* 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 its 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 we 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 implementation 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 implementation 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.

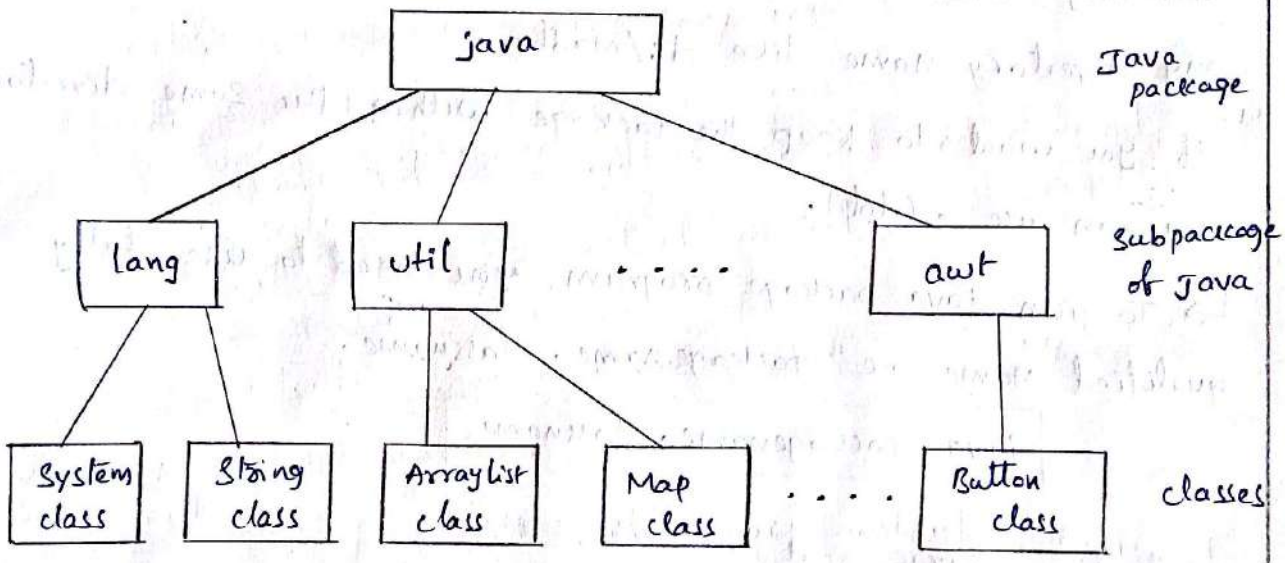


fig:- Built-in package

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.

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

Example:-

X.java

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

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

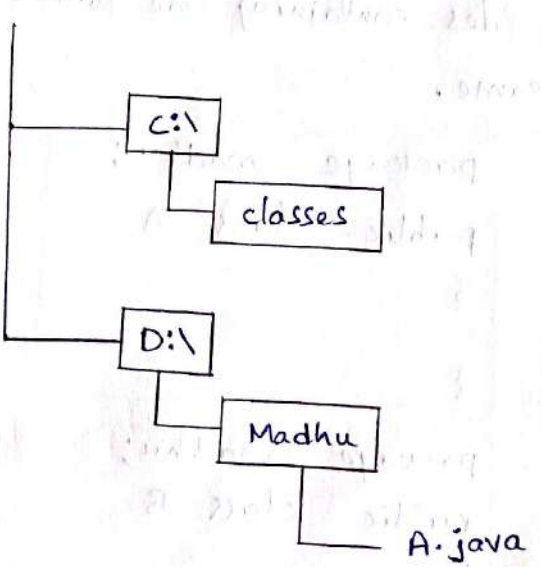
java pack.subpack.Simple

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

* Interface :-

An interface is a collection of abstract methods which are public in scope.

(or)

It is a collection of methods, which are public and abstract by default.

Abstract method :-

It is a method with no body (or) implemenation, ie there is no code at all associated with method.

↳ The best part of an interface is that a class can inherit any number of interfaces, thus allowing multiple inheritance in java.

↳ Java does not support multiple inheritance among classes, but interfaces allow java to support this feature.

→ Interfaces are declared with help of keyword "Interface".

Note :- In interface, None of methods have body.

Syntax :-

```
interface interfacename
{
returntype methodname(arguments);
=
=
}
```

* In simple words, Interface is a blueprint of class, it has static constants and abstract methods.

