CST281
Object Oriented Programming

MODULE 3
More features of Java

Syllabus

More features of Java:

Inheritance - Super Class, Sub Class, The Keyword super, protected Members, Calling Order of Constructors, Method Overriding, the Object class, Abstract Classes and Methods, Using final with Inheritance.

Packages and Interfaces - Defining Package, CLASSPATH, Access Protection, Importing Packages, Interfaces.

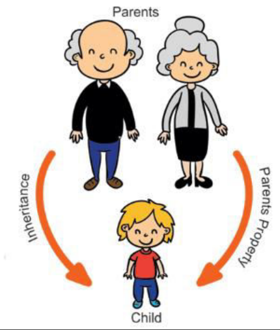
Exception Handling - Checked Exceptions, Unchecked Exceptions, try Block and catch Clause,

Multiple catch Clauses, Nested try Statements, throw, throws and finally.

- Inheritance –
  - Super Class,
  - Sub Class,
  - The Keyword super,
  - protected Members,
  - Calling Order of Constructors,
  - Method Overriding,
  - the Object class,
  - Abstract Classes and Methods,
  - Using final with Inheritance.

Inheritance

- Inheritance helps to create hierarchical classifications.
- Using inheritance we can create a general class(base or super class) that defines features common to a set of related items.
  - This class can then be inherited by other, more specific classes(subclasses).

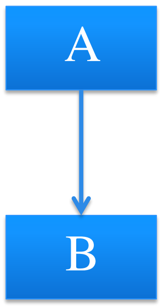


- A subclass is a specialized version of a superclass.
- Subclass inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.
- To inherit a class, we have to use **extends** keyword along with subclass definition.

```
class superclass
{   //statements……}
class subclass extends superclass
{   //statements……}
```

```
// A simple example of inheritance.
class A
{
    int i, j;
    void showij()
    {
    System.out.println("i and j: " + i + " " + j);
    }
}
class B extends  A {
int k;
void showk() {
   System.out.println("k: " + k);
   }
void sum() {
   System.out.println("i+j+k: " + (i+j+k));
        }
}
```

Here A is the superclass of B



```
class A
{
    int i, j;
void showij()
{
System.out.println(i + "  " + j);
}
}
class B extends A
 {
int k;
void showk() {
   System.out.println("k:  " + k);
   }
void sum() {
System.out.println("sum  " + (i+j+k));
         }
}
```

```
class SimpleInheritance
{
public static void main(String args[]) {
A superOb = new A();
B subOb = new B();
superOb.i = 10;
superOb.j = 20;
System.out.println("Superobj Contents ");
superOb.showij();
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("subOb contents ");
subOb.showij();
subOb.showk();
System.out.println("Sum in subOb:");
subOb.sum(); } }
```

```
Superobj
Contents
10 20
subOb contents
7 8
k: 9
Sum in subOb:24
```

Member Access and Inheritance

- Subclass cannot access the private members in superclass.

```
class A {
int i; // public by default
private int j; // private to A
void setj(int x) { j = x; };
}
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j(private) is not accessible here
}  }
```

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.

- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.
- Each subclass can have its own special features also.
- You can only specify one superclass for any subclass that you create.
- Java does not support the inheritance of multiple superclasses into a single subclass.
- You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass.
- However, no class can be a superclass of itself.

A Superclass Variable Can Reference a Subclass Object

- A **reference variable of a superclass** can be assigned a reference to any subclass derived from that superclass.

```
class A
{        }
class B extends A
{        }
class Sample
{
   A oba=new A();
   B obb=new B();
    oba=obb;
}
```

Superclassobject=subclassobject

```
class Parent
{
   int a,b;
   void area()
   {  System.out.println("Product="+ a*b);
    }
}
class Sub extends Parent
{    int i;
    Sub(int x,int y,int z)
    {
       a=x;
       b=y;
       i=z;
    }
}
```

```
class InhRefsub{

public static void main(String args[])
{
  Parent pob=new Parent();
  pob.area();
  Sub subob=new Sub(10,20,30);
  pob=subob;
  pob.area();
//System.out.println("i="+ pob.i);//ERROR
  }
}
```

```
OUTPUT
Product=0
Product=200
```

