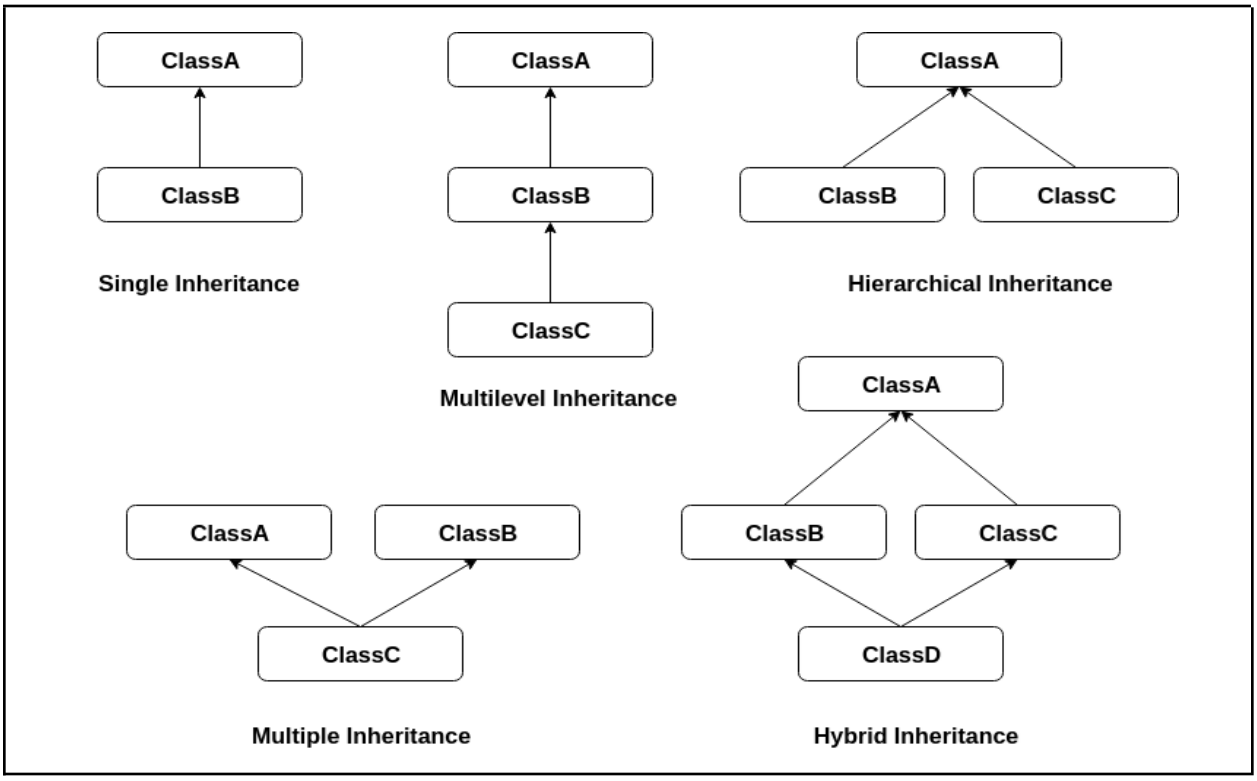
Contents:
Inheritance – Types of Inheritance – Superclass and subclass – calling super class constructor and methods – overloading vs overriding - Final variables, final methods and final classes – Abstract methods and class – polymorphism – dynamic binding – Visibility controls.
Interfaces – Definition – Extending, Implementing and accessing Interfaces- packages – Creating and Accessing Packages.

