CB2305 - ADVANCED JAVA PROGRAMMING

UNIT I - FUNDAMENTALS OF JAVA

Overview of Object-Oriented Programming — Features of Object-Oriented Programming – Java Buzzwords –The Java Programming Environment– Data Types, Variables, constants –Operators – Mathematical Functions and Constants- Conversions between Numeric Types- Casts- Parentheses and Operator Hierarchy- Enumerated Types-Control flow Statements – Arrays- Programming Structures in Java.

1. OBJECT ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a programming paradigm that organizes software design around data, or objects, rather than functions and logic. OOP aims to implement real-world entities like inheritance, data hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

List of object-oriented programming languages

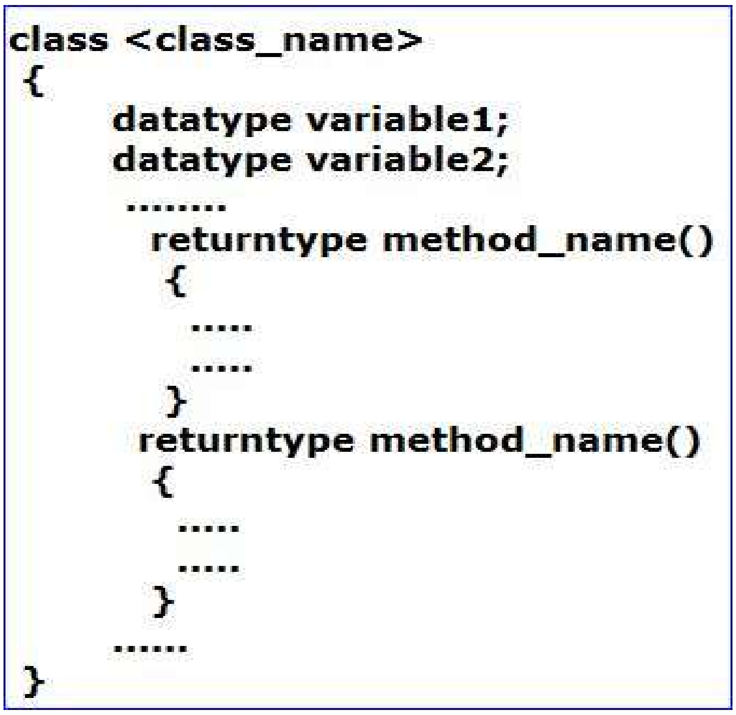
Ada 95	Fortran 2003	PHP since v4, greatly enhanced in v5
BETA	Graphtalk	Python
C++	IDLscript	Ruby
C#	J#	Scala
COBOL	Java	Simula (the first OOP language)
Cobra	LISP	Smalltalk
ColdFusion	Objective-C	Tcl
Common Lisp	Perl since v5	

1.1 Key concepts of Object-oriented programming (OOP)

Key Concepts of Object-Oriented Programming are as follows

- Class
- Object
- Abstraction
- Data Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

- **Class:** A blueprint for creating objects. It defines a set of attributes that will characterize any object that is instantiated from this class, as well as methods for manipulating these attributes.



```
class <class_name>
 {
      datatype variable1;
      datatype variable2;
       ........
        returntype method_name()
         {
           .....
           .....
         }
       returntype method_name()
        {
          .....
          .....
        }
      ......
 }
```

- **Object:** An instance of a class. It has an identity, state, and behaviour. An object is any real-world entity. The two broad classifications of objects are: Tangible Objects (Physical objects) and Intangible Objects (Conceptual Objects).

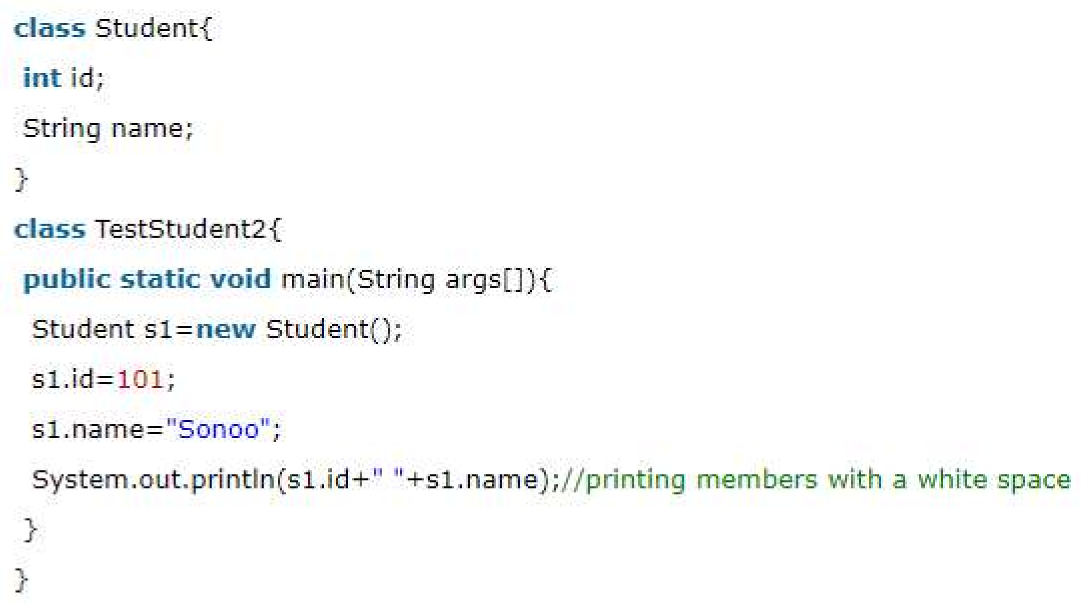
- Tangible objects have physical form that can be perceived by touch E.g. Car, Dishwasher.

- Intangible objects don’t have physical form; they exist only in concept E.g., Bank Account, A course offered by a university.

- Objects have states and behaviours. Example: A dog has states - colour, name, breed and behaviours – wagging the tail, barking, eating.

- An object is an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing details of each other's data or code, the only necessary thing is that the type of message accepted and type of response returned by the objects.

- An object has three characteristics:
  - state: represents data (value) of an object.
  - behaviour: represents the behaviour (functionality) of an object such as deposit, withdraw etc.
  - identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But it is used internally by the JVM to identify each object uniquely.



