* Inheritance in Java :-

→ Inheritance is one of the key features of object oriented programming. Inheritance can be defined as the process where one class acquires the properties (Methods and fields) of another class.

→ The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.

→ Inheritance in Java can be best understood in terms of parent and child relationship, also known as super class (parent) and sub class (child) in Java language.

→ Inheritance defines 'is-a' relationship between a super class and its sub class.

→ The main use of inheritance is code reusability.

Syntax of Java inheritance :

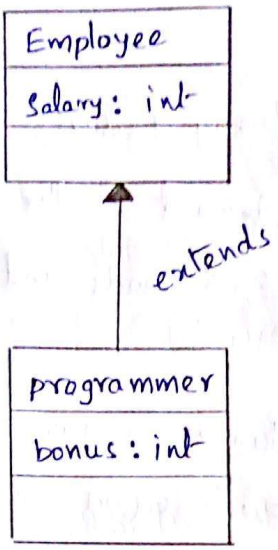
```
class subclass-name extends superclass-name
{
  // Methods and fields
}
```

→ In the above syntax, the extends keyword indicates that you are making a new class that derives from an existing class.

→ In simple words, the extends keyword is used to perform inheritance in Java.

→ In the terminology of Java, a class that is inherited is called a super class, the new class is called a subclass.

Example :-



→ In the figure, programmer is the subclass and Employee is the super class.

→ Relationship between two classes is programmer IS-A Employee.

programmer.java

```
class Employee
{
  int salary = 4000;
}
class programmer extends Employee
{
  int bonus = 1000;
  public static void main (String args[])
  {
    programmer p = new programmer();
    System.out.println ("programmer salary is :" + p.salary);
    System.out.println (" Bonus of programmer is :" + p.bonus);
  }
}
```

output:-

```
javac programmer.java
java programmer
programmer salary is : 4000
Bonus of programmer is : 1000
```

Types of inheritance:

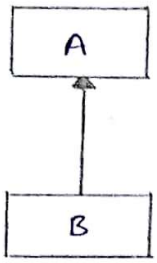
In Java, there can be three types of inheritance

Those are → Single inheritance
→ Multilevel inheritance
→ Hierarchical inheritance

Note:- Multiple inheritance is not supported in Java.

* Single inheritance:

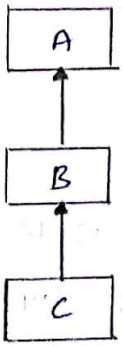
When a class extends another one class only then we call it a single inheritance.



• In the figure, the class B extends the class A. Here A is a parent class and B is a child class.

* Multilevel inheritance:

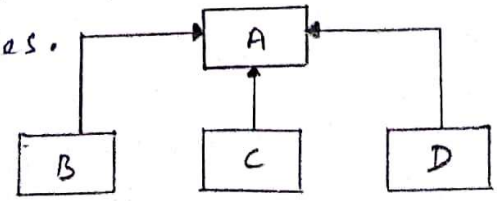
When a class is derived from another class and it acts as the parent class to other class, is known as Multilevel inheritance.



• In the figure, the class 'B' inherits properties from class 'A' and again class 'B' acts as a parent to class 'C'.

* Hierarchical inheritance:

In this, one parent class will be inherited by many sub classes.

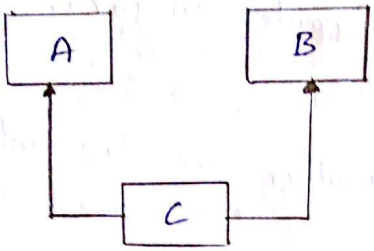


• In the figure, the class 'A' will be inherited by class 'B' class 'c' and class 'D'.

We have two more inheritances: Multiple & Hybrid, which are not directly supported by Java.

* Multiple Inheritance:

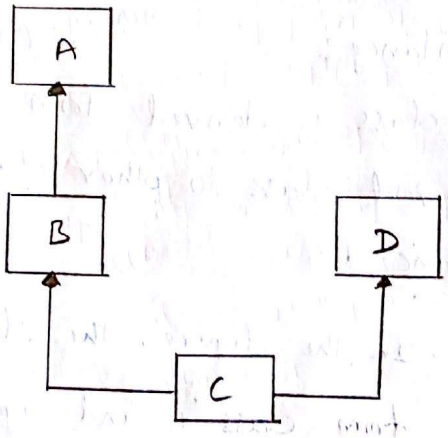
Multiple Inheritance is nothing but one class extending more than one class. It is basically not supported by many Object Oriented programming languages such as Java, Small talk.