INHERITANCE

- Inheritance is one of the importance concept of OOP which provides **reusability** of properties and behavior of already existing objects
- The mechanism of deriving a new class from an old one is called ***Inheritance***
- The old class is known as the ***base class*** or ***super class*** or ***parent class*** and the new class is known as **derived class** or **subclass** or **child class**
- The inheritance allows subclasses to inherit all the variables and methods of their parent classes

Different forms of Inheritance



- **Single Inheritance** - Only one superclass
- **Multiple Inheritance** - More than one super classes to a subclass
- **Hierarchical inheritance** - One super class, many subclasses
- **Multilevel Inheritance** - A class is derived class from another derived class
- **Hybrid Inheritance** - It is a combination of other forms of inheritance

Note:- Java doesn't directly implement multiple inheritance. However, multiple inheritance is implemented using the concept of ***interface***

Syntax of Defining a subclass

```
class  subclassname  extends  superclassname
{
        Variable declarations;
        Methods declarations;
}
```

- The keyword **extends** is used to inherit the super class properties to sub class
- The subclass can get all the members of superclass through inheritance and it can still have its own members
- Sub classes are extended versions of super classes. Super classes will stay as it is without any modifications

super keyword

- It is a special keyword that can be used from a subclass for two purposes:
    1. To call super class constructor from subclass constructor
       super (parameter-list);
    2. To access super class members from subclass methods
       super.member

Constructors and Inheritance

- Both subclass and superclass can have their own constructors
- Superclass constructor constructs the superclass portion of the object. Subclass constructor constructs the subclass part.
- Subclass constructor uses the keyword **super** to call the superclass constructor

super (parameter-list);

- Parameter-list should match with the parameters of super class constructor

- **super()** call must be the **first line** of subclass constructor

Single Inheritance

- **Only one base class for a single derived class**

```
class  A
{
    ……
}
class B extends A
{
     ……
}
```

Example Program:

Question: Define a base class **Rectangle** which contains the following members:

Instance variables - **length, breadth**

Methods - **parameterised constructor, area(), displayLB()**

Derive the subclass **Box** extending the class Room. the sub class contains following additional members:

Instance variable - **height**

Methods - **parameterised constructor, volume(), display( )**

Write **main()** method to implement the above class hierarchy

```
//Single Inheritance
class Rectangle   // Super class
{
   int length, breadth;
   Rectangle(int x, int y)
   { length = x; breadth = y; }
   void displayLB(){
      System.out.println("Length="+length);
      System.out.println("Breadth="+breadth);}
   int area()
   { return length*breadth; }
}
```

```
class Box extends Rectangle   // Subclass
{
   int height;
   Box(int x, int y, int z)
   {
      super(x,y);        height=z;
   }
   int volume()
   {
      return super.area()*height;
   }
   void display()
   {
      super.displayLB();
      System.out.println("Height ="+height);
      System.out.println("Base Area ="+ super.area());
      System.out.println("Volume = "+ volume());
   }
}
```

```
class DemoSingleInheritance   // main class
{
        public static void main(String[] args) {
                Box b1=new Box(10, 15, 50);
                b1.display();
        }
}
```

Multi-level Inheritance

- A class is derived from another derived class

```
class  A
{
    ……
}
class  B   extends  A
{
    ……
}
class C  extends  B
{
   ……
}
```

Example Program

Question:-

Define a parent class **Student** with following members:

Instance variables - regno, name

Methods - constructor, display1()

Define the sub class **Mark** that derives Student with following members:

Instance variables - mark1, mark2, mark3

Methods - constructor, displayMarks(), totalmark()

Define next level subclass **Result** that derives **Mark** with following members:

Instance variable - nil

Methods - constructor, grade(), marklist()

Write main() function to create Result object and print the mark list of student

```
// Sample program illustrating multi-level inheritance (Refer Lab Exercises)
```

Hierarchical Inheritance

- More than one derived classes from a single parent class

```
class A
{
    ……..
}
class B extends A  //B is derived from A
{
     ……
}
class C extends A  // C is also derived from A
{
   ……
}
```

Sample Program

```
//Hierarchical Inheritance
class Base{
   void showBase()
   {
      System.out.println("Common base class");
   }
```