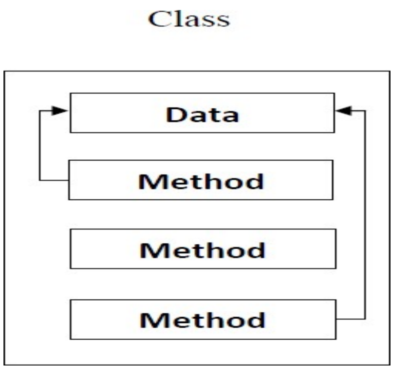
```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

- **Data Abstraction:** The process of hiding certain details and showing only essential information to the user.

- For example: phone call, we don't know the internal processing. In java, we use abstract class and interface to achieve abstraction.

- **Encapsulation:** The process of binding data and functions that manipulate that data together in a single unit, called a class.

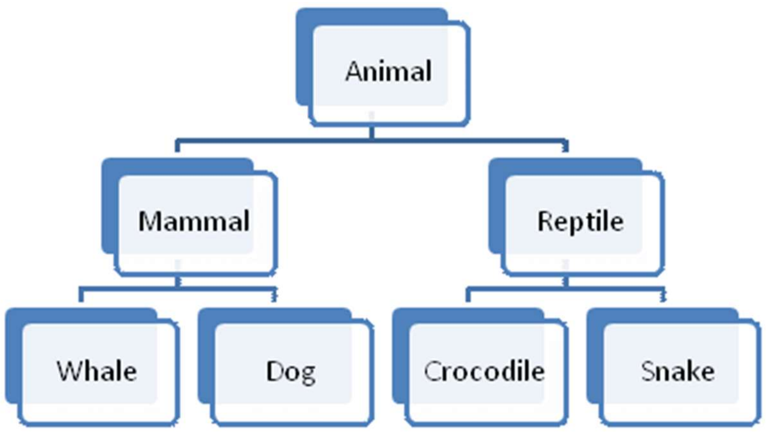
- It is a process of wrapping code and data together into a single unit, for example capsule i.e., mixed of several medicines. A java class is the example of encapsulation. Three main data modifiers are used during encapsulation. Java supports four modifiers for the visibility of classes, methods, and attributes.



  - **Public** — Classes, methods, and variables can be accessed by all the classes.
  - **Private** — Most restrictive modifier, which methods and variables can be accessed within the same class.
  - **Protected** — Methods and variables can be accessed from the package by classes and subclasses.
  - **No Modifier** — Or called package-private because when you don't use any modifiers, you can access it within classes from the package.

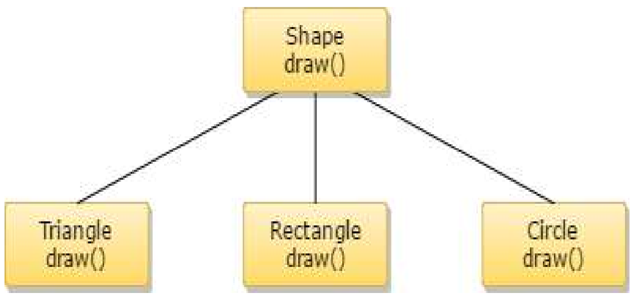
- **Inheritance:** Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behavior of one class to another, i.e., acquiring the properties and behavior by the child class from the parent class. When one object acquires all the properties and behaviors of another object, it is known as inheritance. It provides code reusability and establishes relationships between different classes. A class which inherits the properties is known as Child Class (sub-class or derived class) whereas a class whose properties are inherited is known as Parent class (super-class or base class).

- Types of inheritance in Java: single, multilevel and hierarchical inheritance. Multiple and hybrid inheritance is supported through interface only.



- **Polymorphism:** The ability of an object to take on many forms. In OOP, it refers to the ability of objects of different classes to be used interchangeably.

- 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.



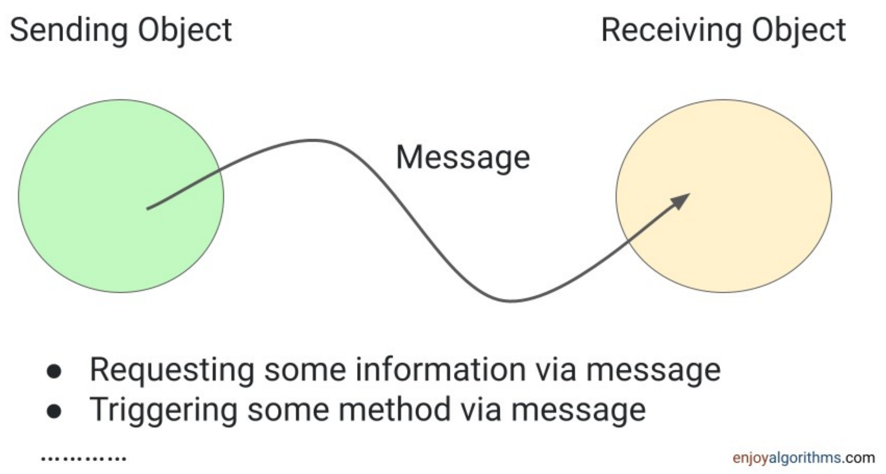
- **Polymorphism is classified into two ways:**
- **Method Overloading (Compile time Polymorphism)**

  Method Overloading is a feature that allows a class to have two or more methods having the same name but the arguments passed to the methods are different. Compile time polymorphism refers to a process in which a call to an overloaded method is resolved at compile time rather than at run time.
- **Method Overriding (Run time Polymorphism)**
  If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java.In other words, If subclass provides the specific implementation of the method that has

been provided by one of its parent class, it is known as method overriding.

- **Dynamic binding** is also referred to as a run-time polymorphism. In this type of binding, the functionality of the method call is not decided at compile-time. In other words, it is not possible to decide which piece of code will be executed as a result of a method call at compile-time.

- **Message Passing** is the idea of message passing in object-oriented programming is a way for objects to communicate from one object to another. Message passing is similar to the idea of sending and receiving messages in real life.


Sending Object
Receiving Object
Message
• Requesting some information via message
• Triggering some method via message
…………
enjoyalgorithms.com

Object-oriented programming has several benefits, including code reusability, scalability, and efficiency.

It is also beneficial for collaborative development, where projects are divided into groups.

1.2 Difference between Procedure Oriented Programming and Object-Oriented Programming:

Comparative Facts	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided intoparts called **objects**.
Importance	In POP, Importance is not givento **data** but to functions as well as **sequence** of actions to bedone.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way toadd new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is less secure.	OOP provides Data Hiding, so provides more security.
Overloading	In POP, Overloading is notpossible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Examples of POP are: C, VB,FORTRAN, and Pascal.	Examples of OOP are: C++, JAVA, VB.NET, C#.NET.

1.3 Benefits of Object-Oriented Programming (OOP) or FEATURES OF JAVA

The main objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some awesome features which play 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 are given below.

1. **Simple:** Java is very easy to learn and its syntax is simple, clean and easy to understand. According to Sun, Javalanguage 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 confusing and rarely-used features e.g. explicit pointers, operatoroverloading etc.
   - There is no need to remove unreferenced objects because there is Automatic Garbage Collection injava.

2. **Code reusability:** Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.

3. **Modularity:** Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.

4. **Productivity:** OOP design is flexible, modular, and abstract, making it particularly useful when creating larger programs.

5. **Data redundancy:** OOP allows for data abstraction, which means that only essential information is shown to the user.

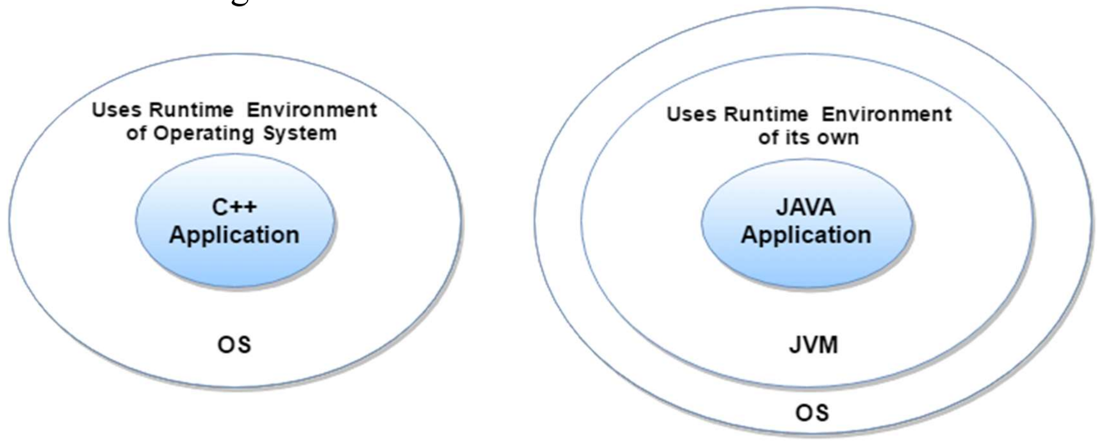
6. **Code flexibility:** OOP design is flexible, modular, and abstract, making it easier to modify code.

7. **Polymorphism:** OOP allows for objects of different classes to be used interchangeably, making code more flexible.

8. **Security:** OOP allows for encapsulation, which means that data and functions that manipulate that data are bound together in a single unit, making it more secure.

   Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

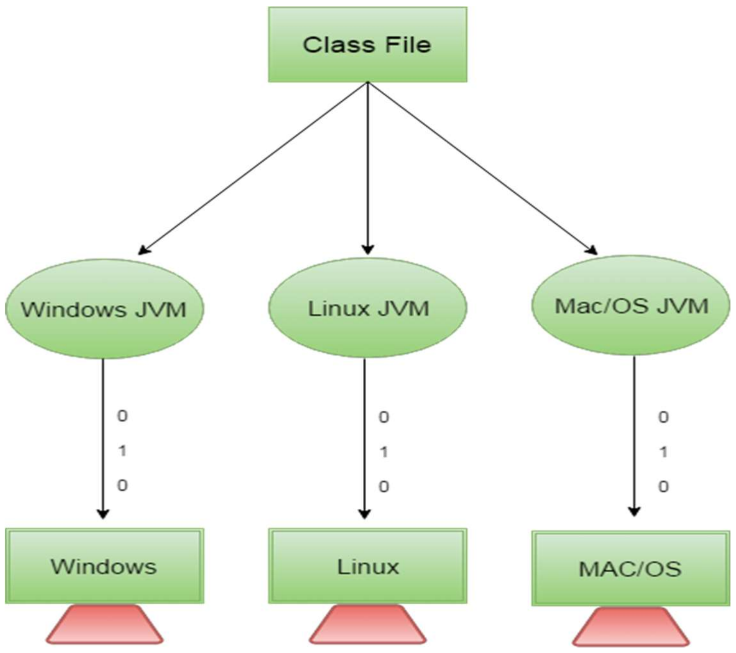
- No explicit pointer
- Java Programs run inside virtual machine sandbox


Uses Runtime Environment of Operating System
C++ Application
OS
Uses Runtime Environment of its own
JAVA Application
JVM
OS

9. **Portable:** Java portability enables Java programs to run on different platforms without modification, thanks to its platform-independent bytecode and the Java Virtual Machine (JVM). This "write once, run anywhere" capability allows developers to deploy Java applications on various devices and operating systems seamlessly. By relying on standard APIs and avoiding platform-specific features, Java ensures consistent behaviour across diverse computing environments.

   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 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).

10. **Object-oriented**: Java is designed around the concept of objects, enabling modular and reusable code, encapsulation of data, and inheritance, promoting a clean and organized programming approach.

11. **Robust**: Java provides strong type checking, exception handling, and automatic garbage collection, enhancing stability and reliability, making it less prone to crashes and memory leaks.

12. **Multithreaded**: Java supports multithreading, allowing concurrent execution of tasks, improving efficiency by utilizing multiple CPU cores and enabling responsive user interfaces and efficient handling of concurrent operations.

13. **Architecture-neutral**: Java's bytecode is platform-independent, allowing it to run on any system with a compatible JVM, ensuring consistent behaviour and easy deployment across diverse architectures.

14. **Interpreted**: Java is first compiled into bytecode and then interpreted by the JVM during runtime, providing platform independence while sacrificing some performance optimizations made possible by direct compilation to machine code.

15. **High Performance**: Java offers just-in-time (JIT) compilation, where bytecode is translated to native machine code at runtime, optimizing performance and making it comparable to statically compiled languages.

16. **Distributed**: Java supports networking and various distributed computing technologies, making it well-suited for building distributed applications and enabling seamless communication between components over networks.

17. **Dynamic**: Java's reflective capabilities allow programs to examine and modify their own structure and behaviour at runtime, providing flexibility for dynamic loading of classes and adaptive programming.

Overall, OOP provides a powerful framework for creating software that is modular, highly scalable, and easy to maintain.

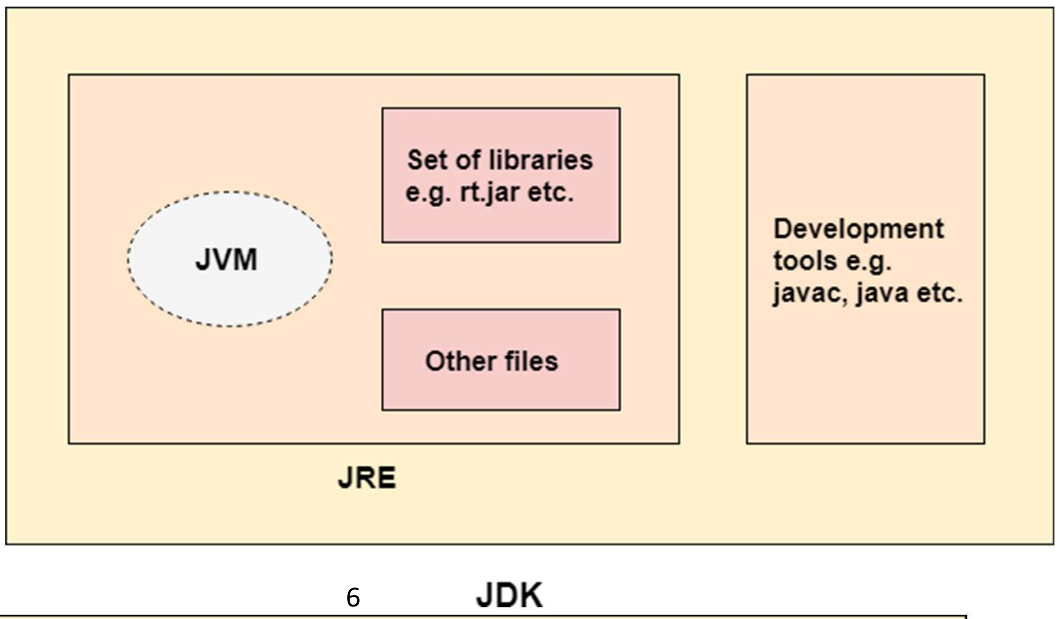
1.4 The Java Environment

In order to understand the Java Environment, one has to understand the Java, Run Time Environment, Java Development Kit and Java Virtual Machine.

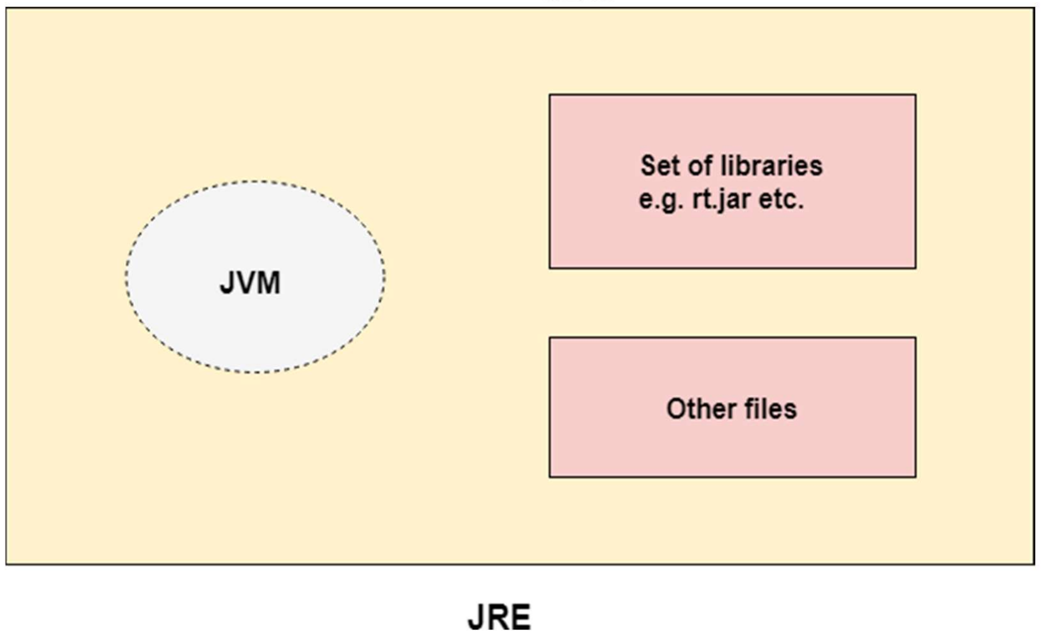
Java Run Time Environment (JRE)

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing java applications. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

Implementation of JVMs is also actively released by other companies besides Sun Micro Systems.



6



Java Development Kit (JDE)

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle corporation: Standard Edition Java Platform

Enterprise Edition Java Platform Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) etc. to complete the development of a Java Application.

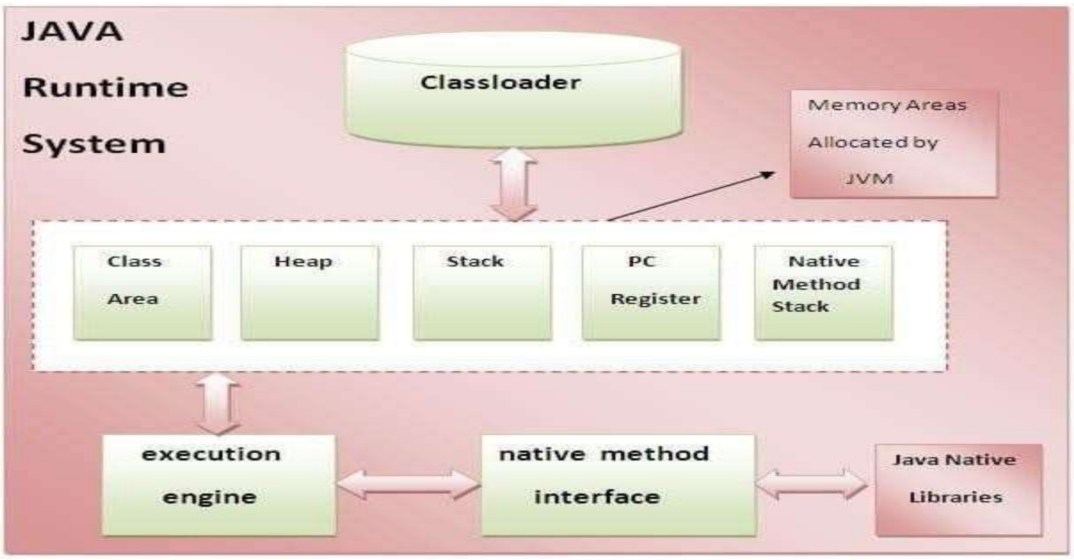
JVM (Java Virtual Machine)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent). The JVM performs following operation:
Loads code Verifies code Executes code
Provides runtime environment JVM provides definitions for the:
Memory area Class file format Register set
Garbage-collected heap Fatal error reporting etc.

Internal Architecture of JVM

1. ***Class loader:*** Classloader is a subsystem of JVM that is used to load class files.
2. ***Class (Method) Area :*** Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.
3. ***Heap:*** It is the runtime data area in which objects are allocated.
4. ***Stack:*** Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as thread. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.
5. **Program Counter Register:** PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.
6. ***Native Method Stack:*** It contains all the native methods used in the application.
7. ***Execution Engine:*** Contains a virtual processor, Interpreter to read bytecode stream then execute the instructions and Just-In- Time (JIT) compiler is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.



1.5 Structure of Java Program

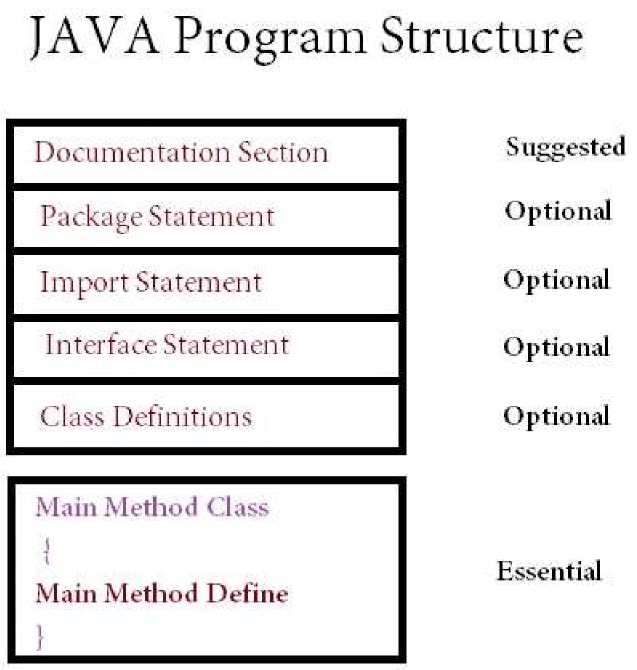
Java program follows a specific structure to be compiled and executed correctly. Here's a general outline of the structure of a typical Java program:

Package Declaration (Optional): The program can begin with an optional package declaration, which helps organize related classes into packages. The package statement is the first line in a Java file and is followed by the package name.

Import Statements (Optional): After the package declaration (if present), import statements can be included to specify which classes from other packages you want to use in the current program.
Interface Statement: An interface defines a contract or a set of method signatures that classes must implement. It enables abstraction, multiple inheritance, and allows the creation of loosely coupled and modular code.

Class Declaration: Every Java program contains at least one class. The class is defined using the class keyword, followed by the class name. The main logic of the program typically resides inside the main class.

Main Method: The entry point of the program is the main method. It is the method that the Java Virtual Machine (JVM) calls to start the execution of the program. The main method must have the following signature:



```
import java.util.Scanner;

public class HelloWorld {

   public static void main(String[] args) {

      // Creates a reader instance which takes
      // input from standard input - keyboard
      Scanner reader = new Scanner(System.in);
      System.out.print("Enter a number: ");

      // nextInt() reads the next integer from the keyboard
      int number = reader.nextInt();

      // println() prints the following line to the output screen
      System.out.println("You entered: " + number);
   }
}
```

Print an Integer entered by an user

Class Body: The class body is enclosed within curly braces {} and contains fields, methods, and nested classes that make up the program's functionality.

Fields (Optional): Fields are variables declared within the class but outside of any method. They represent the data that the class holds. Fields can have various access modifiers to control their visibility.

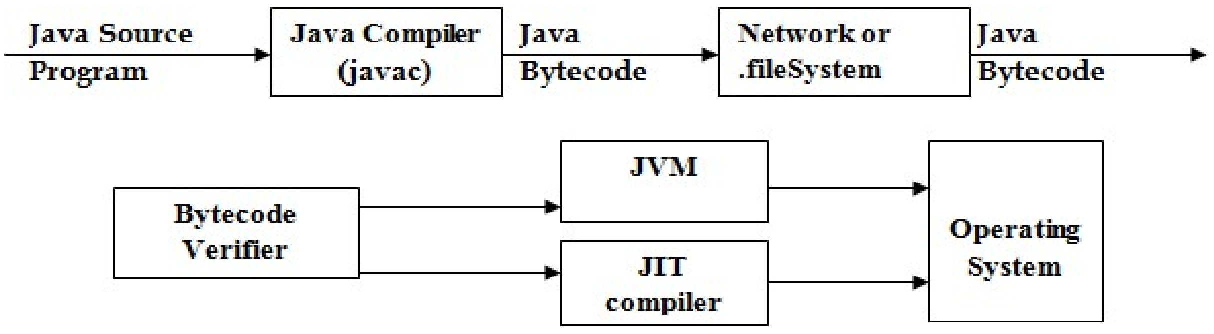
Methods (Optional): Methods are functions defined within the class that perform specific actions or calculations. They can have different access modifiers, return types, and parameters.

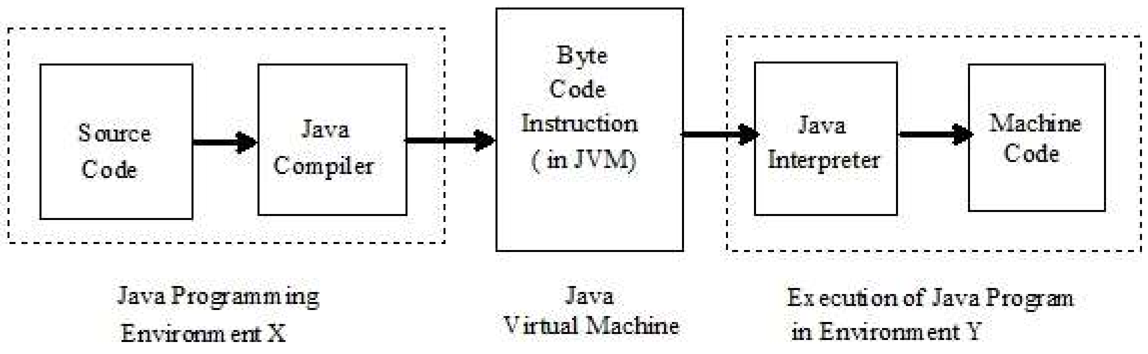
Constructor (Optional): A constructor is a special method used to initialize objects when they are created. It has the same name as the class and is called automatically when a new instance of the class is created.

Comments (Optional): Java allows adding comments in the code to provide explanations and documentation. Comments are ignored by the compiler and JVM and serve as notes for developers.

1.6 Compile and run a Java Program?

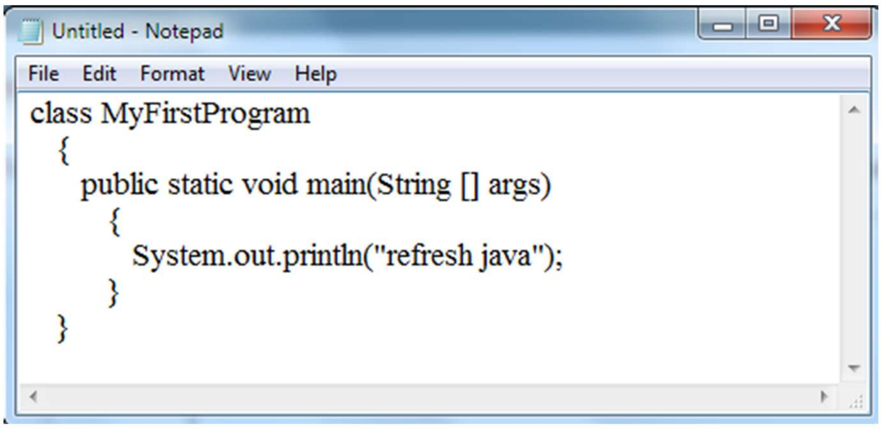
The process of compiling and running Java program is explained in the following diagram neatly and cleanly.




Source Code
Java Compiler
Byte Code Instruction (in JVM)
Java Interpreter
Machine Code
Java Programming Environment X
Java Virtual Machine
Execution of Java Program in Environment Y

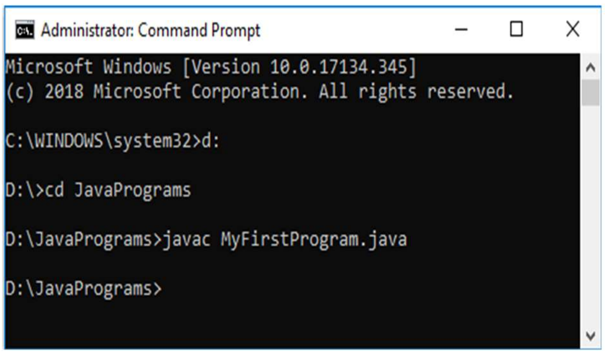
Compile and run a Java Program:

- Type in Notepad and save with the class name as "MyFirstProgram.java"



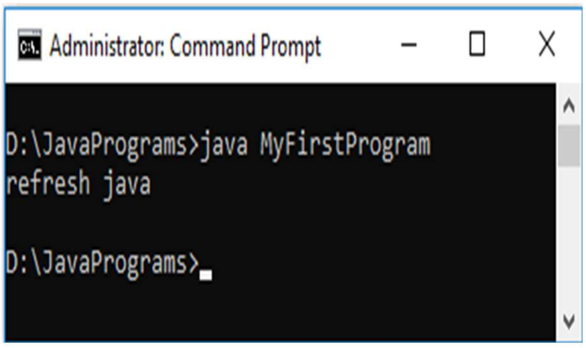
```
class MyFirstProgram
  {
    public static void main(String [] args)
      {
        System.out.println("refresh java");
      }
  }
```

Compile as "javac MyFirstProgram.java"



```
Microsoft Windows [Version 10.0.17134.345]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>d:

D:\>cd JavaPrograms

D:\JavaPrograms>javac MyFirstProgram.java

D:\JavaPrograms>
```

Run as "java MyFirstProgram"



```
D:\JavaPrograms>java MyFirstProgram
refresh java

D:\JavaPrograms>_
```

1.7 Data Types, Variables, constants in Java

Data type is used to allocate sufficient memory space for the data. Data types specify the different sizes and values that can be stored in the variable.

Java is a strongly Typed Language.

Java is a strongly typed programming language because every variable must be declared with a data type. A variable cannot start off life without knowing the range of values it can hold, and once it is declared, the data type of the variable cannot change.

Data types in Java are of two types:

1. Primitive data types (Intrinsic or built-in types) :- : The primitive data types include Boolean, char, byte, short, int, long, float and double.
2. Non-primitive data types (Derived or Reference Types): The non-primitive data types include Classes, Interfaces, Strings and Arrays.

1. Primitive Types:

Primitive data types are those whose variables allow us to store only one value and never allow storing multiple values of same type. This is a data type whose variable can hold maximum one value at a time.

There are eight primitive types in Java:

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are byte, short, int and long. Which type you should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. There are two types: float and double.

Char: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

Boolean: The Boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

2. **Derived Data Types / Reference Data Types:** These Data **Types** will contain a memory address of variable values because the reference types won't store the variable value directly in memory. They are strings, class, objects, arrays, etc.

   Derived data types are those whose variables allow us to store multiple values of same type. But they never allow storing multiple values of different types. A reference variable can be used to refer to any object of the declared type or any compatible type.

Strings

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

Syntax: Declaring a string
<String_Type> <string_variable> = "<sequence_of_string>";

Example:
```
// Declare String without using new operator

String s = "Chennai Institute of Technology";

// Declare String using new operator

String s1 = new String("Chennai Institute of Technology ");
```

Class

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

1. **Modifiers**: A class can be public or has default access. Refer to access specifiers for classes or interfaces in Java.
2. **Class name:** The name should begin with an initial letter (capitalized by convention).
3. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body is surrounded by braces, { }.

Object

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

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

Interface

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.

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

Array

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

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

1.8 Java Variables:

There are three types of variables in Java:

Local Variables
Instance Variables
Static Variables

1. Local Variables

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

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

Time Complexity of the Method:
Time Complexity: O(1)
Auxiliary Space: O(1)

Below is the implementation of the above approach:

- Java

```java
// Java Program to implement
// Local Variables
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        // Declared a Local Variable
        int var = 10;

        // This variable is local to this main method only
        System.out.println("Local Variable: " + var);
    }
}
```

Output

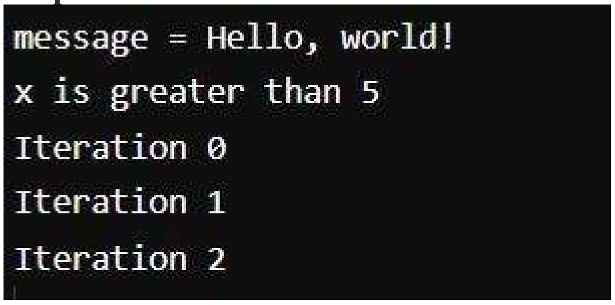
```
Local Variable: 10
```

Example :

- Java

```java
package a;
public class LocalVariable {

    public static void main(String[] args) {
        int x = 10; // x is a local variable
        String message = "Hello, world!"; // message is also a local variable

        System.out.println("x = " + x);
        System.out.println("message = " + message);
        if (x > 5) {
            String result = "x is greater than 5"; // result is a local variable
            System.out.println(result);
        }