Note:- We can achieve multiple inheritance in Java using interfaces.

* Hybrid Inheritance:

Hybrid inheritance is the combination of both Single and Multiple inheritance. It is not directly supported in Java only through interfaces we can achieve this.



Note:-

A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

Examples:-

* Single Inheritance Example

```
class A
{
  void dispA()
  {
   System.out.println("disp method of class A");
  }
}
class B extends A
{
 void dispB()
 {
  System.out.println("disp method of class B");
 }
 public static void main(String args[])
 {
   B b = new B();
   b.dispA(); //Call dispA() method of class A
   b.dispB(); //Call dispB() method of class B
 }
}
```

output:- javac B.java
java B
disp method of class A
disp method of class B

* Multilevel Inheritance Example

```
class A
{
 void dispA()
 {
  System.out.println("disp method of class A");
 }
}
Class B extends A
{
```

```
  void dispB()
  {
   System.out.println("disp method of class B");
  }
}
class C extends B
{
  void dispC()
  {
   System.out.println("disp method of class C");
  }
  public static void main(String args[])
  {
    C c = new C();
    c.dispA(); // call dispA() of class A
    c.dispB(); // call dispB() of class B
    c.dispC(); // call dispC() of class C
  }
}
```

output:- Javac C.Java

Java C

disp method of class A

disp method of class B

disp method of class C

* Hierarchical Inheritance Example

```
class A
{
 public void dispA()
 {
  System.out.println("disp method of class A");
 }
}
class B extends A
{
 public void dispB()
 {
  System.out.println("disp method of class B");
 }
}
```

P.T.O

```
class C extends A
{
 public void dispC()
 {
  System.out.println("disp method of class C");
 }
}
class D extends A
{
 public void dispD()
 {
  System.out.println(" disp method of class D");
 }
}
class HInheritance
{
 public static void main(String args[])
 {
  B b= new B();
  b.dispB();
  b.dispA();
  C c= new C();
  c.dispC();
  c.dispA();
  D d= new D();
  d.dispD();
  d.dispA();
 }
}
```

output:- Javac HInheritance.java

Java HInheritance

disp method of class B
disp method of class A
disp method of class C
disp method of class A
disp method of class D
disp method of class A

* Access Modifiers (or) Member access rules in Java :-

The access modifiers in Java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of access modifiers in Java

1. private
2. default
3. protected
4. public

1. private :-

The private access modifier is accessible only within class.

Example :-

```
class A
{
private int data = 40;
private void msg()
{
 System.out.println("Hello Java");
}
}
public class Simple
{
 public static void main (string args[])
 {
   A obj = new A();
  System.out.println(" obj.data);
  obj.msg();
 }
}
```

output:- Javac Simple.Java
compile Time Error
compile Time Error

In the above example, we have created two classes 'A' and 'Simple'. Class 'A' contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

2. default:-

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package.

Example:-

```
// Save by Adef.java
package pack1;
class Adef
{
 void msg()
 {
  System.out.println("Hello");
 }
}
```

- In This example, we have created two packages pack1 and pack2. We are accessing the Adef class from outside it's package, so it cannot be possible to access from outside the package with default access modifier.

```
// Save by Bdef.java
package pack2;
import pack.*;
class Bdef
{
 public static void main(String args[])
 {
  Adef obj = new Adef(); // compile Time Error
  obj.msg(); // compile Time Error
 }
}
```

output: Javac Bdef.java
Compile Time Error.

Example:-

```
class Adef {
  void msg() { System.out.println("Hello"); }
  }
class Bdef {
  public static void main(String args[])
  {
    Adef obj = new Adef();
    obj.msg();
  }
}
```

output:- Javac Bdef.java
Java Bdef
Hello.

3. protected :-

The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example :-

// save by ProtectA.java

```
package pack;
public class protectA
{
 protected void msg()
 {
  System.out.println("Hello");
 }
}
```

Note :- In compilation, where -d specifies the destination where to put the generated class file.

* We can use . (dot) to keep the package within the same directory.

// save by ProtectB.java

```
package mypack;
import pack.*;
class protectB extends protectA
{
 public static void main(String args[])
 {
  protectB obj = new protectB();
  obj.msg();
 }
}
```

output :- javac -d . protectA.java
javac -d . protectB.java
java mypack.protectB
Hello.

In the above example, we have created the two packages pack and mypack. In the package pack, the msg method is declared as protected, so it can be accessed from outside the class only through inheritance.

