UNIT-2

- Inheritance –Definition
- Single Inheritance
- Benefits of inheritance
- Member access rules
- super classes
- Polymorphism
- Method overriding
- Using final with inheritance
- abstract classes and
- Base class object.

Definition

- Inheritance is the process of acquiring the properties by the **sub class** (or derived class or child class) from the **super class** (or base class or parent class).

- When a **child** class(newly defined abstraction) inherits(extends) its **parent** class (being inherited abstraction), all the properties and methods of parent class becomes the member of child class.

- In addition, child class can add new data fields(properties) and behaviors(methods), and

- It can override methods that are inherited from its parent class.

Inheritance Basics

The key word **extends** is used to define inheritance in Java.

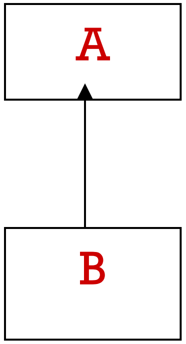
Syntax:-

```
class subclass-name extends superclass-name {
        // body of the class
}
```

Single Inheritance

-Derivation of a class from only one base class is called single inheritance.

```
//base class:
class A{
            //members of A
}
//Derived class syntax:
class  B extends A{
            //members of B
}
```



```
// Create a superclass.
class A {
      int i, j;
      void showij() {
      System.out.println("i and j: " + i + " " + j);
      }
}
// Create a subclass by extending class A.
class B extends A {
      int k;
      void showk() {
            System.out.println("k: " + k);
      }
      void sum() {
            System.out.println("i+j+k: " + (i+j+k));
      }
}
class SimpleInheritance {
      public static void main(String args[]) {
            A superOb = new A();
            B subOb = new B();
            // The superclass may be used by itself.
            superOb.i = 10;
            superOb.j = 20;
            System.out.println("Contents of superOb:");
            superOb.showij();
/* The subclass has access to all public members
of its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;

System.out.println("Contents of   subOb: ");

subOb.showij();
subOb.showk();
System.out.println();

System.out.println("Sum of i, j and k in subOb:");

subOb.sum();
}
}
```

```
Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

The Benefits of Inheritance

- **Software Reusability** (among projects)
  - Code (class/package) can be reused among the projects.
  - Ex., code to insert a new element into a table can be written once and reused.

- **Code Sharing** (within a project)
  - It occurs when two or more classes inherit from a single parent class.
  - This code needs to be written only once and will contribute only once to the size of the resulting program.

- **Increased Reliability** (resulting from reuse and sharing of code)
  - When the same components are used in two or more applications, the bugs can be discovered more quickly.

- **Information Hiding**
  - The programmer who reuses a software component needs only to understand the nature of the component and its interface.
  - It is not necessary for the programmer to have detailed information such as the techniques used to implement the component.

- **Rapid Prototyping** (quickly assemble from pre-existing components)
  - Software systems can be generated more quickly and easily by assembling preexisting components.
  - This type of development is called Rapid Prototyping.

- **Consistency of Interface**(among related objects)
  - When two or more classes inherit from same superclass, the behavior they inherit will be the same.
  - Thus , it is easier to guarantee that interfaces to similar objects are similar.

- **Software Components**
  - Inheritance enables programmers to construct reusable components.

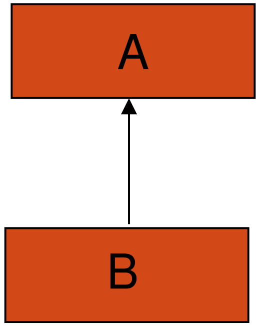
- **Polymorphism and Frameworks** (high-level reusable components)
  - Normally, code reuse decreases as one moves up the levels of abstraction.

  - Lowest-level routines may be used in several different projects, but higher-level routines are tied to a particular application.

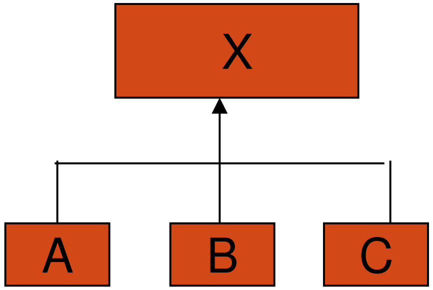
  - Polymorphism in programming languages permits the programmer to generate high-level reusable components that can be tailored to fit different applications by changes in their low-level parts.

Types of Inheritance

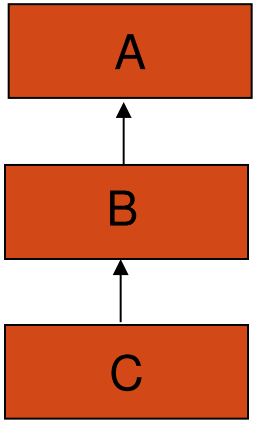
Single Inheritance



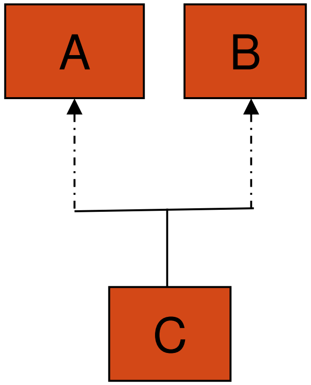
Hierarchical Inheritance



Multilevel Inheritance



Multiple Inheritance