        // Uncommenting the line below will result in a compile-time error
        // System.out.println(result);

        for (int i = 0; i < 3; i++) {
            String loopMessage = "Iteration " + i; // loopMessage is a local variable
            System.out.println(loopMessage);
        }

        // Uncommenting the line below will result in a compile-time error
        // System.out.println(loopMessage);
    }
}
```

output :



```
message = Hello, world!
x is greater than 5
Iteration 0
Iteration 1
Iteration 2
```

2 Instance Variables

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

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
- Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable. For *String* it is *null,* for *float* it is *0.0f,* for *int* it is *0,* for Wrapper classes like *Integer* it is *null, etc.*
- Instance variables can be accessed only by creating objects.
- We initialize instance variables using constructors while creating an object. We can also use instance blocks to initialize the instance variables.

The complexity of the method:

Time Complexity: O(1)

Auxiliary Space: O(1)

Below is the implementation of the above approach:

- Java

```java
// Java Program to demonstrate
// Instance Variables
import java.io.*;

class GFG {

    // Declared Instance Variable
    public String geek;
    public int i;
    public Integer I;
    public GFG()
    {
        // Default Constructor
        // initializing Instance Variable
        this.geek = "Shubham Jain";
    }

    // Main Method
    public static void main(String[] args)
    {
        // Object Creation
        GFG name = new GFG();

        // Displaying O/P
        System.out.println("Geek name is: " + name.geek);
        System.out.println("Default value for int is "
                           + name.i);

        // toString() called internally
        System.out.println("Default value for Integer is "
                           + name.I);
    }
}
```

Output

```
Geek name is: Shubham Jain
Default value for int is 0
Default value for Integer is null
```

3 Static Variables

Static variables are also known as class variables.

- These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor, or block.
- Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.

- Initialization of a static variable is not mandatory. Its default value is dependent on the data type of variable. For *String* it is *null*, for *float* it is *0.0f*, for *int* it is *0*, for *Wrapper classes* like *Integer* it is *null,* etc.
- If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won't halt the program. The compiler will replace the object name with the class name automatically.
- If we access a static variable without the class name, the compiler will automatically append the class name. But for accessing the static variable of a different class, we must mention the class name as 2 different classes might have a static variable with the same name.
- Static variables cannot be declared locally inside an instance method.
- Static blocks can be used to initialize static variables.

The complexity of the method:

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

Below is the implementation of the above approach:

- Java

```java
// Java Program to demonstrate
// Static variables
import java.io.*;

class GFG {
    // Declared static variable
    public static String geek = "Shubham Jain";

    public static void main(String[] args)
    {

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

        // static int c=0;
        // above line,when uncommented,
        // will throw an error as static variables cannot be
        // declared locally.
    }
}
```

Output

```
Geek Name is: Shubham Jain
```

Declaration of a Variable:

- Syntax to declare variables:
  datatype identifier [=value] [, identifier [=value] …];
- Example of Variable names:
  int average=0.0, height, total height;
- Rules followed for variable names (consist of alphabets, digits, underscore and dollar characters)
- A variable name must begin with a letter and must be a sequence of letter or digits.
  1. They must not begin with digits.
  2. Uppercase and lowercase variables are not the same.
  3. Example: Total and total are two variables which are distinct.
  4. It should not be a keyword.
  5. Whitespace is not allowed.
  6. Variable names can be of any length.

- Initializing Variables:
- After the declaration of a variable, it must be initialized by means of assignment statement.
- It is not possible to use the values of uninitialized variables.

  Ex:
  int months=12;   or   int months;
  months=1;

Dynamic Initialization of a Variable:

Java allows variables to be initialized dynamically using any valid expression at the time the variable is declared.

Example: Program that computes the remainder of the division operation:

```
class FindRemainer
{
public static void main(String args[])
{
int num=5,den=2;
int rem=num%den; System.out.println(—Remainder is —+rem);
}
}
```
Output:

Remainder is 1

In the above program there are three variables num, den and rem. num and den ate initialized by constants whereas rem is initialized dynamically by the modulo division operation on num and den.

Difference between Instance variable and Static variable:

INSTANCE VARIABLE	**STATIC VARIABLE**
Each object will have its **own copy** of instancevariable	We can only have **one copy** of a static variable perclass irrespective of how many objects we create.
Changes made in an instance variable using one object will **not be reflected** in other objects as eachobject has its own copy of instance variable	In case of static changes **will be reflected** in other objects as static variables are common to all objectof a class.
We can access instance variables **through objectreferences**	Static Variables can be accessed **directly using classname.**

Class Sample
{
int a;
}
	Class Sample
	{
	static int a;
	}

1.9 JAVA COMMENTS

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

There are 3 types of comments in java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

1) Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

//This is single line comment

Example:

public class CommentExample1

```
{
public static void main(String[] args)
{
int i=10;//Here, i is a variable System.out.println(i);
}
}
```

Output:

```
10
```

2) Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

```
/*
This is
multi line comment
*/
```

Example:

```
public class CommentExample2
{
public static void main(String[] args)
{
/* Let's declare and print variable in java. */ int i=10; System.out.println(i);
}
}
```

Output:

```
10
```

3) Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use javadoc tool.

Syntax:

```
/** This is
documentation comment
*/
```

Example:

```
/** The Calculator class provides methods to get addition and subtraction of
given 2 numbers.*/ public class Calculator
{
/** The add() method returns addition of given numbers.*/ public static int
add(int a, int b)
{
return a+b;
}
/** The sub() method returns subtraction of given numbers.*/ public static int
sub(int a, int b)
{
 return a-b;
}
}
```

This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a */.

1.10 OPERATORS IN JAVA

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.

Java Operator Precedence

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 Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Java Unary Operator Example: ++ and --

```
public class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=10;
System.out.println(a++ + ++a);//10+12=22
System.out.println(b++ + b++);//10+11=21

}}
```

Output:

```
22
21
```

Java Unary Operator Example: ~ and !

```java
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=-10;
boolean c=true;
boolean d=false;
System.out.println(~a);//-11 (minus of total positive value which starts from 0)
System.out.println(~b);//9 (positive of total minus, positive starts from 0)

System.out.println(!c);//false (opposite of boolean value)
System.out.println(!d);//true
}}
```

Output:

```
-11
9
false
true
```

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Java Arithmetic Operator Example

```java
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

Output:

```
15
5
50
2
0
```

Java Arithmetic Operator Example: Expression

```
public class OperatorExample{
public static void main(String args[]){
System.out.println(10*10/5+3-1*4/2);
}}
```

Output:

```
21
```

Java Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Java Left Shift Operator Example

```
public class OperatorExample{
public static void main(String args[]){
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
System.out.println(15<<4);//15*2^4=15*16=240
}}
```

Output:

```
40
80
80
240
```

Java Right Shift Operator

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

```
public OperatorExample{
public static void main(String args[]){
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}}
```

Output:

```
2
5
2
```

Java Shift Operator Example: >> vs >>>

```
public class OperatorExample{
public static void main(String args[]){
    //For positive number, >> and >>> works same
    System.out.println(20>>2);
    System.out.println(20>>>2);
    //For negative number, >>> changes parity bit (MSB) to 0
    System.out.println(-20>>2);
    System.out.println(-20>>>2);
}}
```

Output:

```
5
5
-5
1073741819
```

Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}}
```

Output:

```
false
false
```

Java AND Operator Example: Logical && vs Bitwise &

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a++<c);//false && true = false
System.out.println(a);//10 because second condition is not checked
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
false
10
false
11
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a>b||a<c);//true || true = true
System.out.println(a>b|a<c);//true | true = true
//|| vs |
System.out.println(a>b||a++<c);//true || true = true
System.out.println(a);//10 because second condition is not checked
System.out.println(a>b|a++<c);//true | true = true
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
true
true
true
10
true
11
```

Java Ternary Operator

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

Java Ternary Operator Example

```
public class OperatorExample{
public static void main(String args[]){
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
2
```

Another Example:

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

```
int a=10;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
5
```

Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Java Assignment Operator Example

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
14
16
```

Java Assignment Operator Example

```
public class OperatorExample{
public static void main(String[] args){
int a=10;
a+=3;//10+3
System.out.println(a);
a-=4;//13-4
System.out.println(a);
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}}
```

Output:

```
13
9
18
9
```

Java Assignment Operator Example: Adding short

```
public class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
//a+=b;//a=a+b internally so fine
a=a+b;//Compile time error because 10+10=20 now int
System.out.println(a);
}}
```

Output:

```
Compile time error
```

After type cast:

```
public class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
a=(short)(a+b);//20 which is int now converted to short
System.out.println(a);
}}
```

Output:

```
20
```

1.11 CONTROL STATEMENTS

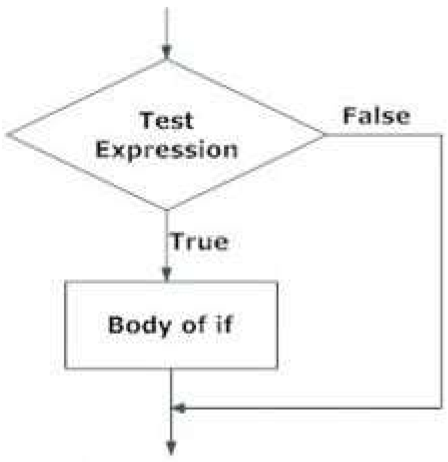
Selection Statements in Java

A programming language uses control statements to control the flow of execution of program based on certain conditions.
Java's Selection statements:
if
if-else
nested-if
if-else-if
switch-case
jump – break, continue, return

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

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

Condition after evaluation will be either true or false. If the value is true then it will execute the block of statements under it. If there are no curly braces '{' and '}' after if(condition) then by default if statement will consider the immediate one statement to be inside its block.



Example:
class IfSample

