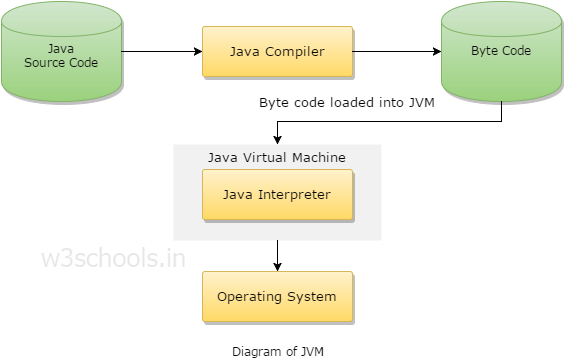
**UNIT 1**

**INTRODUCTION TO JAVA:**

* Java is a programming language created by James Gosling from Sun Microsystems (Sun) in 1991.
* The target of Java is to write a program once and then run this program on multiple operating systems.
* The first publicly available version of Java (Java 1.0) was released in 1995.
* Over time new enhanced versions of Java have been released. The current version of Java is Java 1.8 which is also known as *Java 8*.
* Java is defined by a specification and consists of a programming language, a compiler, core libraries and a runtime (Java virtual machine).
* The Java runtime allows software developers to write program code in other languages than the Java programming language which still runs on the Java virtual machine.
* The *Java platform* is usually associated with the *Java virtual machine* and the *Java core libraries*.

[**JAVA VIRTUAL MACHINE**](http://www.vogella.com/tutorials/JavaIntroduction/article.html#java-virtual-machine) **(JVM):**

* The Java virtual machine (JVM) is a software implementation of a computer that executes programs like a real machine.
* The Java virtual machine is written specifically for a specific operating system, e.g., for Linux a special implementation is required as well as for Windows.



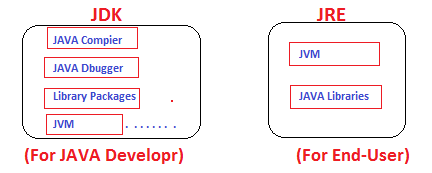
* Java programs are compiled by the Java compiler into *bytecode*. The Java virtual machine interprets this *bytecode* and executes the Java program.

[**CLASSPATH**](http://www.vogella.com/tutorials/JavaIntroduction/article.html#classpath)

* The *classpath* defines where the Java compiler and Java runtime look for .class files to load. These instructions can be used in the Java program.
* For example, if you want to use an external Java library you have to add this library to your classpath to use it in your program.

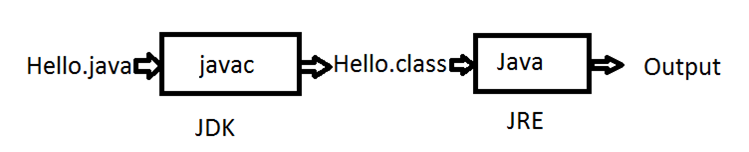
[**JAVA RUNTIME ENVIRONMENT (JRE) VS. JAVA DEVELOPMENT KIT**](http://www.vogella.com/tutorials/JavaIntroduction/article.html#java-runtime-environment-vs-java-development-kit) **(JDK)**

* A Java distribution typically comes in two flavors, the *Java Runtime Environment* (JRE) and the *Java Development Kit* (JDK).
* The JRE consists of the JVM and the Java class libraries. Those contain the necessary functionality to start Java programs.
* The JDK additionally contains the development tools necessary to create Java programs. The JDK therefore consists of a Java compiler, the Java virtual machine and the Java class libraries.



[**DEVELOPMENT PROCESS WITH JAVA**](http://www.vogella.com/tutorials/JavaIntroduction/article.html#development-process-with-java)

* Java source files are written as plain text documents.
* The programmer typically writes Java source code in an *Integrated Development Environment* (IDE) for programming.
* An IDE supports the programmer in the task of writing code, e.g., it provides auto-formating of the source code, highlighting of the important keywords, etc.
* At some point the programmer (or the IDE) calls the Java compiler ( javac ).
* The Java compiler creates the *bytecode* instructions.
* These instructions are stored in .class files and can be executed by the Java Virtual Machine.



[**GARBAGE COLLECTOR**](http://www.vogella.com/tutorials/JavaIntroduction/article.html#garbage-collector)

* The JVM automatically re-collects the memory which is not referred to by other objects.
* The Java *garbage collector* checks all object references and finds the objects which can be automatically released.
* While the garbage collector relieves the programmer from the need to explicitly manage memory, the programmer still need to ensure that he does not keep unneeded object references, otherwise the garbage collector cannot release the associated memory.

## Where it is used?

According to Sun, 3 billion devices run java. There are many devices where java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus etc.
2. Web Applications such as irctc.co.in, javatpoint.com etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games etc.

## Types of Java Applications

There are mainly 4 type of applications that can be created using java programming:

#### 1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

#### 2) Web Application

An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

#### 3) Enterprise Application

An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

#### 4) Mobile Application

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

# C++ vs. Java

There are many differences and similarities between C++ programming language and Java. A list of top differences between C++ and Java are given below:

|  |  |  |
| --- | --- | --- |
| **Index** | **C++** | **Java** |
| Platform-independent | C++ is platform-dependent. | Java is platform-independent. |
| Mainly used for | C++ is mainly used for system programming. | Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications. |
| Goto | C++ supports goto statement. | Java doesn't support goto statement. |
| Multiple inheritance | C++ supports multiple inheritance. | Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java. |
| Operator Overloading | C++ supports operator overloading. | Java doesn't support operator overloading. |
| Pointers | C++ supports pointers. You can write pointer program in C++. | Java supports pointer internally. But you can't write the pointer program in java. It means java has restricted pointer support in java. |
| Compiler and Interpreter | C++ uses compiler only. | Java uses compiler and interpreter both. |
| Call by Value and Call by reference | C++ supports both call by value and call by reference. | Java supports call by value only. There is no call by reference in java. |

# FEATURES OF JAVA

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

Java Features

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

Simple

According to Sun, Java language is simple because:

* Syntax is based on C++ (so easier for programmers to learn it after C++).
* Removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.
* No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

Object-oriented

|  |
| --- |
| * Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour. |
| * Object-oriented programming (OOPs) is a methodology that simplify 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  A platform is the hardware or software environment in which a program runs.  java is platform independent   * There are two types of platforms software-based and hardware-based. Java provides 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 e.g. 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 secured because:

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

how java is secured

* **Class loader:** adds security by separating the package for the classes of the local file system from those that are imported from network sources.
* **Byte code Verifier:** checks the code fragments for illegal code that can violate access right to objects.
* **Security Manager:** determines what resources a class can access such as reading and writing to the local disk.