4. public :-

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example :-

```
package pack;                          // save by X.java
public class X
{
 public void msg()
 {
  System.out.println(" Hello");
 }
}
```

```
package mypack;
import pack.*;          // save by Y.java
class Y
{
public static void main (String args[])
{
  X obj = new X();
  obj.msg();
 }
}
```

output:- javac -d . X.java
javac -d . Y.java
java mypack.Y
Hello

Let's understand the access modifiers by simple table.

Access Modifier	Within class	Within package	outside package by subclass only	outside package
private	Y	N	N	N
Default	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

Note:- Y - Yes, N - No.

* Super Keyword :-

The super keyword in Java is a reference variable that is used to refer immediate parent class.

The super keyword is used for

- → Accessing the variables of the parent class
- → calling the methods of the parent (or) super class
- → Invoking the constructors of the parent class.

Example :-

Bike.java

```
class Vehicle
{
  int speed = 50;
  void message()
  {
   System.out.println("Welcome to Vehicle class");
  }
}
class Bike extends Vehicle
{
 int speed = 100;
 void message()
 {
  System.out.println("Welcome to Bike class");
 }
 void display()
 {
 System.out.println("Bike speed is : " + speed); // call variable of local
 System.out.println("Vehical Avg speed is : " + super.speed);
                                    // variable of parent class
  message(); // invoke local method
  super.message(); //invoke parent class method.
 }
 public static void main(String args[])
 {
   Bike b = new Bike();
    b.display()
 }
}
```

output:- javac Bike.java

java Bike

Bike speed is : 100

Vehical Avg speed is : 50

Welcome to Bike class

Welcome to Vehicle class

→ The super() is used to invoke the parent class constructor

Example:-

Car.java

```
class Vehicle
{
 Vehicle()
 {
  System.out.println("Vehicle is created");
 }
}
class Car extends Vehicle
{
 Car()
 {
  super(); // invoke parent class constructor.
  System.out.println("Car is created");
 }
 public static void main(String args[])
 {
   Car c = new Car();
 }
}
```

output:-

```
javac Car.java
java Car
Vehicle is created
Car is created
```

* Final Keyword :-

The final Keyword in java is used to restrict the user. The final Keyword can be applied on variables, methods and classes.

The Keyword final is used for the following reasons,

→ The final Keyword can be applied on variables to declare constants.

→ The final Keyword can be applied on methods to disallow method overriding

→ The final Keyword can be applied on classes to prevent (or) disallow inheritance.

* final variable:

If you declare any variable as final, you cannot change the value of final variable (It will be constant).

Example:- Glamour.java

```
class Glamour
{
final int speedlimit=100; // final variable
void run()
{
 Speedlimit = 400;
}
public static void main(String args[])
{
 Glamour obj = new Glamour();
 obj.run();
}
}
```

output:- javac Glamour.java

Compile Time Error:
cannot assign a value to final variable

In the above example, we cannot able change the value of variable speedlimit, because it is declared as final.

* final method:

If you declare any method as final, you cannot override that method. It is called as final method.

Example:- Hero.java

```
class Bike
{
final void run() // final method
{
 System.out.println("running");
}
}
class Hero extends Bike
{
void run()
{
 System.out.println("running safely with 60kmph");
}
```

```
public static void main(String args[])
{
 Hero obj = new Hero();
  obj.run();
 }
}
```

output:- javac Hero.java

Compile Time Error:

overridden method is final

* final class:

If you declare any class as final, you cannot extend it. i.e we cannot inherit that class. It is called as final class.

Example:-

Honda.java

```
final class Bike // final class
{
 void run()
 {
  System.out.println("running safely with 60Kmph");
 }
}
class Honda extends Bike
{
public static void main(string args[])
{
 Honda obj = new Honda();
 obj.run();
}
}
```

output:- javac Honda.java

Compile Time Error:

Cannot Inherit from final Bike

* In simple words, The final keyword in Java

→ stop value change.

→ stop Method overridding.

→ stop Inheritance.

* Object class :-

In java, there is one special class i.e "Object" class. The object class is the parent (or) super class of all the classes in java by default. In otherwords, All other classes are sub-classes of object class.

→ It is the topmost class of java.

→ The object class is helpful if you want to refer any object of any class.

The object class defines following methods, which means that they are available in every object.