Output

```
Common base class
Derived Class 1
Common base class
Derived Class 2
```

```
}
class Derived1 extends Base
{
   void showDerived1()
   {
      System.out.println("Derived Class 1");
   }
}
class Derived2 extends Base
{
   void showDerived2()
   {
      System.out.println("Derived Class 2");
   }
}
public class HierarchicalDemo
{
        public static void main(String[] args) {
                Derived1 d1 = new Derived1();
                Derived2 d2 = new Derived2();
                d1.showBase();
                d1.showDerived1();
                d2.showBase();
                d2.showDerived2();
        }
}
```

Visibility Modifiers (Access Modifiers)

The access modifiers in Java specify the accessibility or scope of a field, method, constructor, or class.

There are mainly **four types** of Java access modifiers:

1. **Private**: The access level of a private modifier is **only within the class**. It cannot be accessed from outside the class.
2. **Default or friendly**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through subclass.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Note:- It is possible to use two keywords **private** and **protected** together as well. This gives a visibility between protected access & private access(Java originally had such a modifier. It was written private protected but removed in Java 1.0.)

	public	**protected**	**default (friendly)**	**Private protected**	**private**
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other class in same package	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	**Yes**	No
Non-subclasses in other package	Yes	No	No	No	No

METHOD OVERRIDING

- It is the process of **redefining the super class method in the sub class**.
- Signature of the method (return type, name, parameters) should be the same as that in the super class. Only the definition of the method will be specific in the subclass.
- Method overriding is possible in an **inheritance** hierarchy of classes
- One of the important **use of method overriding is to implement runtime polymorphism** (dynamic method dispatch)

Example:- Overriding the method **show()** in the subclass B which is initially defined in the superclass A

```
class A
{
      int  x;
      A(int a)
      {  x = a; }
      void  show()
       {
            System.out.println(“ show method in superclass A”);
```

```
            System.out.println("x = " + x);
        }
}
class B extends A
{
      int  y;
      B( int m, int n  )
      { super(m);  y = n; }
      void  show()  // overriding superclass method show()
       {
            System.out.println(" show method in subclass B");
            System.out.println("x = " + x);
            System.out.println("y = " + y);
       }
}
class MainShow
        public static void main(String[] args)
        {
                    B  objb = new B(10,15);
                    objb.show( ); /* by default subclass version of show()
                                     will be  invoked */
        }
}
```

Implementing Dynamic Polymorphism (dynamic method dispatch)

- To implement dynamic polymorphism, we create:
  - the object(s) of derived class(es) and
  - a null reference of base class.
- It is possible to assign derived class object's reference to base class reference.
- At run time, the version of the overridden method pointed to by the base class reference will be executed.
- In other words, multiple versions of a method having the same signature can be binded with an object and at a time one of the versions can be chosen and executed. Since this selection occurs at runtime, it is known as runtime polymorphism.

Example: sample program for implementing dynamic polymorphism using method overriding.

```
// Dynamic method dispatch using Method Overriding - runtime polymorphism
class Employee
{
   int empID;
   String ename;

   Employee(int x, String y)
   {  empID=x; ename=y; }

   void display()
   {
      System.out.println("Display from Super Class - Employee");
      System.out.println("************************************");
      System.out.println("Employee ID :"+empID);
      System.out.println("Name :"+ename);
   }
}
class Engineer extends Employee
{
   String branch;
   Engineer(int x, String n, String b)
   {
      super(x,n);
      branch=b;
   }
   void display() // Overridden method
   {
      System.out.println("Display from Sub Class - Engineer");
      System.out.println("************************************");
      System.out.println("Employee ID :"+empID);
      System.out.println("Name :"+ename);
      System.out.println("Engineering Branch :"+branch);
   }
}
class DemoMethodOverriding
{
public static void main(String[] args) {
         Employee obj1; // Obtain a null reference of Employee
         Employee obj2=new Employee(100,"Latha"); // Object of Employee
         Engineer obj3=new Engineer(101,"Hari","Mechanical"); // Object of Engineer
         obj1=obj2;   // Super class reference points to a superclass object
         obj1.display(); // Prints superclass version of display()
         obj1=obj3; // super class reference points to sub class object
         obj1.display(); // Prints subclass version of display()
}
}
```

```
Display from Super Class - Employee
**********************************
Employee ID :100
Name :Latha
Display from Sub Class - Engineer
**********************************
Employee ID :101
Name :Hari
Engineering Branch :Mechanical
```