Robust

* Robust simply means strong.
* Java uses strong memory management.
* There are lack of pointers that avoids security problem.
* There is automatic garbage collection in java.
* There is exception handling and type checking mechanism in java.

All these points makes java robust.

### Architecture-neutral

There is no implementation dependent feature e.g. size of primitive types is fixed.

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

### Portable

* We May carry java bytecode to any platform. And can execute in any platform.

### High-performance

|  |
| --- |
| * Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++) |

### Distributed

|  |
| --- |
| * We can create distributed applications in java. * We may access files by calling the methods from any machine on the internet. |

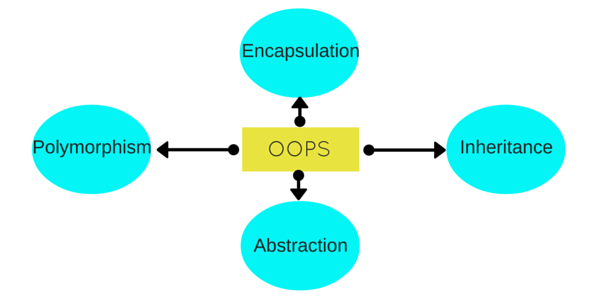
### Multi-threaded

* 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.

## OOPs (Object Oriented Programming System) Concept:

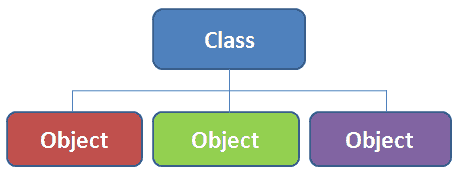
* **Object** means a real word entity such as pen, chair, table etc.
* **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects.
* It simplifies the software development and maintenance by providing some concepts:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation



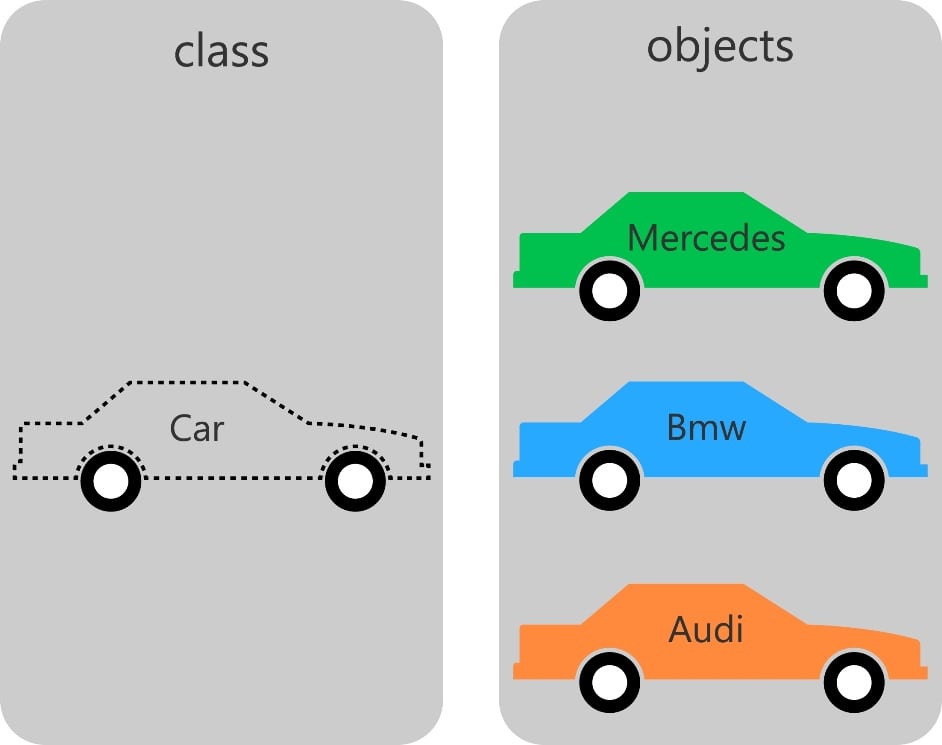
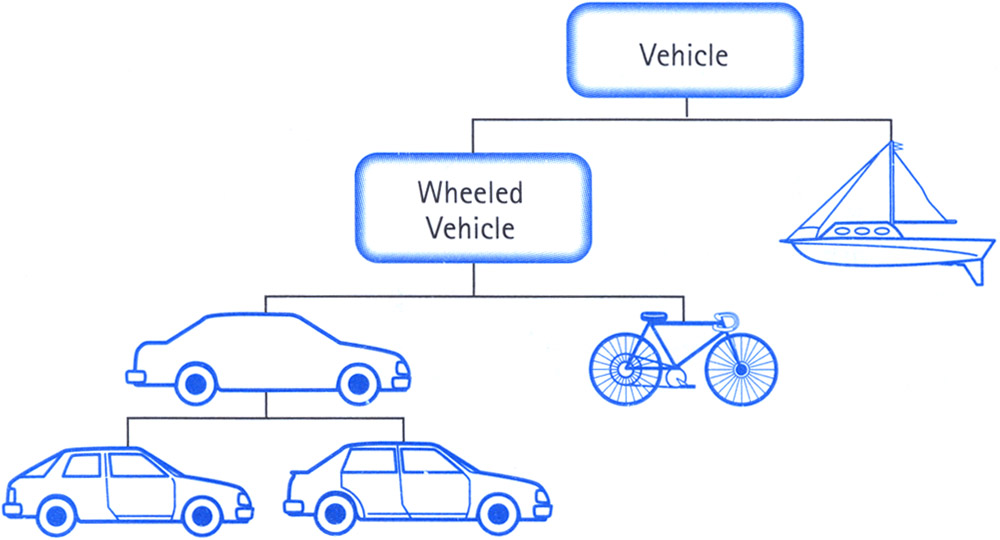
**Class**

## Collection of objects is called class. It is a logical entity.



## OBJECT

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.



#### Inheritance

* When one object acquires all the properties and behaviours of parent object i.e. known as inheritance.
* It provides code reusability.
* It is used to achieve runtime polymorphism.

#### Image result for inheritance in java

**POLYMORPHISM**

* When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.
* In java, we use method overloading and method overriding to achieve polymorphism.
* Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

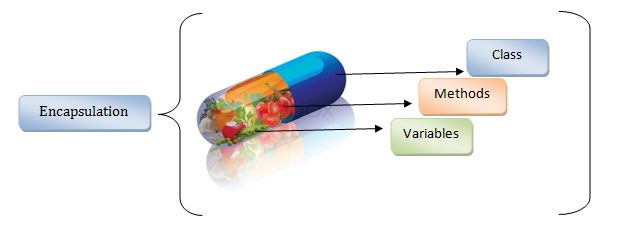
#### Related image

#### ABSTRACTION

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

#### ENCAPSULATION

* Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.
* A java class is the example of encapsulation.
* Java bean is the fully encapsulated class because all the data members are private here.