Using super

- Whenever a subclass needs to refer to its immediate superclass, it can be done using the keyword super.
- super has two general forms.
  1. To call the superclass' constructor.
  2. To access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of super:

super(arg-list);

- Here, arg-list specifies any arguments needed by the constructor in the superclass.
- super( ) must always be the first statement executed inside a subclass' constructor.

```
class Parent
{   Parent()
    {

  System.out.println("Superclass");
}  }
class Sub extends Parent
{
    Sub()
    {
        super();
        System.out.println("Subclass");
    }
}
```

```
class SupersubDemo{
 public static void main(String args[])
 {
     Sub subob=new Sub();
 }
}
```

OUTPUT

```
Superclass
Subclass
```

super keyword to access member

- super always refers to the superclass of the subclass in which it is used.
- To access the member in superclass from subclass

  super.member

  – Here member can be either a method or an instance variable.
- If subclass contains same variable as superclass, then in subclass, the superclass member will be hidden by corresponding subclass member.
  – This can be prevented using super keyword

```
class A {
   int i;
}
class B extends A
{   int i; // this i hides the i in A
   B(int a, int b) {
   super.i = a; // i in A
   i  = b; // i in B
}
void show() {
   System.out.println("i  in superclass:  " + super.i);
   System.out.println("i  in subclass:  " + i);
   }
}
```

```
class UseSuper {
  public static void main(String args[])
  {
     B subOb = new B(1, 2);
      subOb.show();
   }
}
```

OUTPUT

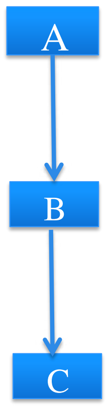
```
i in superclass: 1
i in subclass: 2
```

Creating multiple hierarchy

```
class A
{   int x;
   A(int p)
   {   System.out.println("Superclass  A ");
       x=p;    }
}
class B extends  A
{     int y;
      B(int p,int q)
      {
         super(p);
         System.out.println("B  Subclass  of  A");
         y=q;
      }
}
```

```
Superclass A
B Subclass of A
C Subclass of A
x=10
y=20
z=10
```

```
class C extends B
{  int z;
  C(int p,int q,int r){
   super(p,q);
   System.out.println("C  Subclass  of  A");
    z=r;
}    }
class Multipleinh{
  public static void main(String args[])
  {    C ob=new C(10,20,30);
       System.out.println("x="+ob.x);
       System.out.println("y="+ob.y);
       System.out.println("z="+ob.x);
  }
}
```



- Inheritance –
  - Super Class,
  - Sub Class,
  - The Keyword super,
  - protected Members,
  - Calling Order of Constructors,
  - Method Overriding,
  - the Object class,
  - Abstract Classes and Methods,
  - Using final with Inheritance.

Protected members

- Protected members are declared by prefixing the access specifier protected.

  protected datatype member;
- The protected member in a class can be accessed by
  - any class within the same package.
  - direct sub-classes in other package also.

- If you want to allow an element(member) to be seen outside your current package, but only to classes that subclass your class directly, then declare that element (member) protected.

```
class  A
{
    protected int c;//protected variable
    int a;
    private char b;
    public float f;
    protected void add()//protected method
    {
        //statements
    }
//methods and statements
}
```

Calling Order of Constructors

- Constructors are called in the order of derivation, from superclass to subclass
- When subclass object is created, it first calls superclass constructor then only it calls subclass constructor.
- If super( ) is not used to call superclass constructor, then the default constructor of each superclass will be executed before executing subclass constructors.

```
class A
{  int i;
   A()
   {   System.out.println("Constructor of
    superclass A");
}     }
class B extends  A
{  int j;
   B()
   {   System.out.println("Constructor of
    subclass B");
}  }
```

```
class Consorder
{
  public static void main(String args[])
  {
    B obb =new B();
  }
}
```

OUTPUT

```
Constructor of superclass A
Constructor of subclass B
```

```
class A
{  int i;
   A()
   {   System.out.println("Constructor of
    superclass A");
}  }
class B extends A
{  int j;
   B()
   {
       System.out.println("Constructor of
    subclass B");
}  }
```

```
class C extends B
{  int j;
   C()
   {   System.out.println("Constructor of
    subclass C");
}  }
class Consorder
{  public static void main(String args[])
   {   C obc =new C();  }
}
```

OUTPUT

```
Constructor of superclass A
Constructor of subclass B
Constructor of subclass C
```

- Superclass has no knowledge of any subclass, any initialization it needs to perform is separate and it should be done as a prerequisite to initialize the subclass object.
- Therefore, superclass constructors are executed before executing subclass constructors, when we create subclass object.