Overloading Versus Overriding

Method Overloading	Method Overriding
It implements ***compile-time polymorphism***	It is used to implements ***run-time polymorphism which is a dynamic binding process***
Here, a class consists of more than one methods with same name but difference in parameter list (either the number of parameters or the data type of parameters will be different)	Method overriding is redefining superclass methods in subclass. Here , both superclass and subclass will have methods with the same signature (both name and parameters will be the same) but subclass implements a different definition.

Final Variables

- It is possible to define variables as final variables using the keyword final

```
final   data_ype   variable_name = value;
```

- Final variables are like constant variables.
- A constant value can be assigned to final variables during declaration or from the constructor.
- We cannot change their values anywhere in the program.
- Final variables are like class variables. They do not take any space on individual objects of the class

Sample program with a final variable.

```
1  //final variables are like constants, we cannot change its value
2
3  class A
4  {
5      final int x=80;
6      int y;
7      A(int p, int q)
8      {
9          x=p;   // error: cannot assign a value to final variable x
10         y=q;
11     }
12     void display()
13     {
14         x=x+1; //error: cannot assign a value to final variable x
15         System.out.println("x="+x+"y="+y);}
16 }
17 public class Main
18 {
19     public static void main(String[] args) {
20         A obja=new A(25,60);
21         obja.display();
22     }
23 }
```

In the above program, line number 9 and line number 11 shows error because both statements tries to change the value of already defined final variable **x**

Final methods

- Like variables, we can define methods as final methods
- Final methods of superclass cannot be overridden from the subclass. That means their definition cannot be modified from subclass.
- To define a method as final, just write the keyword final in front of it's signature

```
final  return_type  method_name(parameter-list)
{
   // Body of the method
}
```

Sample program that tries to override a final method in subclass and generates the error message:

```java
//final methods cannot be overridden
class A {
    int x,y;

    A(int p, int q)
    { x=p;  y=q; }

    final void display()
    { System.out.println("x="+x+"y="+y); }
}

class B extends A {
    int z;

    B(int a, int b, int c)
    { super(a,b); z=c; }

    void display() // error: display() in B cannot override display() in A
    { System.out.println("x="+x+"y="+y+"z="+z); }
}

public class Main
{
    public static void main(String[] args) {
        B objb=new B(25,60,75);
        objb.display();
    }
}
```

Final class

- In Java, it is possible to prevent some classes from being extended as subclasses due to security reasons.
- A class that cannot be subclassed is called a final class
- This is achieved in java by using the keyword final as follows:

```
final  class  class_name1
{
      // Body of class definition
}
```

Sample program showing error while trying to extend a final class

```java
//final class cannot be extended to create a subclass
final class A {
    int x,y;

    A(int p, int q)
    { x=p;  y=q; }

    void display()
    { System.out.println("x="+x+"y="+y); }
}

class B extends A  // error: cannot inherit from final A
{
    int z;

    B(int a, int b, int c)
    { super(a,b); z=c; }

    void display() // error: display() in B cannot override display() in A
    { System.out.println("x="+x+"y="+y+"z="+z); }
}

public class Main
{
    public static void main(String[] args) {
        B objb=new B(25,60,75);
        objb.display();
    }
}
```

ABSTRACT METHODS AND CLASSES

- In Java, we can indicate that a method must be redefined in the subclass, thus making overriding compulsory. It is something opposite to that of final methods.