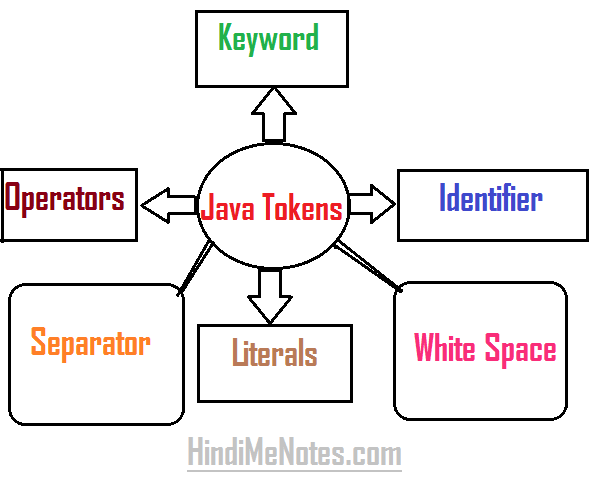


# JAVA TOKENS:

* A token is the smallest element of a program that is meaningful to the compiler.
* These tokens define the structure of the Java language.
* While submit a Java program to the Java compiler, the compiler scans the text and extracts individual tokens.

Java tokens can be broken into five categories:

1. Identifiers
2. Keywords
3. Literals
4. Operators
5. Separators



The Java compiler also recognizes and subsequently removes comments and whitespaces.

**Example:**

Public class Hello

{

Public static void main(String args[])

{

System.out.println(“**welcome in Java**”);   //print welcome in java

}

}

In above Example ,

* The source code contains tokens such as public, class, Hello, {, public, static, void, main, (, String, [], args, {, System, out, println, (, “welcome in 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.

## ****IDENTIFIERS:****

* Identifiers are tokens that represent names.
* These names can be assigned to **variables**, **methods**, and **classes** to uniquely identify them to the compiler.

Identifiers are,

* **Sequence of Uppercase letters(A to Z)**
* **Sequence of Lowercase letters (a to z)**
* **Numbers (0 to 9)**
* **Underscore (\_)**
* **Dollar Symbol ($)**

**RULES OF IDENTIFIERS JAVA**

1. Should be single word. That is spaces are not allowed.  
   Example: mangoprice is valid but mango price is not valid.
2. Should start with a letter (alphabet) or underscore or $ symbol.  
   Example: price, \_price and $price are valid identifiers.
3. Should not be a keyword of Java as keyword carries special meaning to the compiler.  
   Example: class or void etc.
4. Should not start with a digit but digit can be in the middle or at the end.  
   Example: 5mangoescost is not valid and mango5cost and mangocost5 are valid.
5. Length of an identifier in Java can be of 65,535 characters and all are significant.
6. Identifiers are case-sensitive. That is both mango and Mango are treated differently.
7. Can contain all uppercase letters or lowercase letters or a mixture.

**Valid and invalid Java identifiers.**

|  |  |
| --- | --- |
| **Valid** | **Invalid** |
| HelloWorld | Hello World (uses a space) |
| Hi\_JAVA | Hi JAVA! (uses a space and punctuation mark) |
| value3 | 3value(begins with a number) |
| Tall | short (this is a Java keyword) |
| $age | #age (does not begin with any other symbol except \_ $ ) |

## ****KEYWORDS****:

* **A keyword is just a word which has got a special meaning and purpose to the compiler.**
* For this reason, we cannot use them in our program to identify our own (to give a name) a class, variable or method.

###### List of **48 keywords**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [ABSTRACT](http://way2java.com/oops-concepts/abstract-classes/) | [ASSERT](http://way2java.com/java-versions-2/jdk-1-4-j2se-4-version/) | BOOLEAN | BREAK | BYTE |
| CASE | [CATCH](http://way2java.com/exceptions/exception-handling-try-%e2%80%93-catch/) | CHAR | CLASS | CONTINUE |
| DEFAULT | DO | DOUBLE | ELSE | [ENUM](http://way2java.com/java-versions-2/jdk-1-5-java-se-5-version/) |
| [EXTENDS](http://way2java.com/oops-concepts/java-extends/) | [FINAL](http://way2java.com/oops-concepts/many-meanings-of-final-keyword-%e2%80%93-in-and-outs/) | [FINALLY](http://way2java.com/exceptions/try-%e2%80%93-catch-%e2%80%93-finally/) | FLOAT | FOR |
| IF | [IMPLEMENTS](http://way2java.com/java-general/java-interface-multiple-inheritance/) | [IMPORT](http://way2java.com/oops-concepts/your-first-java-program/) | [INSTANCEOF](http://way2java.com/java-lang/instanceof-keyword/) | INT |
| [INTERFACE](http://way2java.com/java-general/java-interface-example/) | LONG | NATIVE | [NEW](http://way2java.com/oops-concepts/reference-variables-anonymous-objects/) | [PACKAGE](http://way2java.com/packages/predefined-packages-%e2%80%93-java-api/) |
| [PRIVATE](http://way2java.com/uncategorized/java-private-variable-accessibility-private-access-specifier/)+ | [PROTECTED](http://way2java.com/packages/access-specifiers-accessibility-permissions-restrictions/) | [PUBLIC](http://way2java.com/oops-concepts/access-specifiers-access-modifiers/) | RETURN | SHORT |
| [STATIC](http://way2java.com/oops-concepts/static-keyword-%e2%80%93-philosophy/) | [STRICTFP](http://way2java.com/oops-concepts/access-modifiers-%e2%80%93-meanings/) | [SUPER](http://way2java.com/oops-concepts/member-hiding-super-keyword/) | SWITCH | [SYNCHRONIZED](http://way2java.com/multithreading/synchronization-and-deadlock/) |
| [THIS](http://way2java.com/oops-concepts/using-this-keyword/) | [THROW](http://way2java.com/exceptions/creating-user-defined-excceptions-throw-keyword/) | [THROWS](http://way2java.com/exceptions/throwing-with-throws-alternative-to-try-catch/) | [TRANSIENT](http://way2java.com/oops-concepts/access-modifiers-%e2%80%93-meanings/) | [TRY](http://way2java.com/exceptions/exception-handling-try-%e2%80%93-catch/) |
| VOID | [VOLATILE](http://way2java.com/oops-concepts/access-modifiers-%e2%80%93-meanings/) | WHILE |  |  |

**1.**strictfp is added in JDK 1.2  
**2.** assert is added in JDK 1.5  
**3.**enum is added in JDK 1.5

The designers placed the two words, **goto** and **const** in reserved word list and not in keyword list. Designers placed in reserved list for the reason thinking that **goto** and **cosnt** may be required in future versions of Java.