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.
- This is called METHOD OVERRIDING
- When an overridden method is called from within a subclass, it will always refer to the method defined by the subclass.
  - The version of the method defined by the superclass will be hidden.

```
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;      }
void show() {
  System.out.println(" i : " + i + " j: " + j);
}  }
class B extends  A {
  int k;
  B(int a, int b, int c) {
    super(a, b);
    k = c;      }
void show() {
     System.out.println("k: " + k);
}    }
```

```
class Override {
public static void main(String args[])
{
  B subOb = new B(1, 2, 3);
  subOb.show(); // this calls show() in B
}    }
```

```
OUTPUT
k: 3
```

When **show( )** is invoked on an object of type B, the version of **show( ) defined within B is used**.
That is, the version of show( ) inside subclass B overrides the version declared in superclass A.

```
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
     i = a;
     j = b;      }
void show() {
  System.out.println(" i : " + i + " j: " + j);
}  }
class B extends  A {
  int k;
  B(int a, int b, int c) {
     super(a, b);
     k = c;      }
void display() {
    System.out.println("k: " + k);
}   }
```

```
class Override {
public static void main(String args[])
{
  B subOb = new B(1, 2, 3);
  subOb.show(); // this calls show() in A
}   }
```

```
OUTPUT
i: 1 j: 2
```

Here when **show( )** is invoked on an object of type B, since the version of **show( ) is not defined within B** the version of show() declared in superclass A is called and excuted.

- To access the superclass version of an overridden method, we can do using super keyword.

```
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;     }
void show() {
  System.out.println(" i : " + i + " j: " + j);
}  }
class B extends A {
  int k;
  B(int a, int b, int c) {
    super(a, b);
    k = c;     }
void show() {
   super.show();
   System.out.println("k: " + k);
}   }
```

```
class Override {
public static void main(String args[])
{
  B subOb = new B(1, 2, 3);
  subOb.show(); // this calls show() in B
}   }
```

```
OUTPUT
i:1 j:2
k: 3
```

When **show( )** is invoked on an object of type B, the version of **show( ) defined within B is used**..

super.show() calls the show() method in its superclas.

- Method overriding occurs only when the names and the type signatures of the methods in subclass and superclass are identical.
- If names and the type signatures of the two methods are different, then the two methods are simply overloaded.

```
class A {
  int i, j;
  A()
  {   i = 0;
      j = 0;   }
  void show()
  {   System.out.println("show in A");
} }
class B extends A {
  int k;
  B()
  {    k = 0;   }
  void show(String msg) {
     System.out.println("show in subclass B");
}  }
```

```
class Sample {
public static void main(String args[]) {
B subOb = new B();
subOb.show("k is "); // this calls show() in B
subOb.show(); // this calls show() in A
}
```

```
OUTPUT
K is show in subclass B
show in A
```

Here show() Methods have differing type signatures. So they are overloaded – **not overridden**

The Object Class

- There is one special class, Object, defined by Java.
- All other classes are subclasses of Object.
- That is, Object is a superclass of all other classes.
- Reference variable of type Object can refer to an object of any other class.

Methods in Object class

Method	Purpose
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.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

- The methods getClass( ), notify( ), notifyAll( ), and wait( ) are declared as final.
- The equals( ) method compares the contents of two objects.
  - It returns true if the objects are equivalent, and false otherwise.
- The toString( ) method returns a string that contains a description of the object on which it is called.
  - This method is automatically called when an object is output using println( ).
  - Many classes override this method.

- Inheritance –
  - Super Class,
  - Sub Class,
  - The Keyword super,
  - protected Members,
  - Calling Order of Constructors,
  - Method Overriding,
  - the Object class,
  - Abstract Classes and Methods,
  - Using final with Inheritance.

Abstract Classes and Methods

- Sometimes we may want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving the implementation to be filled by each subclass.
- If we want some way to ensure that a subclass should override all necessary methods then we can make them abstract methods.
- For making a method an abstract method we have use abstract type modifier.