- Abstract classes and abstract methods are defined by using the keyword **abstract** as shown below:

```
abstract  class class_name
{
.....................
.....................
abstract  return_type  method_name(parameter-list);
```

```
……..
}
```

- A class containing one or more abstract methods must be defined as an abstract class.

Rules for abstract method

1. We cannot use abstract class to instantiate objects directly (i.e. we cannot create objects of abstract classes)
2. The abstract methods of an abstract class must be defined in its subclass
3. Constructors or static methods cannot be declared as abstract

Interfaces

- Java does not support multiple inheritance directly by using the concept of class
- But, java provides an alternate approach known as '**interface**' to support the concept of multiple inheritance
- It is basically a kind of class that contain methods and variables
- **The difference is that interface define only abstract methods and final fields**
- Since abstract methods have no definitions, it is the responsibility of the class that implements the interface to define the abstract methods.

Syntax of interface definition

```
interface interface_name
{
    Variable declarations;
    Method declarations;
}
```

Variables are declared as :

```
static  final  data-type  variable_name = value;
```

Methods are declared as :

```
return_type  method_name(parameter list);
```

Example:

```
interface  Area
{
      final static float pi=3.14;
      float  compute( float x, float y );
      void show();
}
```

Difference between class & interface

	class	interface
1	Variables (data) of a class can be constant or variables	variables of an interface are always declared as constant . Their values are final
2	The methods of a class can be abstract or non-abstract	The methods in an interface are always abstract. . It is later defined in the class that implements the interface
3	We can create objects of class (i.e. class can be instantiated)	It is not possible to create objects of interface. It can only be inherited
4	Class can use various access specifiers like public, private or protected	It can only use the public access specifier

Extending interfaces

- It is possible to define a new interface by extending one or more existing interfaces
- The new interface will inherit all the members of the superinterfaces

Syntax

```
interface   new_interface   extends   super_interface1, super_interface2, ….
{
        // body of new_interface
}
```

Example

```
interface   ItemConstants     //  An interface that contain constant variables
{
        int  code = 101;
        String  name= "Pen";
}
interface   ItemMethods  // An interface that contains abstract methods
{
       void  display();
}
Interface  Item  extends  ItemConstants , ItemMethods
{
    ……
    ……
}
```

Notes:-

1. Since all the variables defined inside an interface are treated as final, it is not compulsory to write the keywords **final** and **static**.
2. An interface cannot extend classes

Implementing interfaces

- Interfaces are used as superclasses whose properties are inherited by classes
- The keyword "**implements**" is used to inherit an interface into a class

Basic Syntax

```
class  class_name   implements   interface_name
{
    // Body of the class
}
```

- Here, the class ***"class_name"*** implements the interface ***"interface_name"***

More General Syntax

```
class  subclass_name   extends super_class implements   interface_name1,
interface_name2, ……
{
    // Body of the class
}
```

- **This shows that a class can extend another class while implementing interfaces.**
- **This syntax is used for multiple inheritance**