Following is the list of **prohibited 3 literals** (given as values) which cannot be used in Java coding.

|  |  |  |
| --- | --- | --- |
| True | false | null |

## ****SEPARATORS:****

* Separators are used to inform the Java compiler of how things are grouped in the code.
* For example, items in a list are separated by commas much like lists of items in a sentence.
* The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements.

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Name** | **Purpose** |
| **;** | Semicolon | Terminates statements. |
| **,** | Comma | Separates consecutive identifiers in a variable  declaration.  Also used to chain statements together inside a **for**statement. |
| **{ }** | Braces | Used to contain the values of automatically initialized arrays.  Also used to define a block of code, for classes, methods, and local scopes. |
| **( )** | Parentheses | Used to contain lists of parameters in method definition and invocation.  Also used for defining precedence in expressions, containing expressions in control statements.  Also used for surrounding cast types. |
| **[ ]** | Brackets | Used to declare array types.  Also used when dereferencing array values. |
| **.** | Period | Used to separate package names from subpackages and classes Also used to separate a variable or method from a reference variable. |

## ****LITERALS:****

## Java Literals are syntactic representations of boolean, character, numeric, or string data.

## Literals provide a means of expressing specific values in your program.

| **Literal** | **type** |
| --- | --- |
| **1** | **int** |
| **3.14** | **double** (**1.** is a **double** too) |
| **true** | **boolean** |
| **'3'** | **char** (**'P'** and **'+'** are **char** too) |
| **"CMU ID"** | **String** |
| **null** | any reference type |

## For example, in the following statement, an integer variable named count is declared and assigned an integer value.

## Types of Java Literals

## Integer Literals

## Floating-point Literals

## Boolean Literals

## Character Literals

## String Literals

## COMMENTS AND WHITESPACES:

* The comments and whitespaces are removed by the Java compiler during the tokenization of the source code.
* White space consists of spaces, tabs, and linefeeds. All occurrences of spaces, tabs, or linefeeds are removed by the Java compiler, as are comments.
* Comments can be defined in three different ways, as shown in Table.

**Types of comments supported by Java.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Syntax** | **Usage** | **Example** |
| **Single-line** | // comment | All characters after the // up to the end of the line are ignored. | //This is a Single-line style comment. |
| **Multiline** | /\* comment \*/ | All characters between /\* and \*/ are ignored. | /\* This is a Multiline style comment. |
| **Documentation** | /\*\* comment \*/ | Same as /\* \*/, except that the comment can be used with the javadoc tool to create automatic documentation. | /\*\* This is a javadoc style comment. \*/ |

**OPERATORS:**

**Operator** in java is a symbol that is used to perform operations. For example: +, -, \*, / etc.

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.

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Category** | **Precedence** |
| Unary | Postfix | *expr*++ *expr*-- |
| Prefix | ++*expr* --*expr* +*expr* -*expr* ~ ! |
| Arithmetic | Multiplicative | \* / % |
| Additive | + - |
| Shift | Shift | << >> >>> |
| Relational | Comparison | < > <= >= instanceof |
| Equality | == != |
| Bitwise | bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| Logical | logical AND | && |
| logical OR | || |
| Ternary | Ternary | ? : |
| Assignment | Assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

**JAVA STATEMENTS:**

* A statement is an executable combination of tokens ending with semicolon (;) mark.
* Statements are usually executed in sequence in the order in which they appear.
* It is possible to control the flow of execution, if necessary, using special statements.
* The following image describes the types of statements.

|  |  |  |
| --- | --- | --- |
| **STATEMENTS** | **DESCRIPTION** | **REMARKS** |
| Empty Statement | * These do nothing and are used during program development as a place holder | Same as C and C++ |
| Labelled Statement | * Any Statement may begin with a label. * Such labels must not be Keywords * It is used as the arguments of jump statements. | Identical to C and C++ |
| Expression statement | * Most statements are expression statements. * Java has seven expression statements   1. **Assignment**   2. **Pre-increment**   3. **Pre-decrement**   4. **Post-Increment**   5. **Post-Decrement**   6. **Method call**   7. **Allocation Expression** | Same as C++ |
| Selection Statement | * These select one of several control flows * There are Three types of selection statements   1. **If statement**   2. **If-else statement**   3. **Switch statement** | Same as C and C++ |
| Iteration(Looping) statement | * Determines when looping will take place * There are three types of iteration statements,   1. **While loop**   2. **Do-while Loop**   3. **For –loop** | Same as C and C++ |
| Jump statement | * It passes control to beginning or end of the current block. * Four types of jump statements are,   1. **Break**   2. **Continue**   3. **Return**   4. **Throw** | C and C++ do not uses jump statement. |
| Synchronization statement | * used for handling issues with multithreading | Now available in C and C++ |
| Guarding Statement | * It is used for safe handling of code that may cause exceptions. * These statements use the Keywords **try**, **catch** and **finally**. | Same as C++ |

**CONSTANTS:**

* Constants in java refer to fixed values that do not change during the execution of a program.
* Java supports several types of constants

JAVA CONSTANTS

NUMERIC CONSTANTS

STRING CONSTANTS

SINGLE CHARACTER CONSTANTS

CHARACTER

CONSTANTS

INTEGER CONSTANTS

REAL CONSTANTS

**Integer constants**

* An integer constant refers to a sequence of digits. It includes,

1. Decimal integer
2. Octal integer
3. Hexadecimal integer
4. **Decimal Integer**

* It consists of a set of digits, 0 through 9, preceded by an optional minus sign.
* Valid examples of integer constants are, 123, -321, 0 ,654321
* Embedded spaces, commas & non-digit characters are not permitted b/w digits.

For eg: 15 750, 20,00 , $1000

1. **Octal integer**

* It consists of any combination of digits from the set 0 through 7, with a leading 0.

eg: 037 , 0, 0435, 0551

1. **Hexadecimal integer**

* A sequence of digits preceded by 0x or 0X is considered as hexadecimal integers.
* It may also include alphabets A through F or a through f.
* A letter A through F represents the numbers 10 through 15.

E.g. 0X2, 0X9F, 0xbcd 0x

**Real constants**

* Quantities are represented by numbers containing fractional part like 17.678.
* such numbers are called real/floating point constants.