- Abstract methods are also called as subclasser responsibility , because they have no implementation in the superclass, but the implementation should be there in subclasses by overriding.
- To declare an abstract method syntax is :

```
abstract type name(parameter-list);
```

- Abstract function has no body in superclass.

.

ABSTRACT CLASS

- Any class that contains one or more abstract methods must also be declared abstract.
- Abstract class can have non abstract methods(concrete methods) also.
- To declare a class abstract, use the abstract keyword in front of the class keyword at the beginning of the class declaration.

```
abstract class classname
{

}
```

- Abstract classes cannot be instantiated using new operator.
  - i.e. Objects are not created from abstract class.
  - Such objects would be useless, because an abstract class is not fully defined.
- There are no abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or it should be declared abstract class.

```
// A Simple demonstration of abstract.
abstract class A {
  abstract void callme();
  void callmetoo() {
    System.out.println(“concrete method.");
    }
}
class B extends A {
  void callme() {
    System.out.println(“callme in B");
    }
}
```

```
class AbstractDemo {
  public static void main(String args[]) {
     B b = new B();
     b.callme();
     b.callmetoo();
  }
}
```

Output

```
callme in B
concrete method.
```

- Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.

```
// Using abstract methods and classes.
abstract class Figure
{   double dim1;
    double dim2;
    Figure(double a, double b)
    { dim1 = a;
      dim2 = b;   }
    abstract double area();
}
class Rectangle extends Figure
{  Rectangle(double a, double b)
   {  super(a, b);   }
   double area()
   {  System.out.println("Rectangle Area");
      return dim1 * dim2;   }
}
```

```
class Triangle extends Figure {
  Triangle(double a, double b)
  {  super(a, b);  }
  double area()
  {  System.out.println("Triangle Area");
     return dim1 * dim2 / 2;   }
}
Class AbstractAreas {
  public static void main(String args[]) {
  // Figure f = new Figure(10, 10); // illegal
  Rectangle r = new Rectangle(9, 5);
  Triangle t = new Triangle(10, 8);
  Figure figref; // no object is created
  figref = r;
  System.out.println("Area is " + figref.area());
  figref = t;
  System.out.println("Area is " + figref.area());
} }
```

OUTPUT

```
Rectangle Area
Area is 45.0
Triangle Area
Area is 40.0
```

```
OUTPUT
Rectangle Area
Area is 45.0
Triangle Area
Area is 40.0
```

- Here all subclasses of abstract class **Figure** must override area( ).

Using final with Inheritance

- Use of final keyword
  - final can be used to create the equivalent of a named constant. E.g. final int TOTAL=0;
  - final helps to prevent overriding in inheritance
  - final helps to prevent inheritance.

Using final to Prevent Overriding

- To disallow a method from being overridden, we can use final as a modifier at the start of its method declaration
- Methods declared as final cannot be overridden by subclass.

```
class A {
   final void show()
   {  System.out.println("This is a final method.");
   }
}
class B extends A {
   void show() // ERROR! Can't override.
   { System.out.println("Illegal!");
   }
}
```

- Here show() method is declared as final in A. So it cannot be overridden in subclass B. If we try to override, COMPILE ERROR will occur in ths program.

- Methods declared as final can sometimes provide a performance enhancement:
  - The compiler is free call them inline because it "knows" they will not be overridden by a subclass.
- When a small final method is called, Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call.
- Inlining is only an option with final methods.

- Normally, Java resolves calls to methods dynamically, at run time. This is called late binding.
- However, since **final methods cannot be overridden**, a call to one can be resolved at **compile time**. This is called **early binding.**

Using final to Prevent Inheritance

- To prevent a class from being inherited it can be declared as final
- Class with final modifier cannot be inherited. It cannot act as superclass.
- To make a class a final class, precede the class declaration with final.
- If we declare a class as final, it implicitly declares all of its methods as final.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself

```
final class A {
   // ...
   }
// The following class is illegal.
class B extends A { //ERROR!cannot create a subclass for final class A
// ...
}
```

It is illegal for B to inherit A since **A is declared as final.**

Packages and Interfaces –

- ➢Defining Package,
- ➢CLASSPATH,
- ➢Access Protection,
- ➢Importing Packages,
- ➢Interfaces.