```
{
public static void main(String args[])
{
int x, y; x = 10;
y = 20;

if(x < y)
System.out.println("x is less than y"); x = x * 2;
if(x == y)
System.out.println("x now equal to y"); x = x * 2;
if(x > y)
System.out.println("x now greater than y");
// this won't display anything if(x == y)
System.out.println("you won't see this");
}
}
```

Output:

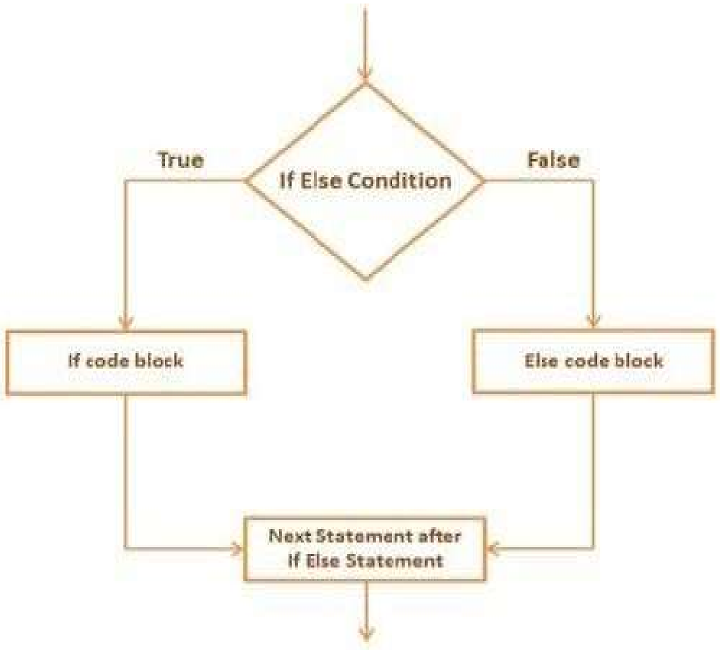
```
x is less than y
x now equal to y
x now greater than y
```

if-else Statement

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

Syntax:.

```
If(condition)
{



}
else
{


}

//Executes this block if
//condition is true

//Executes this block if
//condition is false
```



Example:

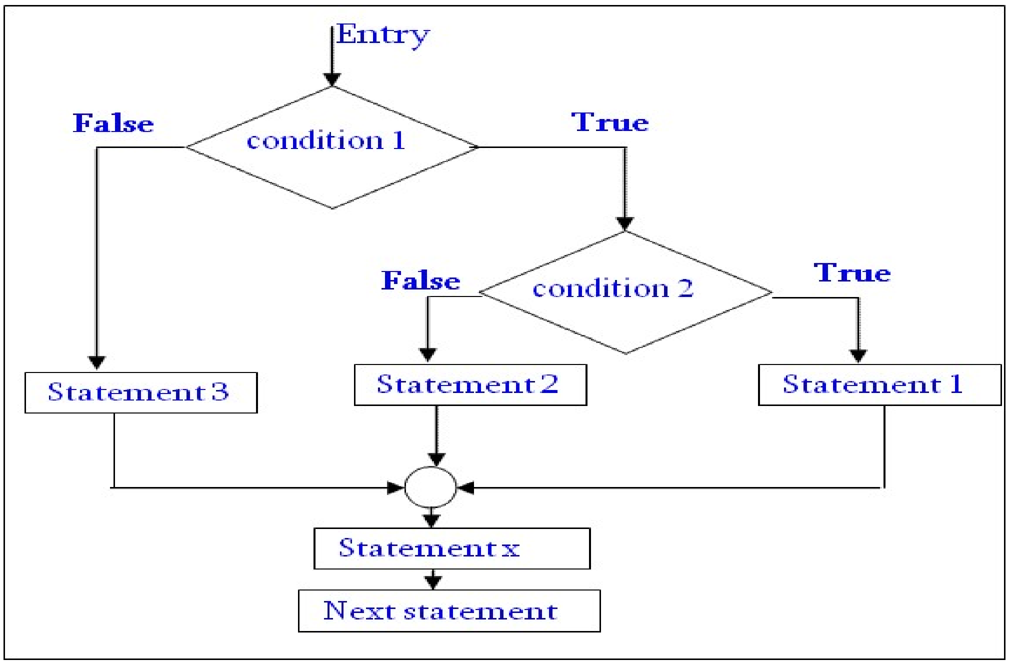
```
public class IfElseExample
{
public static void main(String[] args)
{
int number=13; if(number%2==0){
System.out.println("even number");
}else
{
System.out.println("odd number");
} } }
```

Output:

```
odd number
```

Nested if Statement

Nested if-else statements, is that using one if or else if statement inside another if or else if statement(s).



Example:

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

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

if (i < 12)
System.out.println("i is smaller than 12 too"); else
System.out.println("i is greater than 15");}
}
}
```

Output:

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

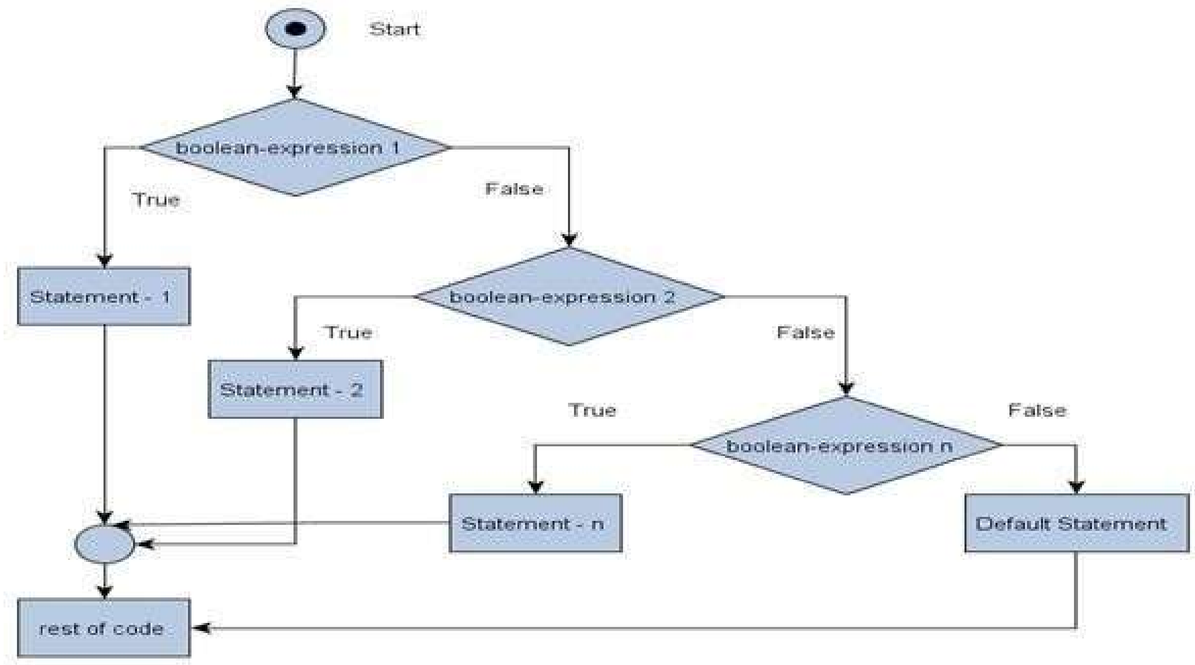
if-else-if ladder statement

The if statements are executed from the top down. The conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:

if(condition) statement;

```
else if(condition) statement;
else if(condition) statement;
.
.
else statement;
```



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

Switch Statements

The switch statement is Java’s multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.

Syntax:

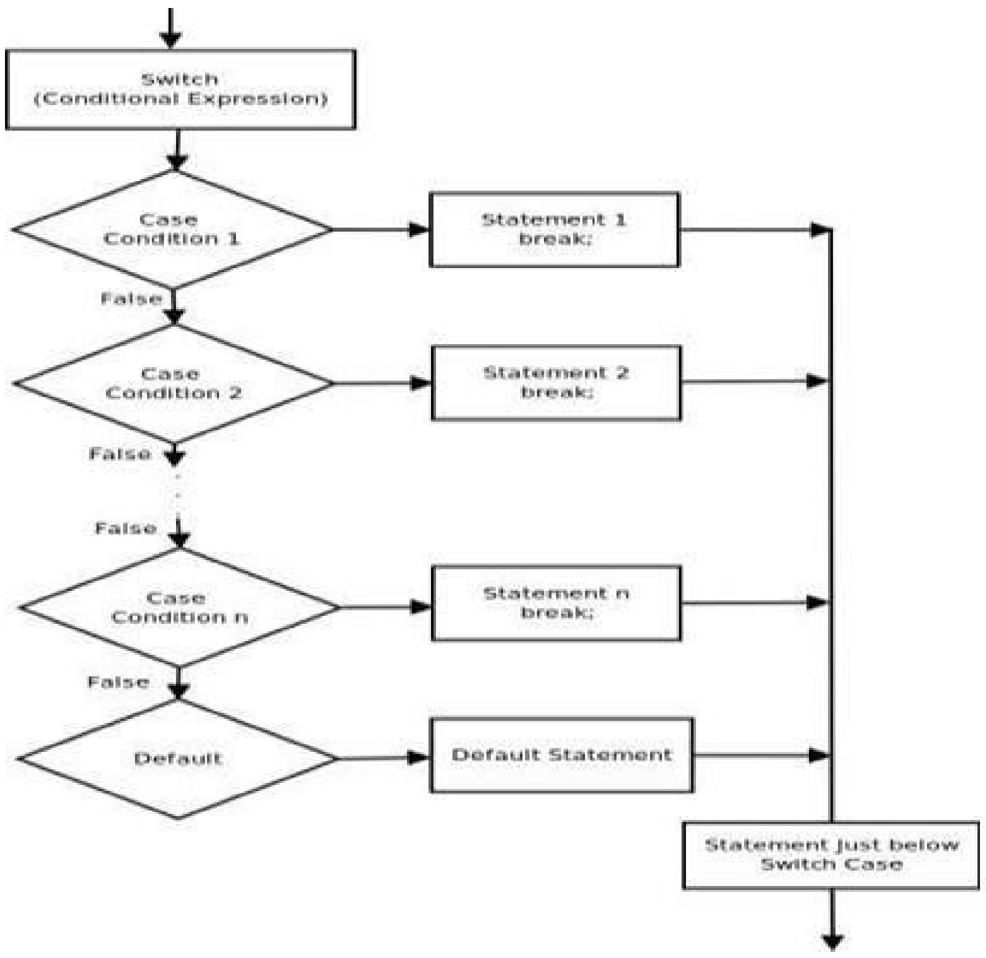
```
switch (expression) { case value1:
// statement sequence break;
case value2:
// statement sequence break;
.
.
case valueN :
// statement sequence break;
default:
// default statement sequence
}
```