E.g. 0.0083, -0.75 , 435.36

* It is possible that the number may not have digits before the decimal point or digit after the decimal points.

E.g. 215.0, .95 , -.71

* A real numbers may also be expressed in exponential or scientific notation.

(E.g.) the value 215.65 may be written as 2.156e2 in exponential notation. e2 means multiply by 10power2. The general form is,

Mantissa e exponent

**Character constants**

**Single Character constants**

* A single character constant or simply character constant contains a single character enclosed within a pair of single quote marks.
* The characters may be alphabets, digits, special characters and blank spaces.

Eg: ‘x’ , ‘A’ , ‘5’, ‘;’, ‘ ‘

* The character constant ‘5’ is not the same as the number 5. The last constant is a blank space.

**String constants**

* A string constant is a sequence of characters enclosed b/w double quotes. The characters may be alphabets, digits, special characters and blank spaces.

Eg: “hello java” “2013” “welcome” “#$%....!” “3+2” “x"  
  
**EXAMPLE PROGRAM:**  
 **class** Simple

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

**int** a=10;

**int** b=10;

**int** c=a+b;

System.out.println(c);

}}

## VARIABLE

**Variable** is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of "vary + able" that means its value can be changed.

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

variables in java

**E.g. int** data=50; //Here data is variable

### Types of Variable

There are three types of variables in java:

* local variable
* instance variable
* static variable

## Local Variables

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

### Example

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

public class Test {

public void pupAge() {

int age = 0;

age = age + 7;

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

}

public static void main(String args[]) {

Test test = new Test();

test.pupAge();

}

}

This will produce the following result −

### Output

Puppy age is: 7

## Instance Variables

* Instance variables are declared in a class, but outside a method, constructor or any block.
* When a space is allocated for an object in the heap, a slot for each instance variable value is created.
* Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
* Instance variables can be accessed directly by calling the variable name inside the class.

import java.io.\*;

public class Employee {

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

public String name;

// salary variable is visible in Employee class only.

private double salary;

// The name variable is assigned in the constructor.

public Employee (String empName) {

name = empName;

}

// The salary variable is assigned a value.

public void setSalary(double empSal) {

salary = empSal;

}

// This method prints the employee details.

public void printEmp() {

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

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

}

public static void main(String args[]) {

Employee empOne = new Employee("Ransika");

empOne.setSalary(1000);

empOne.printEmp();

}

}

This will produce the following result −

### Output

name : Ransika

salary :1000.0

## Class/Static Variables

* Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
* There would only be one copy of each class variable per class, regardless of how many objects are created from it.
* Static variables are created when the program starts and destroyed when the program stops.
* Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
* Static variables can be accessed by calling with the class name *ClassName.VariableName*.

### Example

import java.io.\*;

public class Employee {

// salary variable is a private static variable

private static double salary;

// DEPARTMENT is a constant

public static final String DEPARTMENT = "Development ";

public static void main(String args[]) {

salary = 1000;

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

}

}

This will produce the following result −

### Output

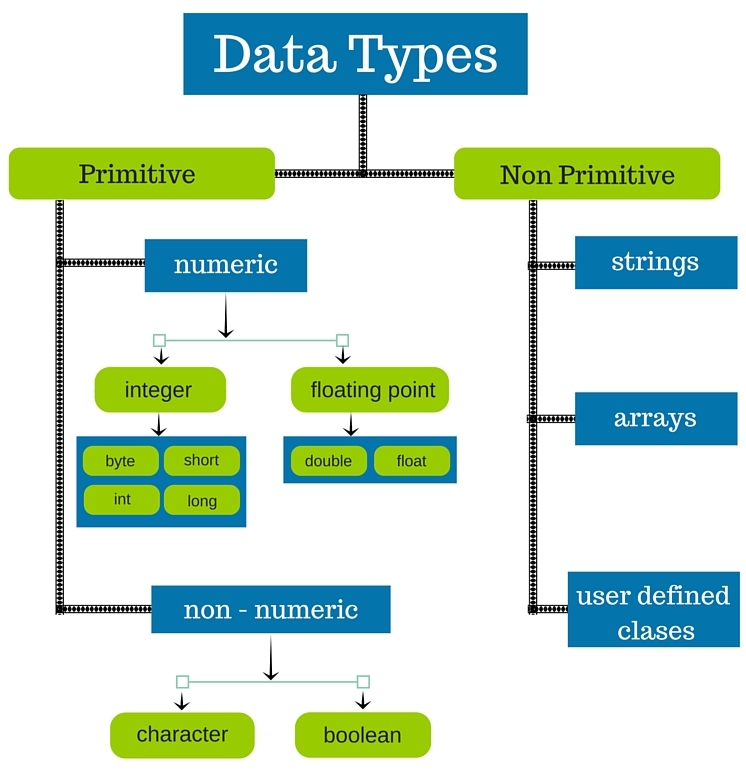
Development average salary:1000

**DATA TYPES:**

* Every variable in java has a data type.
* Data types specify the size and type of values that can be stored.
* Java language is rich in its data types.
* Primitive types are also called as *intrinsic* or *built-in* types.
* Derived data types are also known as reference types.
* Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

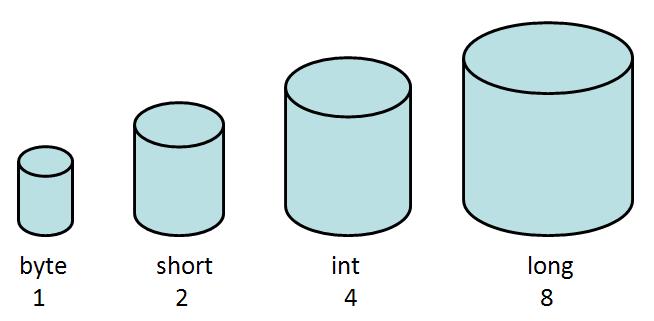
There are two data types available in Java −

* Primitive Data Types
* Reference/Non-Primitive Types



## Primitive Data Types

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

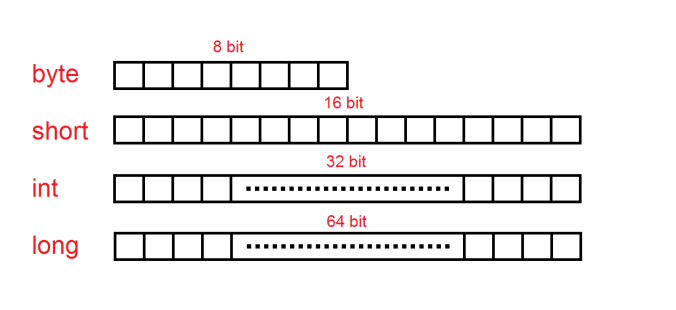
.