Package

- Packages are containers for classes.
- A package in Java is used to group related classes and interfaces.
- They are used to keep the class name space compartmentalized.
  - For example, a package allows us to create a class named List, which we can store in our own package and it will not collide with some other class named List stored elsewhere.
- Packages are stored in a hierarchical manner.
- The package is both a naming and a visibility control mechanism.

- We can define classes inside a package
  - that are not accessible by code outside that package.

(default)

OR

  - that can be also accessed by subclasses outside the

package. (protected)

OR

  - That can be accessed by all classes in all packages(public)

- To create a package, simply include a package command as the first statement in a Java source file.
  - All classes declared in that file will belong to the specified package.
- The package statement defines a name space in which classes are stored.
- If we are not writing package statement, the class names are put into the default package, which has no name.

Defining Package

- General form for creating a package :

package packagename;

- Example: If we write the following statement at the beginning of our java program then it will create a package
- eg package named Oop.

package Oop;

- Java uses file system directories to store packages.
- Example: Any classes that we declare to be part of the package Oop must store their .class files in a directory called Oop.
- Any file can include the same package statement.
- The package statement simply specifies to which package the classes defined in a file belongs to.

- We can create a hierarchy of packages.
  - Separate each package name from other using period(dot) symbol.
- General form of a multileveled package statement is :

```
package pkg1.pkg2.pkg3;
```

- This specifies that package pkg3 is inside package pkg2 and pkg2 package is inside pkg1.

- E.g The package declared as

```
package java.awt.image;
```

  - needs to be stored in the path 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

- How does the Java run-time system know where to look for packages that we create?
  1. By default, the Java run-time system uses the current working directory as its starting point.
     - ➢ if our package is in a subdirectory of the current directory, it will be found.
  2. We can specify a directory path or set paths by setting the CLASSPATH environmental variable.
  3. We can use the -classpath option with java and javac to specify the path to your classes.

CLASSPATH

package MyPack;

- 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
- To execute the program
  - java MyPack.programname

- In the case of CLASSPATH and –classpath option , the class path must not include MyPack, itself. It must simply specify the path to MyPack.
- Suppose the path of MyPack directory is

C:\MyPrograms\Java\MyPack

  – Then the class path to MyPack is C:\MyPrograms\Java

Steps and examples for creating and using packages

- Create a folder pack1 inside E drive
- Create a file A.java

```
package pack1;
public class A
{  public static void main(String args[] )
   {  System.out.println("Hello");  }
   public void show()
   {  System.out.println("show in A");  }
}
```

Method 1

- Take path before pack1 folder in command prompt here it s E drive.
- Compile using
  - E:\>javac pack1/A.java
- To run

E:\>java pack1/A

- Or

E:\>java pack1.A

Method 2

- Set classpath in command prompt to path to folder before the package pack1

C:\Users\USER>**set CLASSPATH=;E:\**

To compile

C:\Users\USER> javac -cp . E:\pack1\A.java

To run

C:\Users\USER> java pack1.A

Hello

Method 3

- Using –classpath option
- Compile using

```
C:\Users\USER> javac E:\pack1\A.java
```

