the scope of a given variable makes it possible to create a variable in an inner scope such that a definition of that variable hides its original value. Program makes this point clear.

Checking the scope of a variable.

```
class ScopeOfVariables

{

         static int var = 10;

         ScopeOfVariables()

        {
             // variable var from outer scope is accessed here

                 System.out.println("Variable accessed from outer scope =" +var);

          }
```

```
    public static void main(String args[])
    {
        int var 25;
          //variable var from current scope is accessed here
        ScopeOfVariables sc= new ScopeOfVariables();
         System.out.println("Variable accessed from current scope =" +var);
    }
}
```

The output of Program is given below:

```
Variable accessed from outer scope = 10

Variable accessed from current scope = 25
```

Java Operators

Operators are special symbols (characters) that carry out operations on operands (variables and values). For example, (+) is an operator that performs addition.
How to declare variables and assign values to variables. Now, you will learn to use operators to manipulate variables.

Assignment Operator

Assignment operators are used in Java to assign values to variables. For example,

```
int age;
age = 5;
```

The assignment operator assigns the value on its right to the variable on its left. Here, 5 is assigned to the variable age using = operator.

Example 1: Assignment Operator

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

            int number1, number2;

            // Assigning 5 to number1
            number1 = 5;
            System.out.println(number1);

            // Assigning value of variable number2 to number1
            number2 = number1;
            System.out.println(number2);
   }
}
```

When you run the program, the output will be:

```
5
5
```

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc.

Java Arithmetic Operators	
Operator	Meaning
+	Addition (also used for string concatenation)
-	Subtraction Operator
*	Multiplication Operator
/	Division Operator
%	Remainder Operator

Example 2: Arithmetic Operator

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

                double number1 = 12.5, number2 = 3.5, result;

                // Using addition operator
                result = number1 + number2;
                System.out.println("number1 + number2 = " + result);
                                // Using subtraction operator
                result = number1 - number2;
                System.out.println("number1 - number2 = " + result);

                // Using multiplication operator

                result = number1 * number2;

                System.out.println("number1 * number2 = " + result);

                // Using division operator

                result = number1 / number2;

                System.out.println("number1 / number2 = " + result);

                // Using remainder operator

                result = number1 % number2;

                System.out.println("number1 % number2 = " + result);

    }

}
```

When you run the program, the output will be:

```
number1 + number2 = 16.0
number1 - number2 = 9.0
number1 * number2 = 43.75
```

```
number1 / number2 = 3.5714285714285716
number1 % number2 = 2.0
```

In above example, all operands used are variables. However, it's not necessary at all. Operands used in arithmetic operators can be literals as well. For example,

```
result = number1 + 5.2;
result = 2.3 + 4.5;
number2 = number1 -2.9;
```

The `+` operator can also be used to concatenate two or more strings.

Example 3: Arithmetic Operator

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

            String start, middle, end, result;

            start = "Talk is cheap. ";
            middle = "Show me the code. ";
            end = "- Linus Torvalds";

            result = start + middle + end;
            System.out.println(result);
    }
}
```

When you run the program, the output will be:

Unary Operators

Unary operator performs operation on only one operand.

Operator	Meaning
+	Unary plus (not necessary to use since numbers are positive without using it)
-	Unary minus; inverts the sign of an expression
++	Increment operator; increments value by 1
- -	decrement operator; decrements value by 1
!	Logical complement operator; inverts the value of a boolean

Example 4: Unary Operator

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

            double number = 5.2, resultNumber;
            boolean flag = false;
            System.out.println("+number = " + +number);
            // number is equal to 5.2 here.

            System.out.println("-number = " + -number);
            // number is equal to 5.2 here.

        // ++number is equivalent to number = number + 1
            System.out.println("number = " + ++number);
        // number is equal to 6.2 here.

        // -- number is equivalent to number = number - 1
            System.out.println("number = " + --number);
        // number is equal to 5.2 here.

            System.out.println("!flag = " + !flag);
            // flag is still false.
    }
}
```

When you run the program, the output will be:

```
+number = 5.2
-number = -5.2
number = 6.2
number = 5.2
!flag = true
```

You can also use `++` and -- operator as both prefix and postfix in Java. The `++` operator increases value by 1 and `--` operator decreases value by 1.

```
int myInt = 5;
++myInt   // myInt becomes 6
myInt++   // myInt becomes 7
--myInt   // myInt becomes 6
myInt--   // myInt becomes 5
```

Simple enough till now. However, there is a crucial difference while using increment and decrement operator as prefix and postfix. Consider this example,

Example 5: Unary Operator

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

            double number = 5.2;

            System.out.println(number++);
            System.out.println(number);
```

```
                System.out.println(++number);
                System.out.println(number);
    }
}
```

When you run the program, the output will be:

```
5.2
6.2
7.2
7.2
```

When System.out.println(number++); statement is executed, the original value is evaluated first. The number is increased only after that. That's why you are getting 5.2 as an output. Then, when System.out.println(number); is executed, the increased value 6.2 is displayed.

However, when System.out.println(++number); is executed, number is increased by 1 first before it's printed on the screen. Similar is the case for decrement - - operator.

Equality and Relational Operators

The equality and relational operators determines the relationship between two operands. It checks if an operand is greater than, less than, equal to, not equal to and so on. Depending on the relationship, it results to either true or false.

Java Equality and Relational Operators

Operator	Description	Example
==	equal to	5 == 3 is evaluated to false
!=	not equal to	5 != 3 is evaluated to true
>	greater than	5 > 3 is evaluated to true
<	less than	5 < 3 is evaluated to false
>=	greater than or equal to	5 >= 5 is evaluated to true
<=	less then or equal to	5 <= 5 is evaluated to true

Equality and relational operators are used in decision making and loops (which will be discussed later). For now, check this simple example.

Example 6: Equality and Relational Operators

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

                int number1 = 5, number2 = 6;

                if (number1 > number2)
                {
                                System.out.println("number1 is greater than number2.");
                }
                else
                {
                                System.out.println("number2 is greater than number1.");
                }
    }
}
```

When you run the program, the output will be:

```
number2 is greater than number1.
```

Here, we have used > operator to check if number1 is greater than number2 or not.