### byte

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

### short

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



### int

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

Example Program:

**class** Simple{

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

**int** a=10;

**float** f=a;

System.out.println(a);

System.out.println(f);

}}

Output:

10

10.0

### long

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

### float

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

### double

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

### boolean

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

### Image result for java data types

### char

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

## Reference Datatypes

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

### TYPE CASTING

Assigning a value of one type to a variable of another type is known as **Type Casting**.

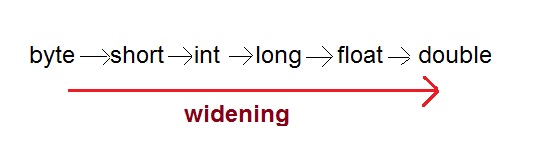
**Example :**

int x=10;

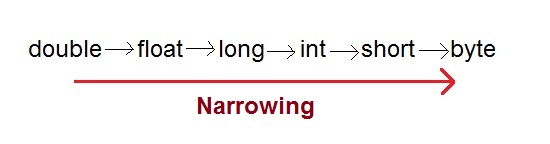
byte y=(byte)x;

In Java, type casting is classified into two types,

* Widening Casting(Implicit)



* Narrowing Casting(Explicitly done)



#### Widening or Automatic type conversion

Automatic Type casting take place when,

* the two types are compatible
* the target type is larger than the source type

**Example :**

public class Test

{

public static void main(String[] args)

{

int i = 100;

long l = i; //no explicit type casting required

float f = l; //no explicit type casting required

System.out.println("Int value "+i);

System.out.println("Long value "+l);

System.out.println("Float value "+f);

}

}

**Output :**

Int value 100  
Long value 100

Float value 100.0

#### Narrowing or Explicit type conversion

While assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

**Example :**

public class Test

{

public static void main(String[] args)

{

double d = 100.04;

long l = (long)d; //explicit type casting required

int i = (int)l; //explicit type casting required

System.out.println("Double value "+d);

System.out.println("Long value "+l);

System.out.println("Int value "+i);

}

}

**Output :**

Double value 100.04

Long value 100

Int value 100

**OPERATORS**

* **Operator** in java is a symbol that is used to perform operations.
* For example: +, -, \*, / etc.
* There are many types of operators in java which are given below:

1. Arithmetic Operator,
2. conditional Operator,
3. Increment and decrement Operator,
4. Relational Operator,
5. Bitwise Operator,
6. Logical Operator,
7. Assignment Operator and
8. Special Operator.

**Arithmetic Operator:**

Arithmetic operators are used to perform arithmetic operations on the operands. The following operators are known as Java arithmetic operators.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition | 10+20 = 30 |
| - | Subtraction | 20-10 = 10 |
| \* | Multiplication | 10\*20 = 200 |
| / | Division | 20/10 = 2 |
| % | Modulus (Remainder) | 20%10 = 0 |
| ++ | Increment | var a=10; a++; Now a = 11 |
| -- | Decrement | var a=10; a--; Now a = 9 |

**Example:**

**class** OperatorExample{

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

{

System.out.println(10\*10/5+3-1\*4/2);

}}

Output: 21

**Relational Operator:**

The Java relational operator compares the two operands. The comparison operators are as follows:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Is equal to | 10==20 = false |
| === | Identical (equal and of same type) | 10==20 = false |
| != | Not equal to | 10!=20 = true |
| !== | Not Identical | 20!==20 = false |
| > | Greater than | 20>10 = true |
| >= | Greater than or equal to | 20>=10 = true |
| < | Less than | 20<10 = false |
| <= | Less than or equal to | 20<=10 = false |

## Example

public class Test {

public static void main(String args[]) {

int a = 10,b = 20;

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

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

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

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

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

}}

## Output

a == b = false

a != b = true

a > b = false

a < b = true

b >= a = true

#### Bitwise operators

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

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Bitwise AND | (10==20 & 20==33) = false |
| | | Bitwise OR | (10==20 | 20==33) = false |
| ^ | Bitwise XOR | (10==20 ^ 20==33) = false |
| ~ | Bitwise NOT | (~10) = -10 |
| << | Bitwise Left Shift | (10<<2) = 40 |
| >> | Bitwise Right Shift | (10>>2) = 2 |
| >>> | Bitwise Right Shift with Zero | (10>>>2) = 2 |

## Logical Operators

The following operators are known as JavaScript logical operators.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Logical AND | (10==20 && 20==33) = false |
| || | Logical OR | (10==20 || 20==33) = false |
| ! | Logical Not | !(10==20) = true |

## Assignment Operators

The following operators are known as JavaScript assignment operators.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Assign | 10+10 = 20 |
| += | Add and assign | var a=10; a+=20; Now a = 30 |
| -= | Subtract and assign | var a=20; a-=10; Now a = 10 |
| \*= | Multiply and assign | var a=10; a\*=20; Now a = 200 |
| /= | Divide and assign | var a=10; a/=2; Now a = 5 |
| %= | Modulus and assign | var a=10; a%=2; Now a = 0 |

## Special Operators

The following operators are known as JavaScript special operators.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| (?:) | Conditional Operator returns value based on the condition. It is like if-else. |
| , | Comma Operator allows multiple expressions to be evaluated as single statement. |
| delete | Delete Operator deletes a property from the object. |
| In | In Operator checks if object has the given property |
| instanceof | checks if the object is an instance of given type |
| New | creates an instance (object) |
| typeof | checks the type of object. |
| Void | it discards the expression's return value. |
| Yield | checks what is returned in a generator by the generator's iterator. |

**Expressions**

* Expressions are a combination of variables, constants & operators arranged as per the syntax of the language.
* Java can handle any complex mathematical expressions.

|  |  |
| --- | --- |
| **Algebraic expression** | **Java expression** |
| Ab-c | A\*b-c |
| (m+n)(x+y) | (m+n)\*(x+y) |
| Ab/c | A\*b/c |
| 3x2+2x+1 | 3\*x\*x+2\*x+1 |

**Evaluation of expression:**

* Expressions are evaluated using an assignment statement of the form

**Variable=expression;**

* Variable is any valid java variable name.
* When the statement is encountered, the expression is evaluated first & the result then replaces the previous value of the variable on the left-hand side.

**Ex:**

X=a\*b-c;

Y=b/c\*a;

Z=a-b/c + d;

* The blank spaces around an operator are optional & it is added only to improve readability.
* The variables a, b, c, d must be defined before they are used in the expression.