* object clone():
  creates a new object that is the same as the object being cloned.
* boolean equals (object object):
  Determines whether one object is equal to another.
* void finalize():
  called before an unused object is recycled.
* class getClass():
  obtains the class of an object at run-time.
* void wait():
  Waits on another thread of execution.
* String toString():
  Returns a string that describes the object.
* int hashCode():
  Returns the hashcode number for this object.
* void notify():
  Resumes execution of thread waiting on the invoking object.

* Polymorphism :-

Polymorphism is a concept by which we can perform a single action by different ways.

polymorphism is derived from two greek words : "poly" and "morphs". The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java, those are compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

The run time polymorphism can be done by method overriding in Java.

* Method Overriding :-

If child class has the same method as declared in the parent class, it is known as method overriding.

In other words, if sub class provides the specific implemenation of the method that has been provided by one of its parent class, it is known as method overriding.

→ Method overriding is used to provide specific implemenation of a method that is already provided by its super class.

→ Method overriding is used for runtime polymorphism.

Rules for method overriding

→ Method must have same name as it's in the parent class.

→ Method must have same parameters as in the parent class.

→ There must be Is-A relationship (Inheritance) between parent and child classes.

Example :-

Bike2.java

```
class Vehicle
{
 void run()
 {
  System.out.println("Vehicle is running");
 }
}
class Bike2 extends Vehicle
{
void run()
{
 System.out.println("Bike is running safely");
}
public static void main(String args[])
{
  Bike2 obj = new Bike2();
  obj.run();
}
}
```

output:- javac Bike2.java
java Bike2
Bike is running safely

In the above example, We have defined the run method in the sub class as defined in the parent class but it has some specific implemenation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

Note:- static (or) final method cannot be overridden.

Note:- Remember when you write two or more classes in a single file, the file will be named upon the name of the class that contains the main method.

* Dynamic Binding :-

Binding is the process of connecting a method call to its body.

There are two types of binding

1. Static binding (early binding)
2. Dynamic binding (late binding)

1. Static binding:

When binding is performed before a program is executed i.e at compile time is called as static binding (or) early binding.

The static binding is performed by compiler when we overload methods.

i.e when multiple methods with the same name exists with in a class (i.e Method overloading) which method will be executed depends upon the arguments passed to the method. So, This binding can be resolved by the compiler.

Example :-

```
class Animal
{
 void eat() //Method without arguments
 {
  System.out.println("animal is eating");
 }
 void eat(String food) // Method with String argument
 { System.out.println("dog is eating..." +food);
 }
 public static void main(String args[])
 {
   Animal a = new Animal();
   a.eat();
   a.eat("Biscuits");
 }
}
```

o/p: javac Animal.java
java Animal
animal is eating
dog is eating... Biscuits

2. Dynamic Binding:

When binding is performed at the time of execution i.e at runtime is called as dynamic binding (or) late binding.

The dynamic binding is performed by JVM (Java virtual Machine) when the overridden methods.

i.e When a method with the same name and signature exists in superclass as well as subclass (i.e Method overriding) Which method will be executed (superclass version or sub class version) will be determined by the type of object. The objects exists at runtime. So this binding is done by JVM.

Example:

```
class Animal
{
 void eat()
 {
  System.out.println(" Animal is eating...");
 }
}
class Dog extends Animal
{
 void eat()
 {
  System.out.println(" dog is eating...");
 }
 public static void main(String args[])
 {
  Animal a = new Dog();
  a.eat(); //call method in child class
  Animal al = new Animal();
  al.eat(); //call method in parent class
  Dog d= new Dog();
  d.eat(); //call method in child
 }
}
```

O/p: javac Dog.java
java Dog
dog is eating...
Animal is eating...
dog is eating...

* Abstract class :-

Abstraction is a process of hiding the implemenation details and showing only functionality to the user.

In another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. you don't know the internal processing about the message delivery.

Def^n:- A class that is declared with abstract keyword, is known as abstract class in Java. It contains one or more abstract methods.

→ An abstract class can have abstract and non-abstract methods.

Abstract Method:

A method that is declared as abstract and does not have implemenlation is known as abstract method.