Since, number2 is greater than number1, the expression number1 > number2 is evaluated to false.

Hence, the block of code inside else is executed and the block of code inside if is skipped.

If you didn't understand the above code, don't worry. You will learn it in detail in *Java if...else*article.

For now, just remember that the equality and relational operators compares two operands and is evaluated to either true or false.

In addition to relational operators, there is also a type comparison operator instanceof which compares an object to a specified type. For example,

instanceof Operator

Here's an example of instanceof operator.

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

            String test = "asdf";
            boolean result;

            result = test instanceof String;
            System.out.println(result);
   }
}
```

When you run the program, the output will be true. It's because test is the instance of String class.

You will learn more about instanceof operator works once you understand *Java Classes and Objects*.

Logical Operators

The logical operators || (conditional-OR) and && (conditional-AND) operates on boolean expressions. Here's how they work.

Java Logical Operators		
Operator	Description	Example
||	conditional-OR; true if either of the boolean expression is true	false || true is evaluated to true
&&	conditional-AND; true if all boolean expressions are true	false && true is evaluated to false

Example 8: Logical Operators

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

            int number1 = 1, number2 = 2, number3 = 9;
            boolean result;

            // At least one expression needs to be true for result to be true
            result = (number1 > number2) || (number3 > number1);
            // result will be true because (number1 > number2) is true
            System.out.println(result);

            // All expression must be true from result to be true
            result = (number1 > number2) && (number3 > number1);
            // result will be false because    (number3 > number1) is false
            System.out.println(result);
    }
}
```

When you run the program, the output will be:

```
true
false
```

Logical operators are used in decision making and looping.

Ternary Operator

The conditional operator or ternary operator ?: is shorthand for if-then-else statement. The syntax of conditional operator is:

```
variable = Expression ? expression1 : expression2
```

Here's how it works.

- If the Expression is true, expression1 is assigned to variable.
- If the Expression is false, expression2 is assigned to variable.

Example 9: Ternary Operator

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

            int februaryDays = 29;
            String result;

            result =  (februaryDays == 28) ? "Not a leap year" : "Leap year";
            System.out.println(result);
    }
}
```

When you run the program, the output will be:

```
Leap year
```

To learn more, visit Java ternary operator.

Bitwise and Bit Shift Operators

To perform bitwise and bit shift operators in Java, these operators are used.

Java Bitwise and Bit Shift Operators	
Operator	Description

~	Bitwise Complement
<<	Left Shift
>>	Right Shift
>>>	Unsigned Right Shift
&	Bitwise AND
^	Bitwise exclusive OR
|	Bitwise inclusive OR

These operators are not commonly used. Visit this page to learn more about *bitwise and bit shift operators*.

More Assignment Operators

We have only discussed about one assignment operator = in the beginning of the article. Except this operator, there are quite a few assignment operators that helps us to write cleaner code.

Java Assignment Operators		
Operator	Example	Equivalent to
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x / 5
<<=	x <<= 5	x = x << 5
>>=	x >>= 5	x = x >> 5
&=	x &= 5	x = x & 5
^=	x ^= 5	x = x ^ 5
|=	x |= 5	x = x | 5

Java Input

There are several ways to get input from the user in Java. You will learn to get input by using Scanner object in this article.

For that, you need to import Scanner class using:

```
import java.util.Scanner;
```

Learn more about *Java import*

Then, we will create an object of Scanner class which will be used to get input from the user.

```
Scanner input = new Scanner(System.in);
int number = input.nextInt();
```

Example 5: Get Integer Input From the User

```
import java.util.Scanner;

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

            Scanner input = new Scanner(System.in);

            System.out.print("Enter an integer: ");
            int number = input.nextInt();
            System.out.println("You entered " + number);
    }
}
```

When you run the program, the output will be:

```
Enter an integer: 23
You entered 23
```

Here, input object of Scanner class is created. Then, the nextInt() method of the Scannerclass is used to get integer input from the user.

To get long, float, double and String input from the user, you can use nextLong(), nextFloat(), nextDouble() and next() methods respectively.

Example 6: Get float, double and String Input

```
import java.util.Scanner;

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

                Scanner input = new Scanner(System.in);

                // Getting float input
                System.out.print("Enter float: ");
                float myFloat = input.nextFloat();
                System.out.println("Float entered = " + myFloat);

                // Getting double input
                System.out.print("Enter double: ");
                double myDouble = input.nextDouble();
                System.out.println("Double entered = " + myDouble);

                // Getting String input
                System.out.print("Enter text: ");
                String myString = input.next();
                System.out.println("Text entered = " + myString);
    }
}
```

When you run the program, the output will be:

```
Enter float: 2.343
Float entered = 2.343
Enter double: -23.4
Double entered = -23.4
Enter text: Hey!
Text entered = Hey!
```

Commonly Used Methods of the java.lang.Math class

The operators +, -, *, /, and % give us a way to add, subtract, multiply, divide, and "mod" values together in java, but having to implement some of the more sophisticated functions of mathematics (like the square root or sine functions) with only these operators would be challenging indeed!

As such, the java.lang.Math class provides us access to many of these functions. The table below lists some of the more common among these. (*There are more than just these in java.lang.Math, however* -- *one should consult the java API for the whole list.*)

Method	**Returns**
Math.sqrt(x)	x−−√x
Math.pow(x,y)	xyxy
Math.sin(x)	sinxsin⁡x
Math.cos(x)	cosxcos⁡x
Math.tan(x)	tanxtan⁡x
Math.asin(x)	sin−1xsin−1⁡x
Math.acos(x)	cos−1xcos−1⁡x
Math.atan(x)	tan−1xtan−1⁡x
Math.exp(x)	exex
Math.log(x)	lnxln⁡x
Math.log10(x)	log10xlog10⁡x
Math.round(x)	the closest integer to x, as athe closest integer to x, as a long
Math.abs(x)	|x||x|
Math.max(x,y)	The maximum of the two valuesThe maximum of the two values
Math.min(x,y)	The minimum of the two valuesThe minimum of the two values

Additionally, the java.lang.Math class provides two very frequently used constants that you might recognize:

Constant	**Type**	**Value**
Math.PI	double	3.14159265358979323846
Math.E	double	2.7182818284590452354

Math.abs

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

  int i1 = 27;
  int i2 = -45;
  double d1 = 84.6;
  double d2 = 0.45;
  System.out.println("Absolute value of i1: " + Math.abs(i1));

  System.out.println("Absolute value of i2: " + Math.abs(i2));

  System.out.println("Absolute value of d1: " + Math.abs(d1));

  System.out.println("Absolute value of d2: " + Math.abs(d2));

 }
}
```

Output:

```
Absolute value of i1: 27
Absolute value of i2: 45
Absolute value of d1: 84.6
Absolute value of d2: 0.45
```

Math.round

```
public class Mathexample {
 public static void main(String args[]) {
  double d1 = 84.6;
  double d2 = 0.45;
  System.out.println("Round off for d1: " + Math.round(d1));

  System.out.println("Round off for d2: " + Math.round(d2));
 }
}
```

Output:

```
Round off for d1: 85
Round off for d2: 0
```

Math.ceil & Math.floor

```
public class Mathexample {
 public static void main(String args[]) {
  double d1 = 84.6;
  double d2 = 0.45;
  System.out.println("Ceiling of '" + d1 + "' = " + Math.ceil(d1));

  System.out.println("Floor of '" + d1 + "' = " + Math.floor(d1));

  System.out.println("Ceiling of '" + d2 + "' = " + Math.ceil(d2));

  System.out.println("Floor of '" + d2 + "' = " + Math.floor(d2));

 }
}
```

Output:

```
Ceiling of '84.6' = 85.0
Floor of '84.6' = 84.0
Ceiling of '0.45' = 1.0
Floor of '0.45' = 0.0
```

Math.min

```
public class Mathexample {
 public static void main(String args[]) {
  int i1 = 27;
  int i2 = -45;
  double d1 = 84.6;
  double d2 = 0.45;
  System.out.println("Minimum out of '" + i1 + "' and '" + i2 + "' = " + Math.min(i1, i2));

  System.out.println("Maximum out of '" + i1 + "' and '" + i2 + "' = " + Math.max(i1, i2));

  System.out.println("Minimum out of '" + d1 + "' and '" + d2 + "' = " + Math.min(d1, d2));

  System.out.println("Maximum out of '" + d1 + "' and '" + d2 + "' = " + Math.max(d1, d2));

 }
}
```

Output:

```
Minimum out of '27' and '-45' = -45
Maximum out of '27' and '-45' = 27
Minimum out of '84.6' and '0.45' = 0.45
Maximum out of '84.6' and '0.45' = 84.6
```

B) Let us have a look at the table below that shows us the **Exponential and Logarithmic methods**and its description-

Method	**Description**	**Arguments**
exp	Returns the base of natural log (e) to the power of argument	Double
Log	Returns the natural log of the argument	double
Pow	Takes 2 arguments as input and returns the value of the first argument raised to the power of the second argument	Double
floor	Returns the largest integer that is less than or equal to the argument	Double
Sqrt	Returns the square root of the argument	Double

Below is the code implementation of the above methods: (The same variables are used as above)

```
public class Mathexample {
 public static void main(String args[]) {
  double d1 = 84.6;
  double d2 = 0.45;
  System.out.println("exp(" + d2 + ") = " + Math.exp(d2));

  System.out.println("log(" + d2 + ") = " + Math.log(d2));

  System.out.println("pow(5, 3) = " + Math.pow(5.0, 3.0));

  System.out.println("sqrt(16) = " + Math.sqrt(16));

 }
}
```

Output:

```
exp(0.45) = 1.568312185490169
log(0.45) = -0.7985076962177716
pow(5, 3) = 125.0
sqrt(16) = 4.0
```

C) Let us have a look at the table below that shows us the **Trigonometric methods** and its description-

Method	**Description**	**Arguments**
Sin	Returns the Sine of the specified argument	Double
Cos	Returns the Cosine of the specified argument	double
Tan	Returns the Tangent of the specified argument	Double
Atan2	Converts rectangular coordinates (x, y) to polar(r, theta) and returns theta	Double
toDegrees	Converts the arguments to degrees	Double
Sqrt	Returns the square root of the argument	Double
toRadians	Converts the arguments to radians	Double

Default Arguments are in Radians

Below is the code implementation:

```
public class Mathexample {
 public static void main(String args[]) {
  double angle_30 = 30.0;
  double radian_30 = Math.toRadians(angle_30);

  System.out.println("sin(30) = " + Math.sin(radian_30));

  System.out.println("cos(30) = " + Math.cos(radian_30));

  System.out.println("tan(30) = " + Math.tan(radian_30));

  System.out.println("Theta = " + Math.atan2(4, 2));

 }
}
```

Output:

```
sin(30) = 0.49999999999999994
```

```
cos(30) = 0.8660254037844387
tan(30) = 0.5773502691896257
Theta = 1.1071487177940904
```

Example: Using Java Math.Random
Now, if we want 10random numbers generated java but in the range of 0.0 to 1.0, then we should make use of math.random()

You can use the following loop to generate them-

```
public class DemoRandom{
  public static void main(String[] args) {
    for(int xCount = 0; xCount< 10; xCount++){
      System.out.println(Math.random());
    }
  }
}
```

Bitwise Operators
The Java Bitwise Operators allow access and modification of a particular bit inside a section of the data. It can be applied to integer types and bytes, and cannot be applied to float and double.