Or

```
C:\Users\USER> javac -classpath . E:\pack1\A.java
```

- Run using

```
C:\Users\USER> java -classpath E:\ pack1.A
```

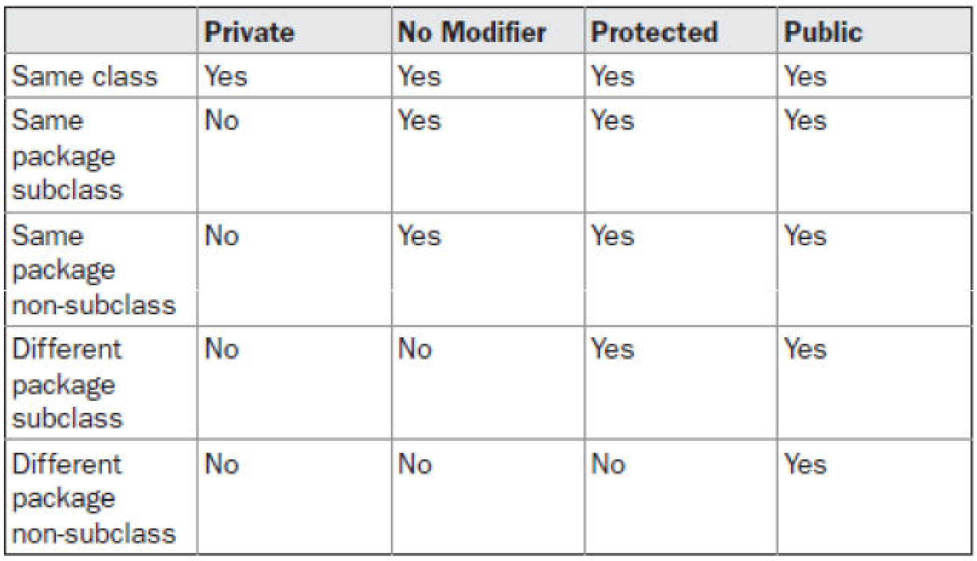
E.g using import statement

- Create a folder pack2 inside E drive
- Create a file B.java in it

```
package pack2;
import pack1.*;
class B{
  public static void main(String args[])
  {  A obj = new A();
     obj.show();
     System.out.println(“main in class B");  }
}
```

Access Protection

- Addresses four categories of visibility for class members:
  - – Subclasses in the same package
  - – Non-subclasses in the same package
  - – Subclasses in different packages
  - – Classes that are neither in the same package nor subclasses



	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

- A non-nested class has only two possible access levels:
  - default
  - public.
- When a class is declared as public, it is accessible by anyother code.

```
public class A {////
}
```

- If a class has default access, then it can only be accessed by other code within its same package.

```
class B
{ }
```

- When a class is public, it must be the only public class declared in the file, and the file must have the same name as the public class.

Importing Packages

- All of the standard classes are stored in some named package.
- If we want to use classes in some other packages, they must be fully qualified with their package name or names,.
- It is difficult to type in the long dot-separated package path name for every class we want to use.
  - TO SOLVE THIS PROBLEM, we can use import statement.
  - The import statement helps to bring certain classes, or entire packages, into visibility.
- To use a class or a package from the library, we need to use the import keyword
- import statements is written after the package statement(if exists) and before all class definitions.

- General form of the import statement:

import pkg1[.pkg2].(classname|*);

  - Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the package pkg1 separated by a dot
  - (.). Here square bracket denotes that it is optional.

- E.g.

import pack1; // import the package pack1

import java.io.*; // import all the classes from the package java.io

import java.util.Date; //import the Date class from the package java.util

- All of the standard Java classes included with Java are stored in a package called **java**
- The basic language functions are stored in a package inside of the java package called **java.lang**
  - it is implicitly imported by the compiler for all programs.

```
//Using an import statement:
import java.util.*;
class MyDate extends Date {
//statements , methods,variables
}
//Without the import statement looks like this:
class MyDate extends java.util.Date
{
}
```

```
//Program A.java
package pack1;
public class A
{  int a=100;
   public int c=20;
   protected int d=50;
   public void msg()
   {   System.out.println("Base class A
    Hello");
   }
}
```

```
//Program B.java
package pack2;
class B{
  public static void main(String args[])
  {  pack1.A obj = new pack1.A();
    obj.msg();
    System.out.println(“c="+obj.c);
    //System.out.println("d="+obj.d);
    // cannot access protected of
   //different package i.e. pack1
   //System.out.println("a="+obj.a);
   / /cannot access private of other class
  } }
```

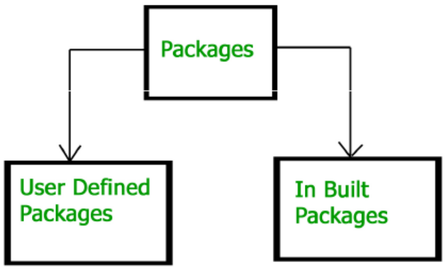
```
//Program A.java
package pack1;
public class A
{
  int a=100;
  public int c=20;
  protected int d=50;
  public void msg()
  {  System.out.println("Base class A
    Hello");  }
}
```

```
//Program B.java
package pack2;
import pack1.*;
class B{
  public static void main(String args[])
  {  A obj = new A();
     obj.msg();
     System.out.println("c="+obj.c);
     //System.out.println("d="+obj.d);
    // cannot access protected of different
     package pack1
    //System.out.println("a="+obj.a);/
    //cannot access private of other class
} }
```