**Example:**

**class** OperatorExample{

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

{

System.out.println(10\*10/5+3-1\*4/2);  //expression

}  
}

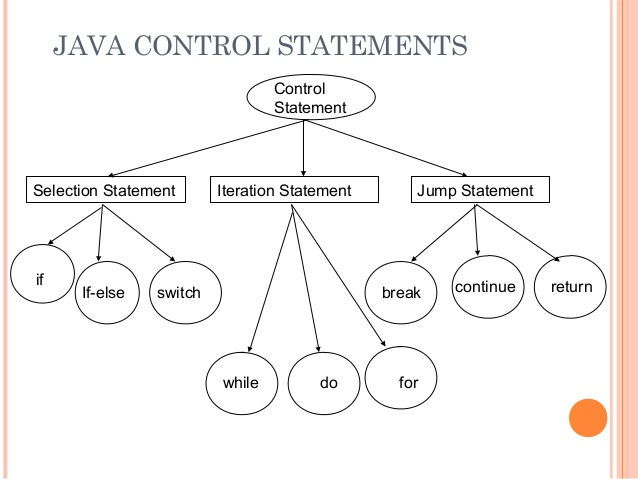
Output:

21

**CONTROL STRUCTURE:**

* Controlstatements are used to provide the **flow of execution** with condition.
* In java program, control structure is can divide in three parts:

1. Selection statement
2. Iteration statement
3. Jumps in statement



**Selection Statement:**

* Selection statement is also called as Decision making statements because it provides the decision making capabilities to the statements.
* In selection statement, there are two types:

1. if statement
2. Switch statement.

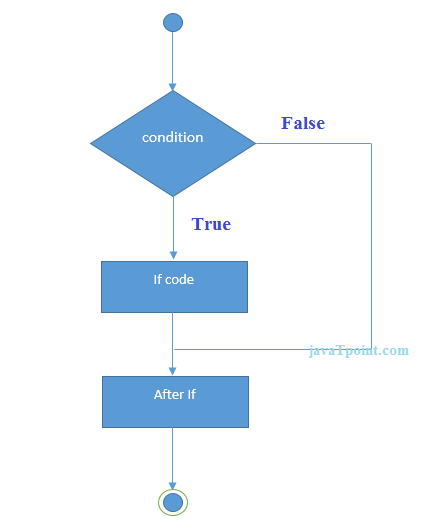
**Java if statement**

The Java if statement is used to test the condition. It checks boolean condition: true or false. There are various types of if statement in java.

* if statement
* if-else statement
* Nested if-else statement
* if-else-if ladder

## 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

}

**Example:**

**public** **class** IfExample {

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

**int** age=20;

**if**(age>18){

        System.out.print("Age is greater than 18");

    }

}

}

Output:

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

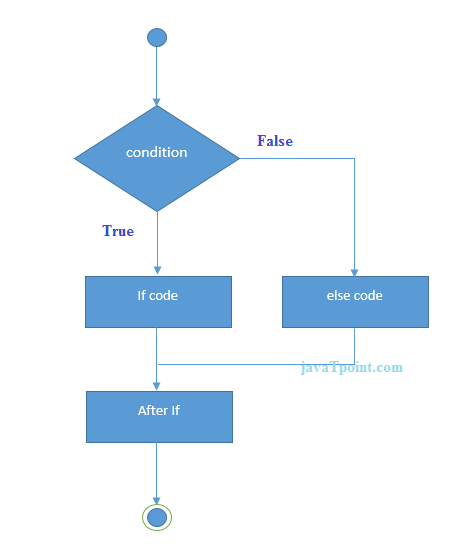
}

**Else**

{

//code if condition is false

}



**Example:**

**public** **class** IfElseExample {

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

**int** number=13;

**if**(number%2==0){

        System.out.println("even number");    }

**Else**

{

        System.out.println("odd number");

    }

}

}

**Output**:

odd number

## Java Nested if – else statement

The Nested if-else statement executes one if or else if statement inside another if or else if statement.

**Syntax:**if(Boolean\_expression 1)   
{  
 // Executes when the Boolean expression 1 is true if(Boolean\_expression 2)   
{  
 // Executes when the Boolean expression 2 is true  
 }  
}  
 **Example:**  
public class Test   
{  
 public static void main(String args[])   
{

## int x = 30;

## int y = 10;

## if( x == 30 )

## {

## if( y == 10 )

## {

## System.out.print("X = 30 and Y = 10");

## }

## }

## }

## }

## Image result for nested if else in java

## Java IF-else-if ladder Statement

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

**Syntax:**

**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

}

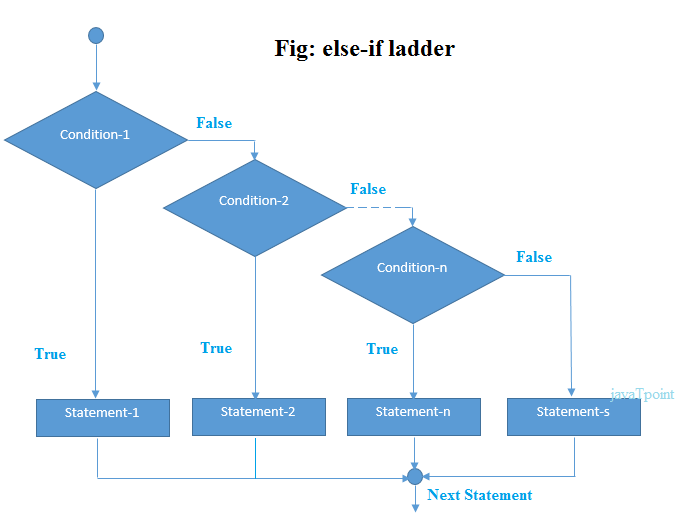
...

**Else**

{

//code to be executed if all the conditions are false

}



**Example:**

**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 Switch Statement**

The Java switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement.