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

Program for bitwise operator

```
public class bitwiseop {
 public static void main(String[] args) {
  //Variables Definition and Initialization
  int num1 = 30, num2 = 6, num3 =0;

  //Bitwise AND
  System.out.println("num1 & num2 = " + (num1 & num2));

  //Bitwise OR
  System.out.println("num1 | num2 = " + (num1 | num2) );

  //Bitwise XOR
  System.out.println("num1 ^ num2 = " + (num1 ^ num2) );

  //Binary Complement Operator
  System.out.println("~num1 = " + ~num1 );

  //Binary Left Shift Operator
  num3 = num1 <<  2;
  System.out.println("num1 << 1 = " + num3 );

  //Binary Right Shift Operator
  num3 = num1 >>  2;
  System.out.println("num1 >> 1  = " + num3 );

  //Shift right zero fill operator
  num3 = num1 >>> 2;
  System.out.println("num1 >>> 1 = " + num3 );

 }
}
```

Output

```
num1 & num2 = 6
num1 | num2 = 30
num1 ^ num2 = 24
~num1 = -31
num1 << 1 = 120
num1 >> 1  = 7
num1 >>> 1 = 7
```

Operator Precedence

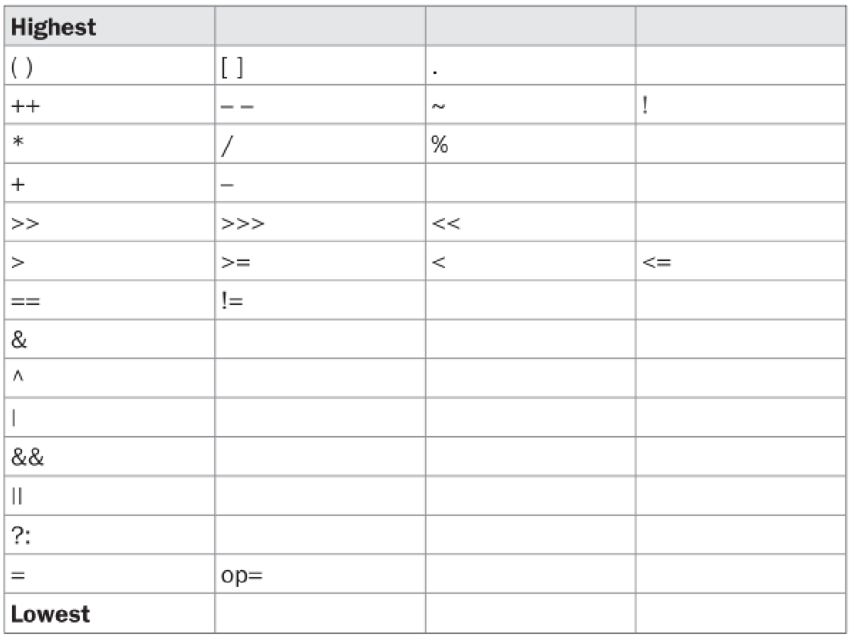
Shows the order of precedence for Java operators, from highest to lowest. Noticethat the first row shows items that you may not normally think of as operators: parentheses, square brackets, and the dot operator. Technically, these are called separators, but they act like operators in an expression. Parentheses are used to alter the precedence of an operation. the square brackets provide array indexing. The dot operator is used to dereference objects and will be discussed later in this book.

Using Parentheses

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

a >> b + 3

This expression first adds 3 to b and then shifts a right by that result. That is, this expression can be rewritten using redundant parentheses like this: a >> (b + 3)



Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

However, if you want to first shift a right by b positions and then add 3 to that result,

you will need to parenthesize the expression like this: (a >> b) + 3

In addition to altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression. For anyone reading your code, a complicated expression can be difficult to understand. Adding redundant but clarifying parentheses to complex expressions can help prevent confusion later. For example, which of the following expressions is easier to read?

a | 4 + c >> b & 7

(a | (((4 + c) >> b) & 7))

One other point: parentheses (redundant or not) do not degrade the performance of your program. Therefore, adding parentheses to reduce ambiguity does not negatively affect your program.

Control Statement

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration, and jump. Selection statements allow your program to choose different paths of execution based

upon the outcome of an expression or the state of a variable. Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

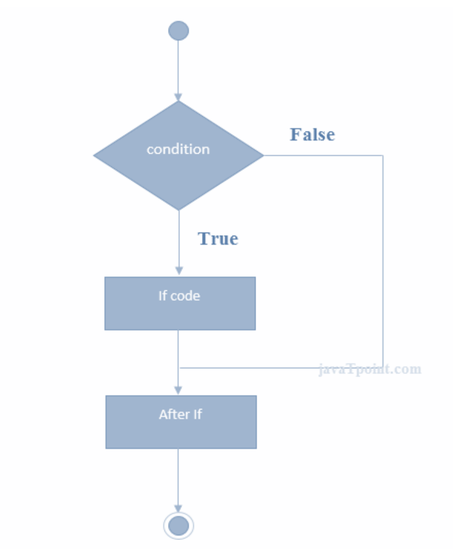
Java if Statement

The Java if statement tests the condition. It executes the if block if condition is true.