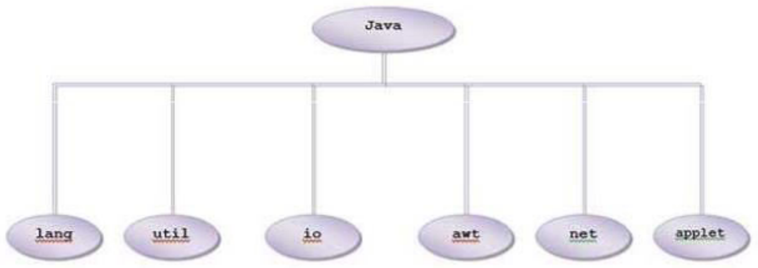
- Using import package.classname statement to import class A in pack1 to program file in pack2

```
//Program A.java
package pack1;
public class A
{
  int a=100;
  public int c=20;
  protected int d=50;
  public void msg()
  {  System.out.println("Base class A
    Hello");  }
}
```

```
//Program B.java
package pack2;
import pack1.A;
class B{
  public static void main(String args[])
  {  A obj = new A();
     obj.msg();
     System.out.println(“c="+obj.c);
     //System.out.println("d="+obj.d);
    // cannot access protected of different
     package pack1
    //System.out.println("a="+obj.a);/
    //cannot access private of other class
} }
```



Built-in Packages



Java Foundation Packages

- Java provides a large number of classes grouped into different packages based on their functionality.

- The six foundation Java packages are:
  - java.lang
    - Contains classes for primitive types, strings, math functions, threads, and exception
  - java.util
    - Contains classes such as vectors, hash tables, date etc.
  - java.io
    - Stream classes for I/O
  - java.awt
    - Classes for implementing GUI – windows, buttons, menus etc.
  - java.net
    - Classes for networking
  - java.applet
    - Classes for creating and implementing applets

Packages and Interfaces –

- ➢Defining Package,
- ➢CLASSPATH,
- ➢Access Protection,
- ➢Importing Packages,
- ➢Interfaces.

Interfaces

- Interface can be created using the keyword interface.
- Interfaces are syntactically similar to classes.
- Interface does not have instance variables.
- The methods in interface are declared without any body.
  - Interface never implements methods.
- Any number of classes can implement an interface.
- One class can implement any number of interfaces.
  - This helps to achieve multiple inheritance.

- To implement an interface,
  - a class must create and define the complete set of methods that are declared by the interface.
- Each class can have its own implementation of the methods.
- By providing the interface keyword, Java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism.
- Interfaces support dynamic method resolution at run time.

- General form of an interface:

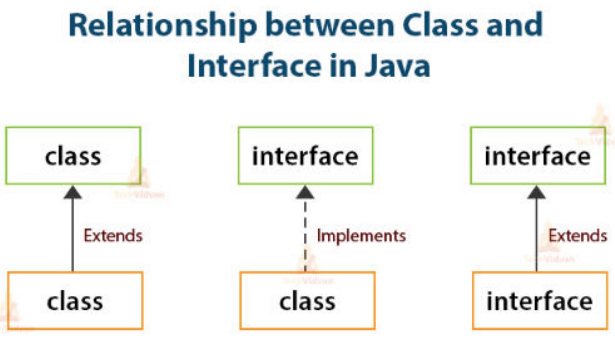
```
accessspecifier interface name {
  return-type method-name1(parameter-list);
  return-type method-name2(parameter-list);
  type final-varname1 = value;
  type final-varname2 = value;
  // ...
  return-type method-nameN(parameter-list);
  type final-varnameN = value;
}
```

- When no access specifier is included, then it has default access.
  - the interface is only available to other members of the package in which it is declared.
- The methods are declared have no bodies. They end with a semicolon after the parameter list.
- They are abstract methods.
- Each class that includes an interface must implement all of the methods.
- Variables re implicitly final and static, meaning they cannot be changed by the implementing class.
  - They must also be initialized.
- All methods and variables are implicitly public

```
interface Callback {
  void show(int param);
}
```