(or)

An abstract method is a method that is declared with no body or implementation.

example:- abstract void run();
abstract void display();

Example for abstract class :-

Ex1:-
```
abstract class Bike
{
  abstract void run();
}
```

Ex2:-
```
abstract class Animal
{
  abstract void sound();
} abstract void eat();
```

Example :-

```
abstract class Animal //abstract class
{
abstract void sound(); //abstract Method
void eat(String food) // normal method
{
System.out.println("this animal likes "+food);
}
}
class Lion extends Animal
{
void sound()
{
System.out.println("Lions Roar! Roar!");
}
public static void main(String args[])
{
Lion l=new Lion();
l.sound();
l.eat("flesh");
}
}
```

output:-
```
javac Lion.java
java Lion
Lion Roar! Roar!
this animal likes flesh
```

→ The abstract classes cannot be instantiated, and they require subclasses to provide implementation for their abstract methods by overriding them and then the subclasses can be instantiated.

→ Abstract classes contain one or more abstract methods. It does not make any sense to create an abstract class without abstract methods, but if done, the Java compiler does not complain about it.

→ An abstract class can have data member, abstract method, method body, constructor and even main() method.

* Package :-

A package is a group of classes, interfaces and sub-packages.

packages are used in java, in-order to avoid name conflicts and to control access of class, interface and enumeration and etc. using package it becomes easier to locate the related classes.

package are categorized into two forms

→ Built-in package

→ user-defined package.

* There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql and etc.

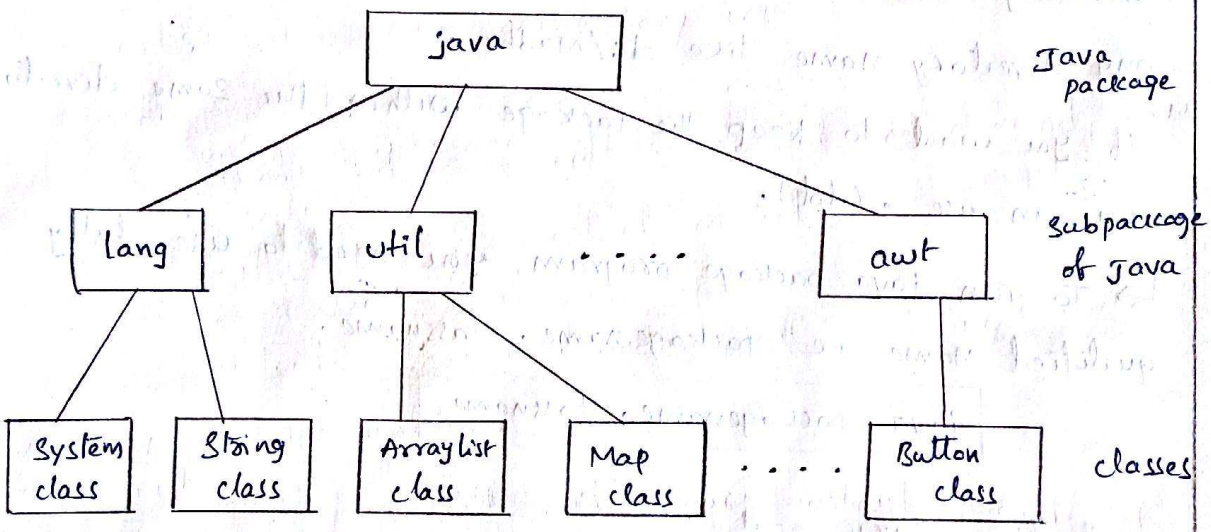


fig:- Built-in package

In the above figure, java is main package and it has many sub packages (lang, util, io, net, ... awt). And each subpackage has several classes (system, string, map ... Button class).

* In java, the user can able to create package by using a keyword i.e "package". The package keyword is used to create a package in java.

```
package <package name>;
```

example :- Simple.java

```
package mypack;
public class Simple
{
public static void main(String args[])
{
System.out.println("Welcome to package");
}
}
```

↳ To compile java package, you need to follow the syntax

```
javac -d directory javafilename
```

In the above syntax, The -d specifies the destination (or) directory where to put the generated class file. you can say any directory name like d:/madhu.

If you want to keep the package within the same directory, you can use . (dot).

↳ To run java package program, you need to use fully qualified name. i.e packagename.classname.

```
java packagename.classname
```

output:- javac -d . Simple.java

java mypack.Simple

Welcome to package.

Note:- If we declare package in source file, the package declaration must be the first statement in the source file.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. using packages, it is easier to provide access control and it is also easier to locate the related classes.

* Importing packages :-

The import keyword is used to import built-in and user-defined packages into your java source file. so that your class can import to a class that is in another package by directly using its name.

There are three ways to access the package from outside the package.

1. using fully qualified name
2. import the only class you want to use.
3. import all the classes from the particular package.