Syntax:

```
if(condition){
//code to be executed
```

}



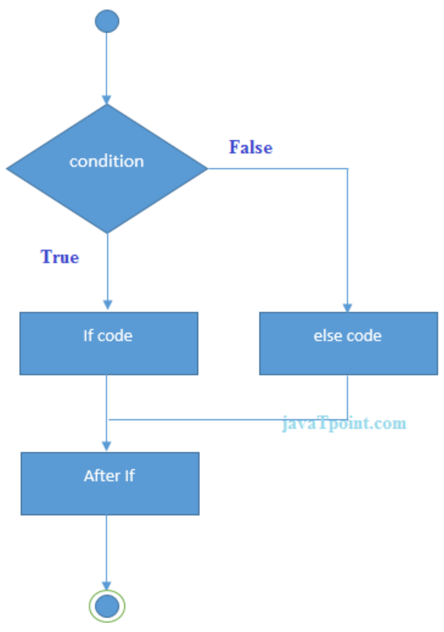
```
//Java Program to demonstate the use of if statement.
public class IfExample {
public static void main(String[] args) {
    //defining an 'age' variable
    int age=20;
    //checking the age
    if(age>18){
        System.out.print("Age is greater than 18");
    }
}
}  op :
Age is greater than 18
```

Java if-else Statement

The Java if-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.

Syntax:

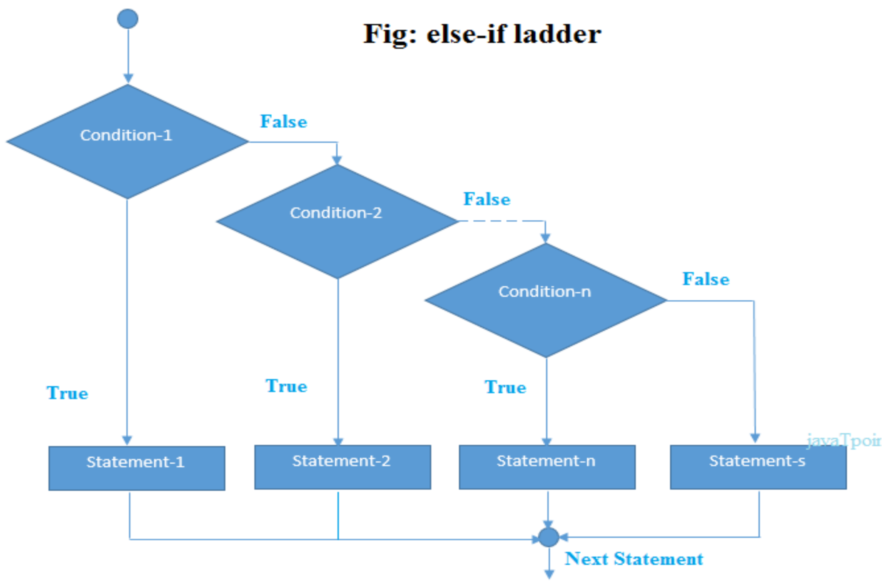
```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```



```
//A Java Program to demonstrate the use of if-
else statement.
//It is a program of odd and even number.
public class IfElseExample {
public static void main(String[] args) {
    //defining a variable
    int number=13;
    //Check if the number is divisible by 2 or not
    if(number%2==0){
        System.out.println("even number");
    }else{
        System.out.println("odd number");
    }
}
}  op : odd number
```

Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.



```
if(condition1){
//code to be executed if
condition1 is true
}else if(condition2){
//code to be executed if
condition2 is true
}
else if(condition3){
//code to be executed if
condition3 is true
}
...
else{
//code to be executed if all the
conditions are false
}
```

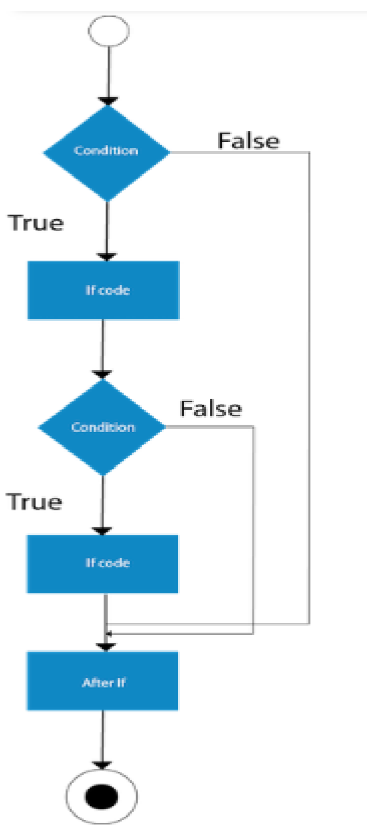
```
//Java Program to demonstrate the use of If else-if ladder.
//It is a program of grading system for fail, D grade, C grade, B grade, A grade
and A+.
public class IfElseIfExample {
public static void main(String[] args) {
    int marks=65;

    if(marks<50){
        System.out.println("fail");
    }
    else if(marks>=50 && marks<60){
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
}
} output  C grade
```

Java Nested if statement

The nested if statement represents the if block within another if block. Here, the inner if block condition executes only when outer if block condition is true.

Syntax:

```
if(condition){
  //code to be executed
    if(condition){
        //code to be executed
    }
 }
```



```
//Java Program to demonstrate the use of Nested If Statement.
public class JavaNestedIfExample {
public static void main(String[] args) {
    //Creating two variables for age and weight
    int age=20;
    int weight=80;
    //applying condition on age and weight
    if(age>=18){
        if(weight>50){
            System.out.println("You are eligible to donate
blood");
        }
    }
}  Output:

You are eligible to donate blood
```

Java Switch Statement

The Java switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.

In other words, the switch statement tests the equality of a variable against multiple values.

Points to Remember

- There can be one or N number of case values for a switch expression.
- The case value must be of switch expression type only. The case value must be literal or constant. It doesn't allow variables.
- The case values must be unique. In case of duplicate value, it renders compile-time error.
- The Java switch expression must be of byte, short, int, long (with its Wrapper type), enums and string.

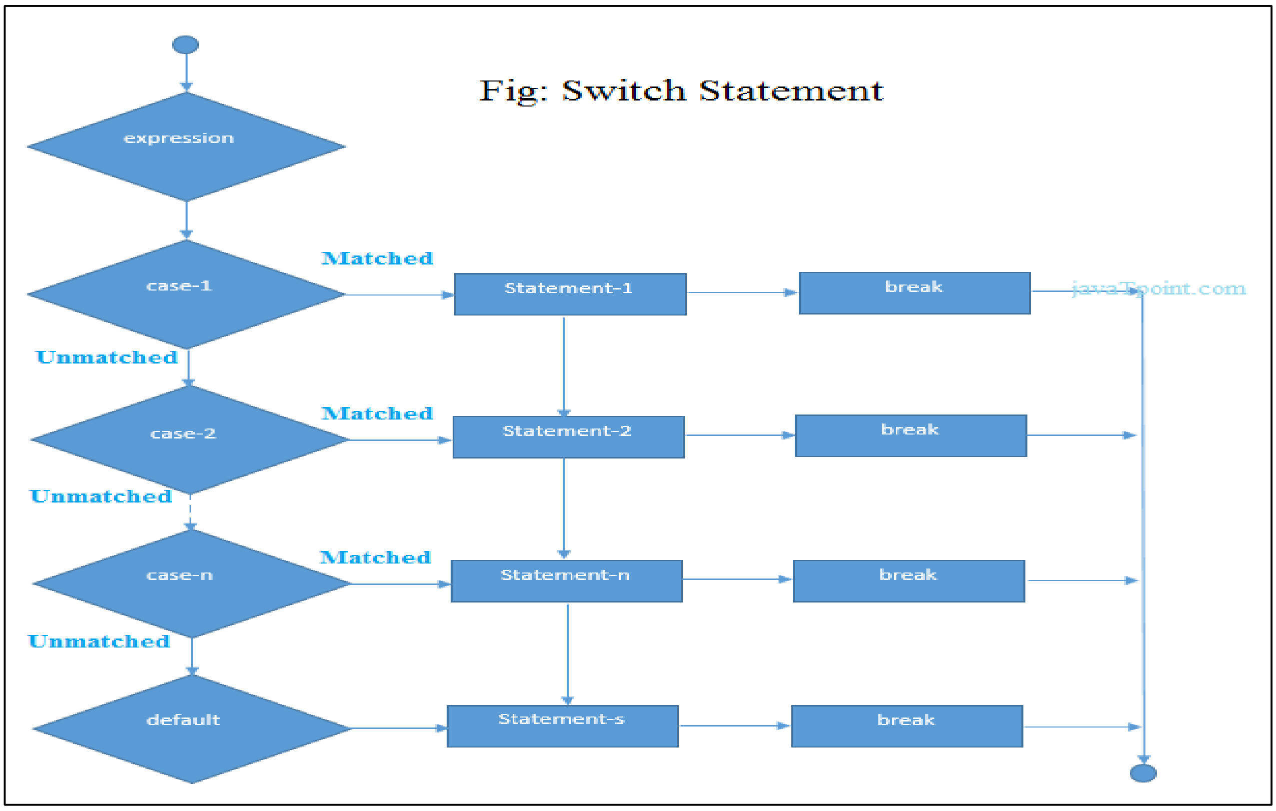
Each case statement can have a break statement which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.

The case value can have a default label which is optional.

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......

default:
 code to be executed if all cases are not matched;
}
```

```
public class SwitchExample {
public static void main(String[] args) {
    //Declaring a variable for switch
expression
    int number=20;
    //Switch expression
    switch(number){
    //Case statements
    case 10: System.out.println("10");
    break;
    case 20: System.out.println("20");
    break;
    case 30: System.out.println("30");
    break;
    //Default case statement
    default:System.out.println("Not in 10, 20
or 30");
    }
}
}
```

```
Output:

20
```



Java Switch Statement with String

```
//Java Program to demonstrate the use of Java Switch
//statement with String
public class SwitchStringExample {
public static void main(String[] args) {
   //Declaring String variable
   String levelString="Expert";
   int level=0;
```

```
    //Using String in Switch expression
    switch(levelString){
    //Using String Literal in Switch case
    case "Beginner": level=1;
    break;
    case "Intermediate": level=2;
    break;
    case "Expert": level=3;
    break;
    default: level=0;
    break;
    }
    System.out.println("Your Level is: "+level);
}
}   Output:Your Level is: 3
```

Java Nested Switch Statement

We can use switch statement inside other switch statement in Java. It is known as nested switch statement.

```
//Java Program to demonstrate the use of Java Nested Switch
public class NestedSwitchExample {
    public static void main(String args[])
      {
      //C - CSE, E - ECE, M - Mechanical
        char branch = 'C';
        int collegeYear = 4;
        switch( collegeYear )
        {
            case 1:
  System.out.println("English, Maths, Science");
                break;
            case 2:
                switch( branch )
                {
                    case 'C':
     System.out.println("Operating System, Java, Data Structure");
                        break;
                    case 'E':
          System.out.println("Micro processors, Logic switching theory");
                        break;
                    case 'M':
         System.out.println("Drawing, Manufacturing Machines");
                        break;
                }
                break;
            case 3:
                switch( branch )
                {
                    case 'C':
       System.out.println("Computer Organization, MultiMedia");
                        break;
                    case 'E':
      System.out.println("Fundamentals of Logic Design, Microelectronics");
                        break;
                    case 'M':
     System.out.println("Internal Combustion Engines, Mechanical Vibration");
```

```
                        break;
                  }
                  break;
              case 4:
                  switch( branch )
                  {
                      case 'C':
      System.out.println("Data Communication and Networks, MultiMedia");
                        break;
                      case 'E':
  System.out.println("Embedded System, Image Processing");
                        break;
                      case 'M':
   System.out.println("Production Technology, Thermal Engineering");
                        break;
                  }
                  break;
          }
      }
}   op Data Communication and Networks, MultiMedia
```

Looping statement

are the statements execute one or more statement repeatedly several number of times. In java programming language there are three types of loops; while, for and do-while.