The relationship between classes and interfaces

Class	Interface
A class describes the attributes and behaviors of an object.	An interface contains behaviors that a class implements.
A class may contain abstract methods, concrete methods.	An interface contains only abstract methods.
Members of a class can be public, private, protected or default.	All the members of the interface are public by default.



```
/* File name : Animal.java */
interface Animal {
   public void eat();
   public void travel();
}
```

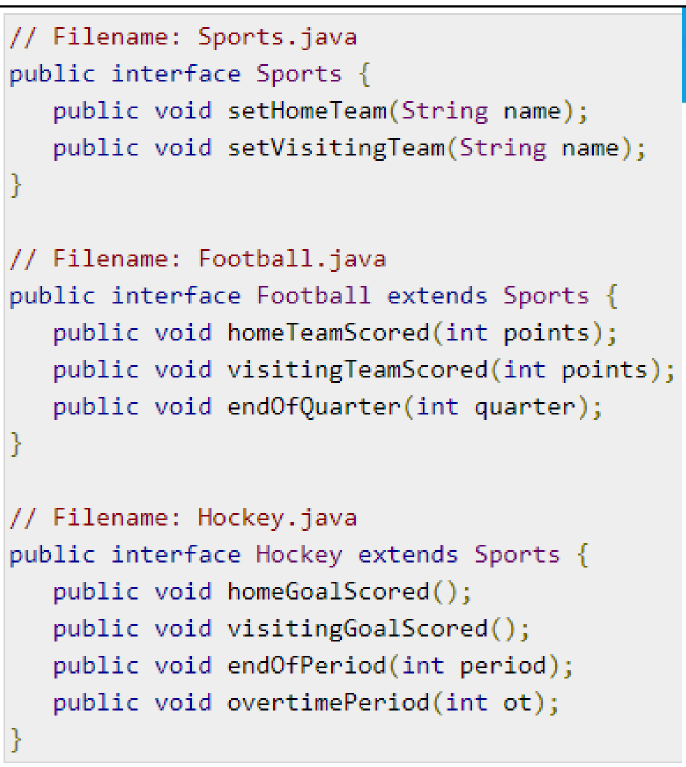
Output

```
Mammal eats
Mammal travels
```

```
/* File name : MammalInt.java */
public class MammalInt implements Animal {

   public void eat() {
      System.out.println("Mammal eats");
   }

   public void travel() {
      System.out.println("Mammal travels");
   }

   public int noOfLegs() {
      return 0;
   }

   public static void main(String args[]) {
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

Extending Interfaces

- The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.



```java
// Filename: Sports.java
public interface Sports {
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

// Filename: Football.java
public interface Football extends Sports {
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

// Filename: Hockey.java
public interface Hockey extends Sports {
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

- The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods.
- Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Variables in Interfaces

- When we include an interface in a class (using “implement” the interface), all of those **variable names in** the interface will be in scope as **constants.**
  - That is they are imported to class name space as **final** variables.

```
import java.util.Random;
interface Interf {
int NO = 0;
int YES = 1;
}
class Question implements Interf
{
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 50)
return NO; // 30%
else
return YES;
}
}
```

```
class AskMe implements Interf
{
static void answer(int result) {
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break; } }
public static void main(String args[])
    {
Question q = new Question();
answer(q.ask()); }
}
```

Partial Implementations

- If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as abstract.

```
interface Callback {
   void show(int param);        }
abstract class Incomplete implements Callback {
   int a, b;
   void display()
   {       System.out.println("display");
   }}
```

- Here the class Incomplete does not implement show() in the
- interface Callback. So the class Incomplete is abstract class

Nested Interfaces

- An interface can be declared a member of a class or another interface. Such an interface is called a member interface or a nested interface.
- A nested interface can be declared as public, private, or protected.
- The top level interface must either be declared as public or use the default access level.
- If we want to use a nested interface outside of its enclosing scope, the nested interface must be qualified by the name of the class or interface of which it is a member.

```
class A {
        // this is a nested interface
         public interface NestedIF
         {
        boolean isNotNeg(int x);
         }
}
class B implements A.NestedIF {
 public boolean isNotNeg (int x)
{
 return x < 0 ? false: true;
 }
}
```

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

 A.NestedIF nif = new B();
 if(nif.isNotNeg(10))
 System.out.println("10 is not negative");
 }

 }
```