Various forms of interface implementation

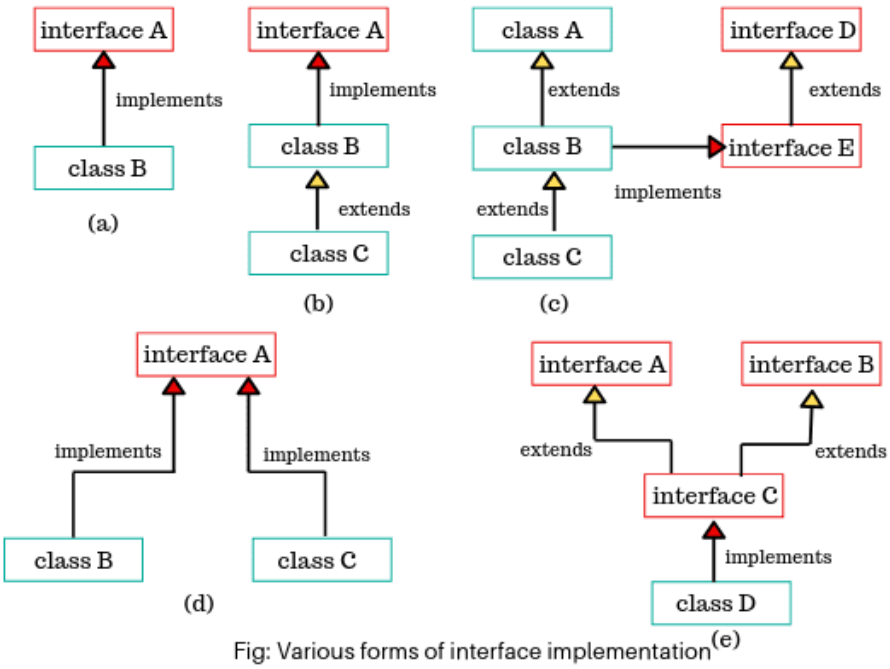


Fig: Various forms of interface implementation

Your Exercise :- Write Java code block for various forms of inheritance shown above

Sample program for multiple inheritance

```
interface Home
{
        void homeLoan();
 }
interface Car
{
        void carLoan();
}
interface Education
{
        void educationLoan();
 }
class Loan implements Home, Car, Education
{
// Multiple inheritance using multiple interfaces.
        public void homeLoan()
        {
```

```
            System.out.println("Rate of interest on home loan is 10%");
        }
        public void carLoan()
        {
            System.out.println("Rate of interest on car loan is 12.5%");
        }
        public void educationLoan()
        {
            System.out.println("Rate of interest on education loan is 9.25%");
        }
}
public class MainLoan
{
        public static void main(String[] args)
        {
            Loan l = new Loan();
            l.homeLoan();
            l.carLoan();
            l.educationLoan();
        }
}
```

Structure of a Java program for multiple inheritance that extends a super class and implements an interface

```
class A  // super class
{
   ……….
}
interface  X  // Interface
{
   ……..
}
class B   extends A implements X   // B is the subclass with multiple inheritance
{
    ……..
}
```

Sample Program (Student result with sports weightage - refer lab record)

Accessing Interface variables

Interface variables can be used from methods of any class in which it is implemented

```
interface ConstantValues
{
// Declaration of interface variables.
            int x = 20;
            int y = 30;
 }
class Add implements ConstantValues
{
           int a = x;
           int b = y;
          void m1()
          {
           System.out.println("Value of a: " +a);
           System.out.println("Value of b: " +b);
          }
         void sum()
         {
          int s = x + y;
          System.out.println("Sum: " +s);
         }
}
class Sub implements ConstantValues
{
          void sub()
          {
           int p = y - x;
           System.out.println("Sub: " +p);
          }
}
public class Test
{
          public static void main(String[] args)
          {
                    Add a = new Add();
                     a.m1();
                     a.sum();

                    Sub s = new Sub();
                     s.sub();
           }
}
```

Packages in Java

- In Java, related classes and/or interfaces can be grouped together to form packages.
- Packages are Java's way of grouping classes and/or interfaces.
- Package provides a means by which classes can be encapsulated

Types of packages

- Java packages are classified into two types
    a. **Built-in API packages** (**java.lang, java.util, java.io, java.awt, java.net, java.applet** etc.)
    b. User defined packages

Defining or Creating a User Defined package

- To create a package, use the **package statement** at the top of the java source file.
- A class declared within that file will belong to the specified package

```
package  package_name;

Eg:-
package  mypackage;
```

- Java uses the file system to manage packages
- The **.class** files for any classes in a package "**mypackage**" must be stored in a directory called **mypackage**
- Since Java is case sensitive, remember to create the package directory carefully . Often, lower case is used for package names.