Why use loop ?

When you need to execute a block of code several number of times then you need to use looping concept in Java language.

Advantage with looping statement

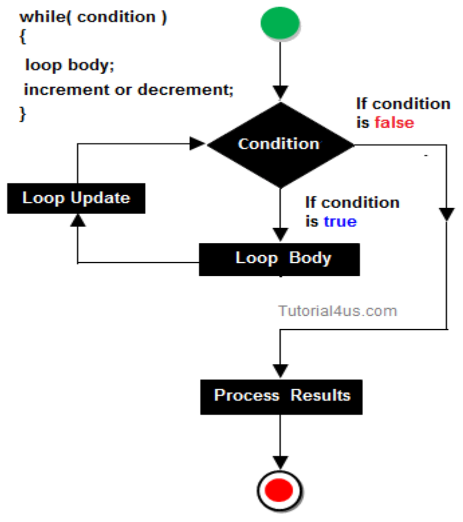
- Reduce length of Code
- Take less memory space.
- Burden on the developer is reducing.
- Time consuming process to execute the program is reduced.

Difference between conditional and looping statement

Conditional statement executes only once in the program where as looping statements executes repeatedly several number of time.

While loop

In **while loop** first check the condition if condition is true then control goes inside the loop body otherwise goes outside of the body. while loop will be repeats in clock wise direction.



```
while( condition )
{
 loop body;
 increment or decrement;
}
```

Syntax

```
while(condition)
{
 Statement(s)
 Increment / decrements (++ or --);
}
```

Example while loop

```
class  whileDemo
{
 public static void main(String args[])
 {
 int i=0;
 while(i<5)
 {
  System.out.println(+i);
  i++;
 }
```

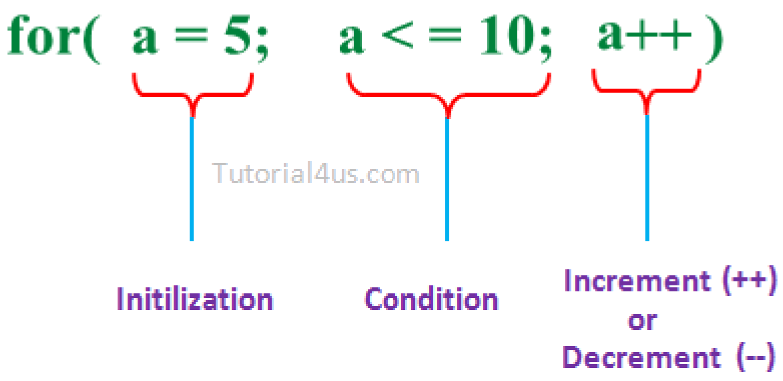
```
Output
0
2
3
```

for loop

for loop is a statement which allows code to be repeatedly executed. For loop contains 3 parts Initialization, Condition and Increment or Decrements

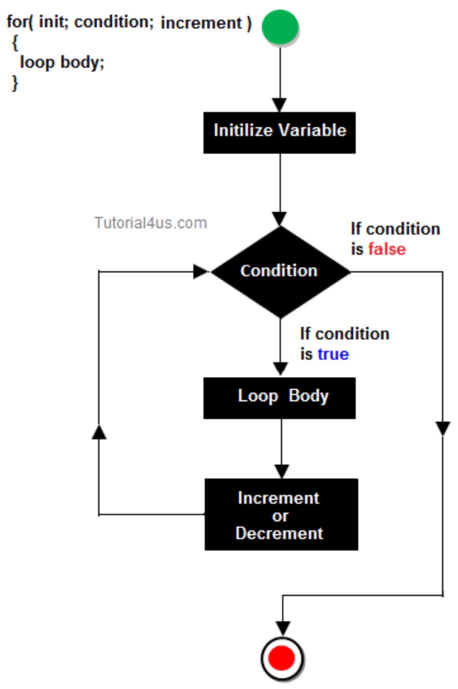
Syntax

```
for ( initialization; condition; increment )

{

 statement(s);

}
```

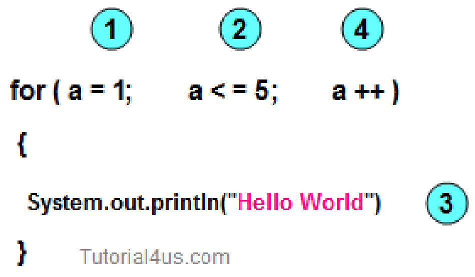


- **Initialization:** This step is execute first and this is execute only once when we are entering into the loop first time. This step is allow to declare and initialize any loop control variables.
- **Condition:** This is next step after initialization step, if it is true, the body of the loop is executed, if it is false then the body of the loop does not execute and flow of control goes outside of the for loop.
- **Increment or Decrements:** After completion of Initialization and Condition steps loop body code is executed and then Increment or Decrements steps is execute. This statement allows to update any loop control variables.

Flow Diagram



Control flow of for loop



- First initialize the variable
- In second step check condition
- In third step control goes inside loop body and execute.
- At last increase the value of variable
- Same process is repeat until condition not false.

Improve your looping conceptFor Loop

Display any message exactly 5 times.