Example:

```
// A simple example of the switch. class SampleSwitch {
public static void main(String args[]) { for(int i=0; i<6; i++)
switch(i) { case 0:
System.out.println("i is zero.");

break; case 1:
System.out.println("i is one."); break;
case 2:
System.out.println("i is two."); break;
case 3:
System.out.println("i is three."); break;
default:
System.out.println("i is greater than 3.");
}}}
```

Output:

```
i is zero. i is one. i is two.
i is three.
i is greater than 3. i is greater than 3.
```

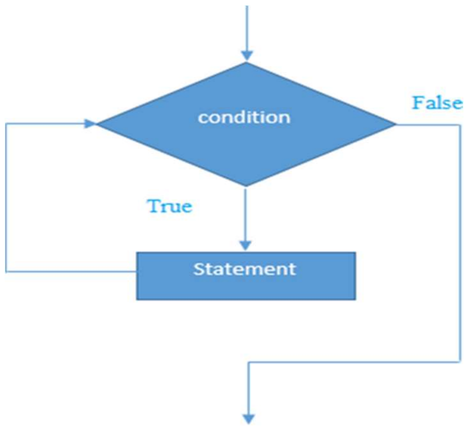


ITERATIVE STATEMENTS

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in java.

while loop
do-while loop
For loop
while loop
A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.
Syntax:

```
while(condition) {
// body of loop
}
```



While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. It is called as Entry controlled loop.

Normally the statements contain an update value for the variable being processed for the next iteration.

When the condition becomes false, the loop terminates which marks the end of its life cycle.

Example:

```
// Demonstrate the while loop. class While {
public static void main(String args[]) { int n = 5;
while(n > 0) { System.out.println("tick " + n); n--;
}
}
}
```

Output:

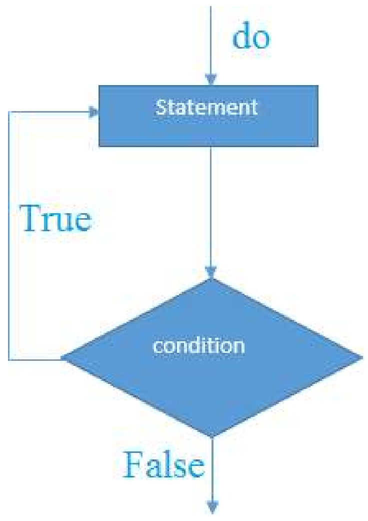
```
tick 5
tick 4
tick 3
tick 2
tick 1
```

do-while loop:

do while loop checks for condition after executing the statements, and therefore it is called as Exit Controlled Loop.

Syntax:

```
do {
// body of loop
} while (condition);
```



do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.

After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.

When the condition becomes false, the loop terminates which marks the end of its life cycle.

It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.

Example

```
public class DoWhileExample { public static void main(String[] args) {
int i=1; do{
System.out.println(i); i++;
}while(i<=5);
}
}
```

Output:

```
1
2
3
4
5
```

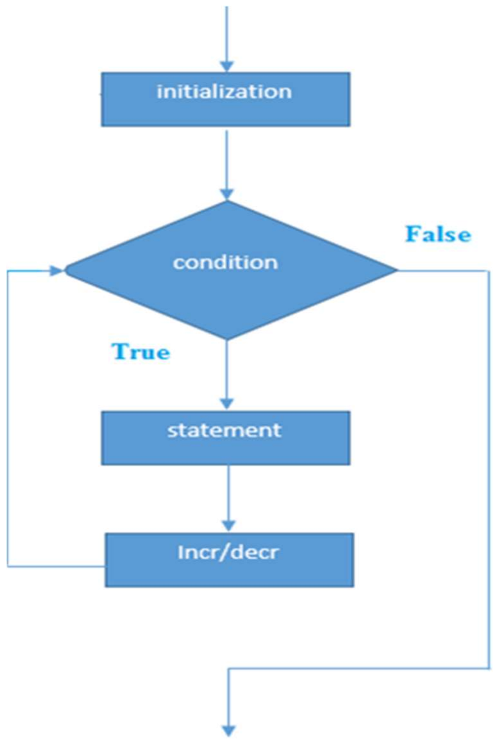
for loop

for loop provides a concise way of writing the loop structure. A for statement consumes the initialization, condition and increment/decrement in one line.

Syntax

```
for(initialization; condition; iteration) {
// body
```

}



Initialization condition: Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.

Testing Condition: It is used for testing the exit condition for a loop. It must return a boolean value. It is also an Entry Control Loop as the condition is checked prior to the execution of the loop statements.

Statement execution: Once the condition is evaluated to true, the statements in the loop body are executed.

Increment/ Decrement: It is used for updating the variable for next iteration.
Loop termination:When the condition becomes false, the loop terminates marking the end of its life cycle.

Example

```
public class ForExample {
public static void main(String[] args) { for(int i=1;i<=5;i++){
System.out.println(i);
}
} }
```

Output:

```
1
2
```

```
3
4
5
```

for-each Loop

The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation. It works on elements basis not index. It returns element one by one in the defined variable.
Syntax:

```
for(type itr-var : collection) statement-block
```

Example:

```
// Use a for-each style for loop. class ForEach {
public static void main(String args[]) {
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
// use for-each style for to display and sum the values for(int x : nums) {
System.out.println("Value is: " + x); sum += x;
}
System.out.println("Summation: " + sum);
}
}
```

Output:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

Nested Loops

Java allows loops to be nested. That is, one loop may be inside another.
Example:

```
// Loops may be nested. class Nested {
public static void main(String args[]) { int i, j;
for(i=0; i<10; i++) { for(j=i; j<10; j++)

System.out.print("."); System.out.println();
}}
}
```

Output:

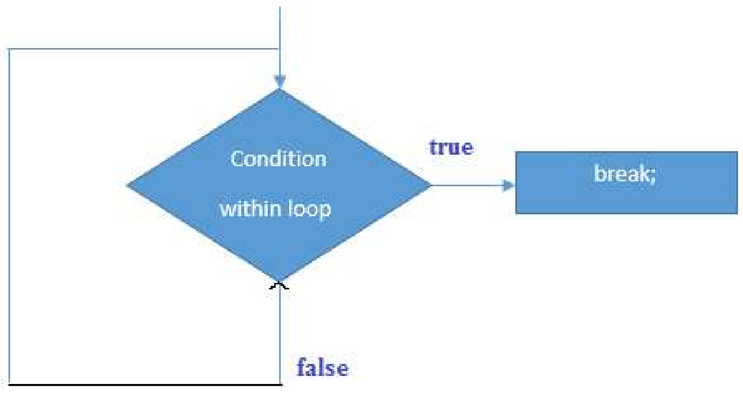
```
..........
.........
........
.......
......
.....
....
...
..
.
```

JUMP STATEMENTS

Java Break Statement

When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

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.



Example:

```
// Using break to exit a loop. class BreakLoop {
public static void main(String args[]) { for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10 System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

Output:

```
i: 0
i: 1

i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

Java Continue Statement

The continue statement is used in loop control structure when you need to immediately jump to the next iteration of the loop. It can be used with for loop or while loop.

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.

Example:

```
// Demonstrate continue. class Continue {
public static void main(String args[]) { for(int i=0; i<10; i++) { System.out.print(i
+ " ");
if (i%2 == 0) continue; System.out.println("");
}
}
}
```

This code uses the % operator to check if i is even. If it is, the loop continues without printing a newline.

Output:

```
0 1
2 3
4 5
6 7
8 9
```

Return

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

Example:

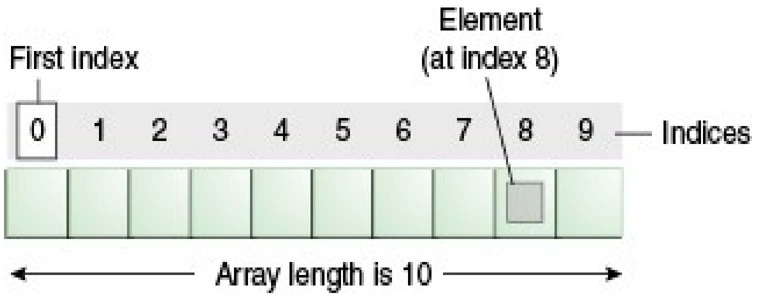
```
// Demonstrate return. class Return {
public static void main(String args[]) { boolean t = true;
System.out.println("Before the return."); if(t) return; // return to caller
System.out.println("This won't execute.");
}
}
```

Output:

```
Before the return.
```

1.12 ARRAYS

1. Array is a collection of similar type of elements that have contiguous memory location.
2. In Java all arrays are dynamically allocated.
3. Since arrays are objects in Java, we can find their length using member length.
4. A Java array variable can also be declared like other variables with [] after the data type.
5. The variables in the array are ordered and each have an index beginning from 0.
6. Java array can be also be used as a static field, a local variable or a method parameter.
7. The size of an array must be specified by an int value and not long or short.
8. The direct superclass of an array type is Object.
9. Every array type implements the interfaces Cloneable and java.io.Serializable.