1. using fully qualified name :

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g java.util and java.sql packages contains 'Date' class.

Example :-

A.java

```
package pack;
public class A
{
public void msg()
{
  System.out.println("Hello");
}
}
```

B.java

```
package mypack;
class B
{
public static void main(String args[])
{
pack.A obj = new pack.A(); // using fully qualified name
obj.msg();
}
}
```

output:- javac -d . A.java
javac -d . B.java
java mypack.B
Hello.

2. using packagename.classname (or) import the only class you want to use:

If you import package.classname then only declared class of this package will be accessible.

Example:-

X.java

```
package pack;
public class X
{
 public void msg()
 {
  System.out.println("Hello");
 }
}
```

* In this, we can import only class that you want to use. i.e we can able to access that class in another package.

Y.java

```
package mypack;
import pack.X;
class Y
{
public static void main(String args[])
{
```

```
    X obj = new X();
    obj.msg();
  }
}
```

output:- javac -d . X.java
javac -d . Y.java
java mypack.Y
Hello.

3. Import all the classes from the particular package (or) using packagename.* :-

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example:-

Humans.java

```
package animals;
class Animals
{
public void display()
{
 System.out.println(" All are the animals in the world");
}
}
public class Humans extends Animals
{
public void msg()
{
 System.out.println (" class A animals are Humans");
}
public void msgBC)
{
 System.out.println(" Humans are animals with intelligence");
}
}
```

World.java

```
package world;
import animals.*;
class World
{
 public static void main(String args[])
 {
  Humans obj = new Humans();
  obj.display();
  obj.msg();
  obj.msgB();
 }
}
```

output:- javac -d . Humans.java
javac -d . World.java
java world.World
All are the animals in the World
Class A animals are Humans
Humans are animals with intelligence.

Note:- If you import a package, subpackages will not be imported. i.e If you import a package, all the classes and interfaces of that package will be imported excluding the classes and interfaces of subpackages. Hence, you need to import the subpackages as well.

Note:- Sequence of the program must be package then import then class. i.e import statement is kept after the package statement.

Ex: package statement
import statement
class statement
:
} This is sequence in java program.

Rule:- There can be only one public class in java source file and it must be saved by the public class name.

* Subpackage :-

package inside the package is called the subpackage. It should be created to categorize the package further.

The standard of defining subpackage is packagename.subpackagename. for example "pack.subpack".

Example :-

Simple.java

```
package pack.subpack;
class Simple
{
public static void main (String args[])
{
System.out.println ("Hello subpackage");
}
}
```

output:- javac -d . Simple.java

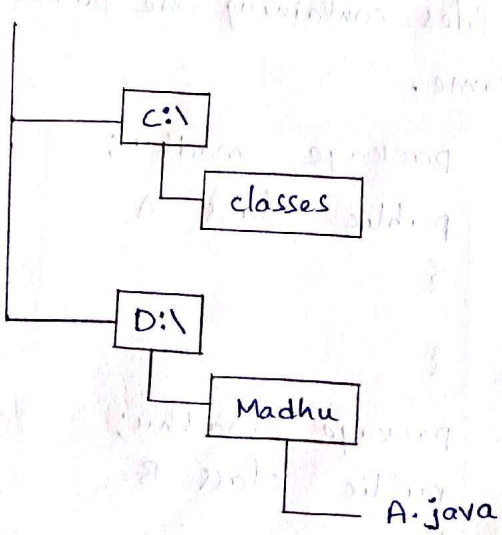
java pack.subpack.Simple

Hello subpackage.

* Setting CLASSPATH :-

If you want to save (or) put Java source files and class files in different directories & drives then we need to set classpath to run or execute those class files.

for example:



Now, I want to put the class file of A.java source file in the folder of C: drive.

Example:-

S.java

```
package pack;
public class S
{
public static void main (String args[])
{
System.out.println("This is example of setting classpath");
}
}
```

To compile:

```
D:\Madhu > javac -d c:\classes S.java
```

To Run:

To run This program from D:\Madhu directory, you need to set classpath of the directory where the class file resides.

```
D:\Madhu > set classpath = c:\classes;.;
D:\Madhu > java pack.S
This is example of setting classpath.
```

* How to put two public classes in a package :-

In java, There can be only one public class in a package. If you want to put two public classes in a single package, have two java source files containing one public class, but keep the package name same.

for example :

```
package madhu;
public class A
{
  =
}
```

// Save as A.java

```
package madhu;
public class B
{
  =
}
```

// Save as B.java