Example of for loop

```
class  Hello
{
public static void main(String args[])
{
int i;
for (i=0: i<5; i++)
{
System.out.println("Hello Friends !");
}
}
}
```

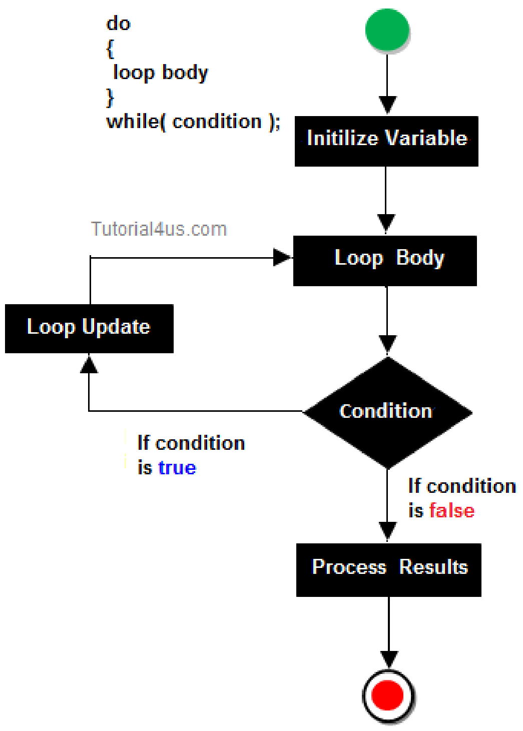
Output

```
Hello Friends !
Hello Friends !
Hello Friends !
Hello Friends !
Hello Friends !
```

do-while

A **do-while** loop is similar to a while loop, except that a do-while loop is execute at least one time.

A do while loop is a control flow statement that executes a block of code at least once, and then repeatedly executes the block, or not, depending on a given condition at the end of the block (in while).



When use do..while loop

when we need to repeat the statement block at least one time then use do-while loop. In do-while loop post-checking process will be occur, that is after execution of the statement block condition part will be executed.

Syntax

```
do
{
Statement(s)

increment/decrement (++ or --)
}while();
```

In below example you can see in this program i=20 and we check condition i is less than 10, that means condition is false but do..while loop execute onec and print Hello world ! at one time.

Example do..while loop

```
class  dowhileDemo
{
 public static void main(String args[])
 {
 int i=20;
do
 {
System.out.println("Hello world !");
 i++;
}
while(i<10);
}
}
```

Output

```
Hello world !
```

Example do..while loop

```
class  dowhileDemo
{
public static void main(String args[])
{
int i=0;
do
{
System.out.println(+i);
i++;
}
while(i<5);
}
}
```

Output

```
1
2
3
4
5
```

For-each loop in Java
Prerequisite: Decision making in Java

For-each is another array traversing technique like for loop, while loop, do-while loop introduced in Java5.

It starts with the keyword for like a normal for-loop.
Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
In the loop body, you can use the loop variable you created rather than using an indexed array element.
It’s commonly used to iterate over an array or a Collections class (eg, ArrayList)

Syntax:

```
for (type var : array)

{

   statements using var;

}
```

```
// Java program to illustrate
// for-each loop
class For_Each
{
   public static void main(String[] arg)
   {
      {
         int[] marks = { 125, 132, 95, 116, 110 };
```

```
            int highest_marks = maximum(marks);
            System.out.println("The highest score is " + highest_marks);
        }
    }
    public static int maximum(int[] numbers)
    {
        int maxSoFar = numbers[0];

        // for each loop
        for (int num : numbers)
        {
            if (num > maxSoFar)
            {
                maxSoFar = num;
            }
        }
    return maxSoFar;
    }
}
```

Output:

The highest score is 132

Limitations of for-each loop

For-each loops are not appropriate when you want to modify the array:

```
for (int num : marks)
{
    // only changes num, not the array element
    num = num*2;
}
```

For-each loops do not keep track of index. So we can not obtain array index using For-Each loop

```
for (int num : numbers)
{
    if (num == target)
    {
        return ???;   // do not know the index of num
    }
}
```

For-each only iterates forward over the array in single steps

```
// cannot be converted to a for-each loop
for (int i=numbers.length-1; i>0; i--)
{
      System.out.println(numbers[i]);
}
```

For-each cannot process two decision making statements at once

```
// cannot be easily converted to a for-each loop
for (int i=0; i<numbers.length; i++)
{
    if (numbers[i] == arr[i])
    { ...
    }
}
```

Questions

1) .State and explain any four feature of java
2) Why java becomes platform independent language ?Explain
3) What is JVM ?What is byte Code ?
4) What is byte code ?Explain any two tools available in JDK.
5) How JAVA is different than other programming language ?
6) Explain Classes and Object in brief with example (8 marks)
7) Define a class item having data member code and price .Accept data for one object and display it (4 Mark)
8) Define a class and object .Write syntax to create class and its object with and an example.
9) What is use of new operator? Is it necessary to be used whenever object of class is created? Why? (4 marks)
10) Write a program to create a class account having variable accno,accname, and balance .Define deposite() and withdraw() method .Create one object of class and perform operation.
11) State and Explain Scope of the variable with an example.
12) What is the scope of the variable ?Give example of class variable, instance variable and instance variable.
13) What is type casting Explain it type with proper syntax and example
14) Write all primitive data types available in java with their storage size in bytes
15) Describe any two relational and any two logical operator in java with simple example
16) Explain any two relational operator in java with example
17) ‘?:” What is this operator is called ?Explain with suitable example .
18) Explain any two bitwise operator with example
19) Explain following bitwise operator with an example a) left shift b) Right shift operator .
20) Write a program to generate Fibonacci series 1 1 2 3 5 8 13 21 34 55 89
21) Write a general syntax of any two decision statement and also give its example ?
22) Explain break and continue statement with example .
23) State syntax and describe working of for each version of for loop with one example
24) Illustrate with example the use of switch case statement
25) Describe break and continue statement with example+