Advantage of Java Array

- Code Optimization: It makes the code optimized, we can retrieve or sort the data easily.
- Random access: We can get any data located at any index position.

Disadvantage of Java Array

- Size Limit: We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java
1. One- Dimensional Array
2. Multidimensional Array

One-Dimensional Arrays
An array is a group of like-typed variables that are referred to by a common name. An array declaration has two components: the type and the name. type declares the element type of the array. The element type determines the data type of each element that comprises the array. We can also create an array of other primitive data types like char, float, double..etc or user defined data type(objects of a class).Thus, the element type for the array determines what type of data the array will hold.
Syntax:
type var-name[ ];
Instantiation of an Array in java array-var = new type [size]; Example:

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation a[0]=10;//initialization

a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array Example:

```
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```
Output:
```
33
3
4
5
```

Passing Array to method in java

We can pass the java array to method so that we can reuse the same logic on any array.

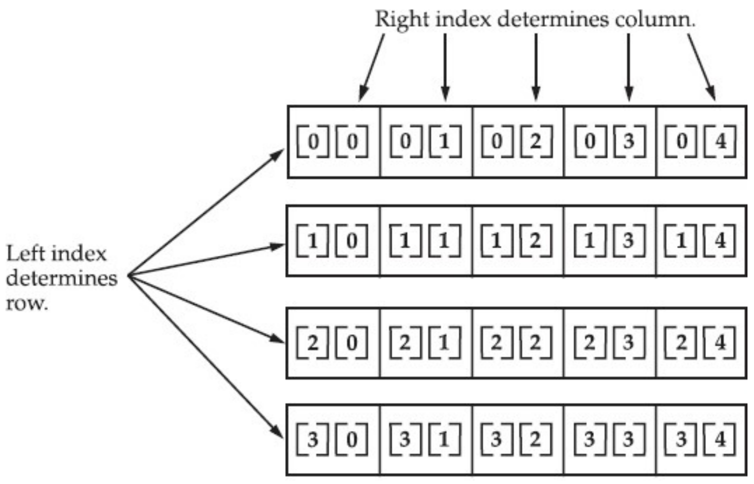
Example:
```
class Testarray2{
static void min(int arr[]){ int min=arr[0];
for(int i=1;i<arr.length;i++) if(min>arr[i])
min=arr[i]; System.out.println(min);
}
public static void main(String args[]){ int a[]={33,3,4,5};
min(a);//passing array to method
}}
```
Output:
```
3
```

Multidimensional Arrays

Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other array. These are also known as Jagged Arrays. A multidimensional array is created by appending one set of square brackets ([]) per dimension.

Syntax:
```
type var-name[ ][ ]=new type[row-size ][col-size ];
```
Example:
```
// Demonstrate a two-dimensional array. class TwoDArray {
public static void main(String args[]) { int twoD[][]= new int[4][5];
int i, j, k = 0; for(i=0; i<4; i++) for(j=0; j<5; j++) { twoD[i][j] = k; k++;
}
for(i=0; i<4; i++) { for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " "); System.out.println();
}
}
}
```
Output:
```
0 1 2 3 4
```

```
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```



Given: int twoD [ ] [ ] = new int [4] [5] ;

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of twoD when it is declared. It allocates the second dimension manually.

Syntax:

```
int twoD[][] = new int[4][]; twoD[0] = new int[5]; twoD[1] = new int[5]; twoD[2]
= new int[5];

twoD[3] = new int[5];
```

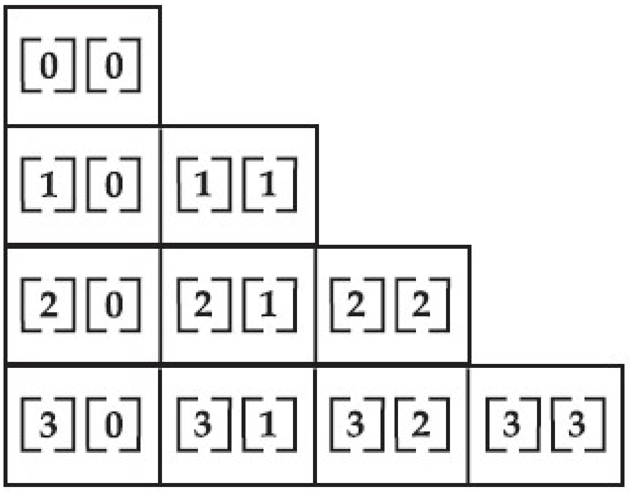
Example:

```
// Manually allocate differing size second dimensions. class TwoDAgain {
public static void main(String args[]) { int twoD[][] = new int[4][];
twoD[0] = new int[1]; twoD[1] = new int[2]; twoD[2] = new int[3]; twoD[3] =
new int[4]; int i, j, k = 0;
for(i=0; i<4; i++) for(j=0; j<i+1; j++) { twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) { for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " "); System.out.println();
}
}
}
```

Output:

```
0
1 2
3 4 5
6 7 8 9
```

The array created by this program looks like this:



1.13 PACKAGES

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

Defining a Package

To create a package the programmer need to include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If package statement is omitted, the class names are put into the default package, which has no name.

Syntax:

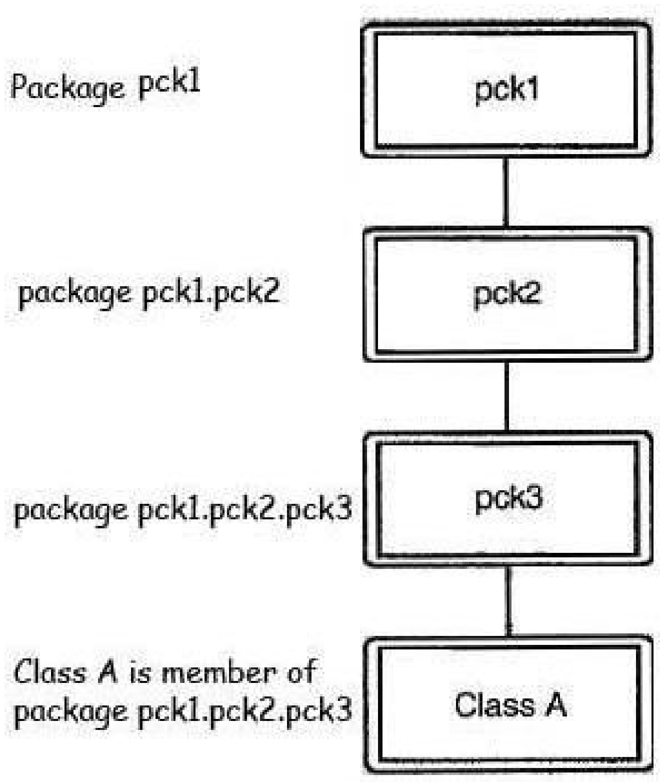
```
package <fully qualified package name>;
package pkg;
```

Here, pkg is the name of the package. For example, the following statement creates a package called MyPackage.

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage.

It is possible to create a hierarchy of packages. The general form of a multileveled package statement is shown here:
package pkg1[.pkg2[.pkg3]];



A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as
package java.awt.image;

needs to be stored in java\awt\image in a Windows environment. We cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable. Third, you can use the classpath option with java and javac to specify the path to your classes.

consider the following package specification:

```
package MyPack
```

In order for a program to find MyPack, one of three things must be true. Either the program can be executed from a directory immediately above MyPack, or the CLASSPATH must be set to include the path to MyPack, or the -classpath option must specify the path to MyPack when the program is run via java.
When the second two options are used, the class path must not include MyPack, itself. It must simply specify the path to MyPack. For example, in a Windows environment, if the path to MyPack is C:\MyPrograms\Java\MyPack
then the class path to MyPack is C:\MyPrograms\Java
Example:

```
// A simple package package MyPack; class Balance { String name; double bal;
Balance(String n, double b) { name = n;
bal = b;
}
void show() { if(bal<0)
System.out.print("--> "); System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) { Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23); current[1] = new
Balance("Will Tell", 157.02); current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file AccountBalance.java and put it in a directory called MyPack.
Next, compile the file. Make sure that the resulting .class file is also in the MyPack directory. Then, try executing the AccountBalance class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above MyPack when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path MyPack.)
As explained, AccountBalance is now part of the package MyPack. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Example: package pck1; class Student

```
{
private int rollno; private String name; private String address;
public Student(int rno, String sname, String sadd)
{
rollno = rno; name = sname; address = sadd;
}
public void showDetails()
{
System.out.println("Roll No :: " + rollno); System.out.println("Name :: " +
name); System.out.println("Address :: " + address);
}
}
public class DemoPackage
{
public static void main(String ar[])
{
Student st[]=new Student[2];
st[0] = new Student (1001,"Alice", "New York"); st[1] = new
Student(1002,"BOb","Washington"); st[0].showDetails();
st[1].showDetails();
}
}
```

There are two ways to create package directory as follows:

1. Create the folder or directory at your choice location with the same name as package name. After compilation of copy .class (byte code file) file into this folder.
2. Compile the file with following syntax.

```
javac -d <target location of package> sourceFile.java
```

The above syntax will create the package given in the sourceFile at the <target location of pacakge> if it is not yet created. If package already exist then only the .class (byte code file) will be stored to the package given in sourceFile.

Steps to compile the given example code:

Compile the code with the command on the command prompt. javac -d DemoPackage.java

1. The command will create the package at the current location with the name pck1, and contains the file DemoPackage.class and Student.class
2. To run write the command given below java pckl.DemoPackage

Note: The DemoPackate.class is now stored in pck1 package. So that we've to use fully qualified type name to run or access it.

Output:

```
Roll No :: 1001 Name :: Alice Address :: New York Roll No :: 1002 Name :: Bob
Address :: Washington
```