**Syntax:**

**switch**(expression){

**case** value1:

 //code to be executed;

**break**;  //optional

**case** value2:

 //code to be executed;

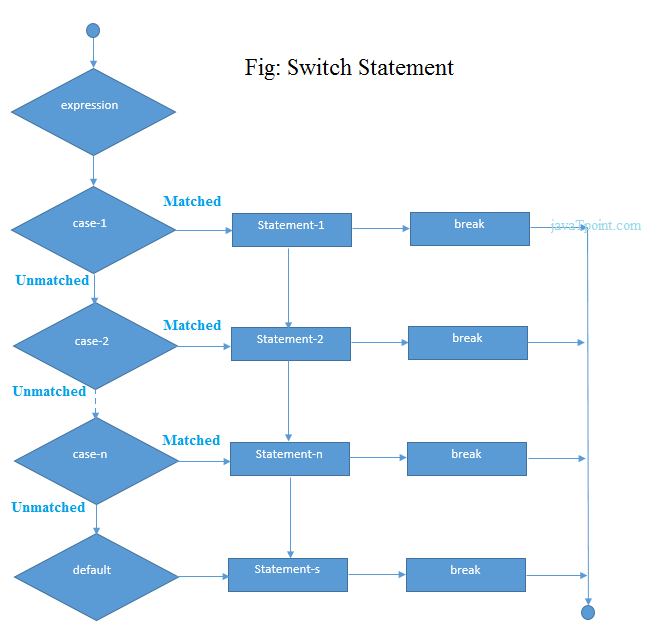
**break**;  //optional

......

**default**:

 code to be executed **if** all cases are not matched;

}



**Example:**

**public** **class** SwitchExample {

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

**int** number=20;

**switch**(number){

**case** 10: System.out.println("10");**break**;

**case** 20: System.out.println("20");**break**;

**case** 30: System.out.println("30");**break**;

**default**:System.out.println("Not in 10, 20 or 30");

    }

}

}

**Output**: 20

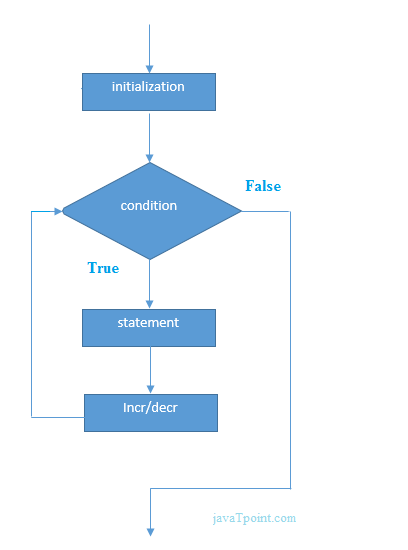
**Iteration Statement:**

* The process of repeatedly executing a statements and is called as looping. The statements may be executed multiple times (from zero to infinite number).
* If a loop executing continuous then it is called as Infinite loop. Looping is also called as iterations.
* In Iteration statement, there are three types of operation:

1. for loop
2. while loop
3. do-while loop

## Java Simple For Loop

The simple for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

**Syntax:**

**for**(initialization;condition;incr/decr){

//code to be executed

}

**Example:**

**public** **class** ForExample

{

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

 {

**for**(**int** i=1;i<=10;i++){

        System.out.println(i);

    }

}

}

**Output:**

1

2

3

4

5

6

7

8

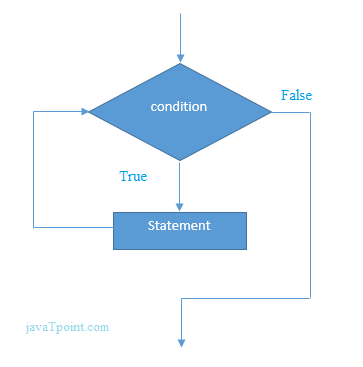
9

10

# Java While Loop

The Java while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

**Syntax:**

**while**(condition){

//code to be executed

}

**Example:**

**public** **class** WhileExample {

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

**int** i=1;

**while**(i<=10){

        System.out.println(i);

    i++;

    }

}

}

Output:

1

2

3

4

5

6

7

8

9

10

## Java Infinitive While Loop

If you pass **true** in the while loop, it will be infinitive while loop.

**Syntax:**

**while**(**true**){

//code to be executed

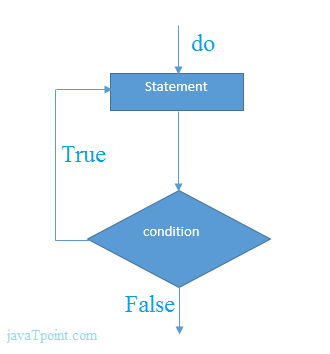
}

# Java do-while Loop

The Java do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The Java do-while loop is executed at least once because condition is checked after loop body.

**Syntax:**

**do**{

//code to be executed

}**while**(condition);

**Example:**

**public** **class** DoWhileExample {

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

**int** i=1;

**do**{

        System.out.println(i);

    i++;

    }**while**(i<=8);

}

}

Output:

1

2

3

4

5

6

7

8

## Java Infinitive do-while Loop

If you pass **true** in the do-while loop, it will be infinitive do-while loop.

**Syntax:**

**do**{

//code to be executed

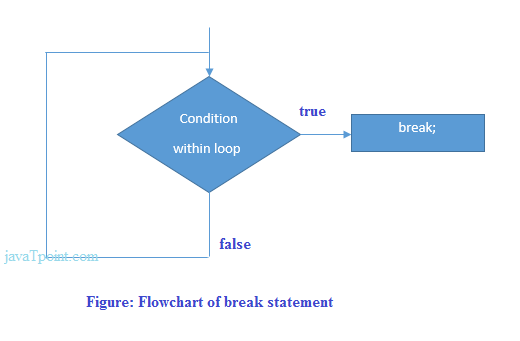
}**while**(**true**);

**JUMPS IN STATEMENT:**

* Statements or loops perform a set of operations continually until the control variable will not satisfy the condition.
* But if we want to break the loops when condition will satisfy then Java give a permission to jump from one statement to end of loop or beginning of loop as well as jump out of a loop.
* “break” keyword use for exiting from loop and “continue” keyword use for continuing the loop.
* Break statement is used to terminate from a loop while a test condition is true.  
  + - 1. Break statement
      2. Continue statement

# Java Break Statement

The Java break is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

**Syntax:**

jump-statement;

**break**;

**Example:**

**public** **class** BreakExample

 {

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

**for**(**int** i=1;i<=10;i++)

{

**if**(i==5)

{

**break**;

        }

        System.out.println(i);

    }

}}

**Output:**

1

2

3

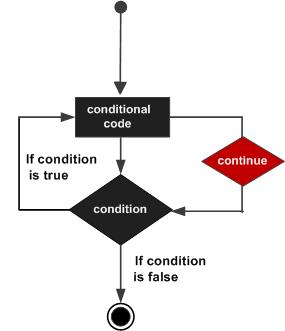
4

# Java Continue Statement

The Java continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

**Syntax:**

jump-statement;

**continue**;

**Example:**

**public** **class** ContinueExample

{

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

{

**for**(**int** i=1;i<=10;i++){

**if**(i==5){

**continue**;

        }

        System.out.println(i);

    }

}

}

Output:

1

2

3

4

6

7

8

9

10