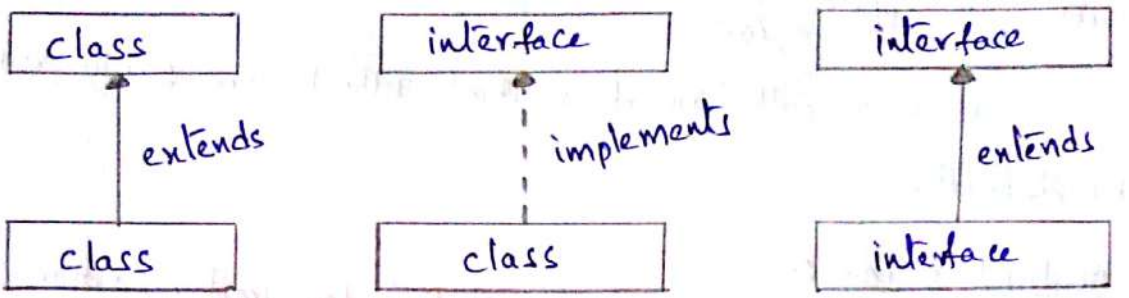
* The interface in java is a mechanism,

↳ to achieve abstraction

↳ to achieve Multiple inheritance

Note :- It is mandatory to add the access specifier public to the method declaration, otherwise the compiler will not compile the program (The older version of Java, i.e before Java 8).

* In Java, a class extends another class, an interface extends another interface but a class implements an interface.



Example:-

Calculator.java

```
interface CalInterface
{
 int add (int a, int b);
 int sub (int a, int b);
}
class Calculator implements CalInterface
{
 public int add (int a, int b)
 {
  return a+b;
 }
 public int sub (int a, int b)
 {
  return a-b;
 }
public static void main(String args [])
{
 Calculator cal = new Calculator();
 System.out.println ("value after addition =" + cal.add (5,2));
 System.out.println ("value after substraction =" + cal.sub(5,2));
 }
}
```

output:- javac Calculator.java
java Calculator
value after addition = 7
value after substraction = 3

* Extending interfaces :- (or) Inheritance in interfaces

Just like inheritance in classes, the interfaces can also be extended. An interface can inherit another interface using the same Keyword "extends".

Example :-

InherDemo.java

```
interface A
{
 void showA();
}
interface B extends A
{
 void showB();
}
class InherDemo implements B
{
public void showA()
{
 System.out.println ("Method of interface A");
}
public void showB()
{
 System.out.println ("Method of interface B");
}
public static void main (String args[])
{
 InherDemo d = new InherDemo();
 d.showA();
 d.showB();
}
}
```

output:- javac InherDemo.java

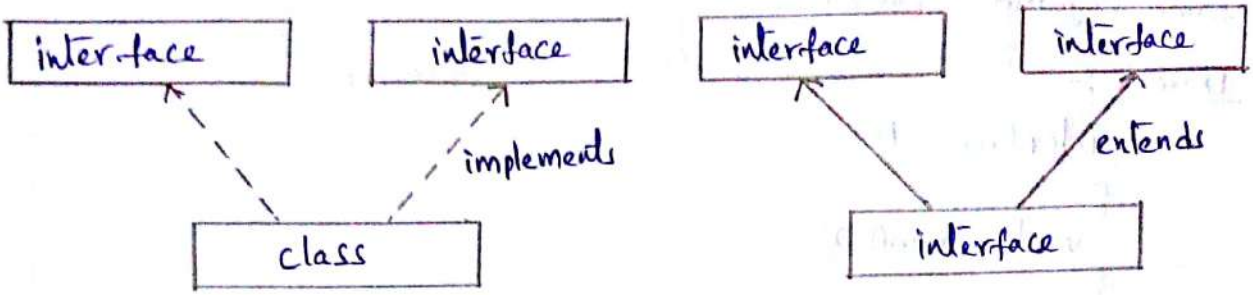
java InherDemo

Method of interface A

Method of interface B

* Multiple inheritance in java by interface :-

If a class implements multiple interfaces (or) an interface extends multiple interfaces is known as Multiple inheritance.



Example :- MultipleInheritanceDemo.java

```
interface printable
{
  void print();
}
interface Showable
{
  void show();
}
class MultipleInheritanceDemo implements printable, Showable
{
public void print()
{
 System.out.println("Hai");
}
public void show()
{
 System.out.println("Welcome");
}
public static void main(String args[])
{
MultipleInheritanceDemo obj = new MultipleInheritanceDemo();
  obj.print();
  obj.show();
}
}
```

output :- javac MultipleInheritanceDemo.java
java MultipleInheritance
Hai
Welcome.