Steps for creating & accessing user defined packages

- Following are the steps to create our own packages:
    1. Declare the package at the beginning of our source program file as shown below:

```
package  package_name;
```

    2. Define the class that is to be included in the package and declare it **public**
    3. Create a subdirectory under the directory where the main source files are stored
    4. Store the listing as **classname.java** in the subdirectory created
    5. Compile the file to create **.class** file in the subdirectory
- ***Note that only one class can be public in a Java source file***

Hierarchy of Packages

- **It is possible to build a package hierarchy**
- Of course, you must create directories that support the hierarchy you create
- **The general form of a multileveled package statement is shown below:**

```
package  pack1.pack2.pack3…..packN;

Eg:-
package  alpha.beta.gamma;
```

The **.class** files must be stored in …./alpha/beta/gamma

Finding packages and CLASSPATH

- There are **three ways** for the Java run-time system to locate packages
    1. Java run-time system will take the **current directory** as starting point and will take the package if it is stored in a subdirectory under the current directory
    2. You can specify a directory path or paths by setting the **CLASSPATH** environmental variable
    3. Use **-classpath** option with **java** and **javac** to specify the path to your classes

Packages and Member Access

- The visibility of an element is determined by its access specification - **private, public, protected** , or **default** - and the package in which it resides
- The visibility of an element is determined by its visibility within a class and its visibility within a package

	Private Member	**Default** Member	**Protected** Member	**Public** Member
Visible within same class	Yes	Yes	Yes	Yes
Visible within same package by subclass	No	Yes	Yes	Yes
Visible within same package by non-subclass	No	Yes	Yes	Yes
Visible within different package by subclass	No	No	Yes	Yes
Visible within different package by non-subclass	No	No	No	Yes

Thumb rules for visibility

- ***Public members are visible everywhere, in different classes and different packages***
- ***A private member is accessible only to the other members of its class***
- ***A protected member is accessible within its package and to all subclasses, including subclasses in other packages***

Accessing Packages (import statement)

- To access classes from other packages, either use a fully qualified class name or use **import** statement at the beginning of our source code

General form of import statement

import package_name.classname; **// to import a specific class from a package**

import package_name.* ; **// To access all classes from a package**

Example: *Accessing the class **ArithmeticException** from **java.lang** package*

java.lang.Exception.ArithmeticException obj; // fully qualified class name

OR

import java.lang.Exception; // first import the class
ArithmeticException obj; // no need to write fully qualified class name

Sample Program for illustration of package in Java

- **In current working directory, Create a folder 'shapes' (top level package)**
- **Second level of package hierarchy is :**
  - **TwoDShapes**
  - **ThreeDShapes**
- **twoDShapes package contains two classes :**
  - **Circle (Circle.Java)**
  - **Rectangle (Rectangle.Java)**
- **threeDShapes package contains two classes :**
  - **Sphere (Sphere.Java)**
  - **Box (Box.Java)**
- **Write a separate Java source file which contains a main() method to access the above created packages**

./shapes			
./shapes/twoDShapes		**./shapes/threeDShapes**	
Circle.Java **Circle.class**	**Rectangle.Java** **Rectangle.class**	**Sphere.Java** **Sphere.class**	**Box.Java** **Box.class**

(Write the programs by yourself)

finalize() method

- In java, we use constructor to create and initialize an object
- To finalize an object, we can use the **finalize()** method. Finalization will clear the memory occupied by the object.
- Java has an automatic Garbage Collecting system. It automatically clears the memory occupied by the objects. But garbage collectors cannot clear resources such as file descriptors, windows system fonts etc. that may be held by the objects. In order to free these resource, we have to use the destructor method **finalize()**
- **finalize()** method can be added to any class. Java calls that method whenever it is about to reclaim the space used by objects