```
//Single Inheritance

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

```
//Multilevel Inheritance

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

```
//Hierarchical Inheritance

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

```
//Multiple Inheritance

interface one{
}
interface two{
}
class A implements one, two{
}
```

- Multiple Inheritance can be implemented by implementing multiple interfaces not by extending multiple classes.

Example :

```
class B extends A implements C , D{

}
```

OK

```
class C extends A extends B{

}
```

```
class C extends A ,B{

           }
```

WRONG

A Superclass Variable Can Reference a Subclass Object

- When a reference to a subclass object is assigned to a superclass variable, you will have access only to those parts of the object defined by the superclass.

  Ex:

```
class A{
        int i=10;
}
class B extends A{
        int j=30;
}
class Test{
  public static void main(String args[]){
        A a=new A();
        B b=new B();
        a=b;
        System.out.println(a.i);
        //System.out.println(a.j);
  }
}
```

Super Keyword

- Subclass refers to its immediate superclass by using **super** keyword.

- **super** has two general forms.
  - First it calls the superclass constructor.
  - Second is used to access a member of the superclass that has been hidden by a member of a subclass.

- **Using super to call superclass constructors**

  super (parameter-list);

  - parameter-list specifies any parameters needed by the constructor in the superclass.
  - **super( )** must always be the first statement executed inside a subclass constructor.

```
class Box {
        Box() {
           System.out.println("Box() in super class");
        }
        Box(int a){
          System.out.println("Box(int a) in super class");
        }
}
class BoxWeight extends Box {
        BoxWeight(){
        System.out.println("BoxWeight() in sub class");
        }
}
class DemoBoxWeight{
        public static void main(String args[]) {
            BoxWeight mybox1 = new BoxWeight();
        }
}
```

Output:

```
Box() in super class
BoxWeight() in sub class
```

//Using super to call superclass constructors

```
class Box {
Box() {
        System.out.println("Box() in super class");
        }
        Box(int a){
             System.out.println("Box(int a) in super class");
        }
}
class BoxWeight extends Box {
        BoxWeight(){
            super(10);
            System.out.println("BoxWeight() in sub class");
        }
}
class DemoBoxWeight{
        public static void main(String args[]) {
                BoxWeight mybox1 = new BoxWeight();
        }
}
```

Output:

```
Box(int a) in super class
BoxWeight() in sub class
```

- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.

Syntax: super.member

- Here, member can be either a method or an instance variable.

- This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
// Using super to overcome name hiding.
class A {
    int i;
}
// Create a subclass by extending class A.
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();
    }
}
```

This program displays the following:

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

When Constructors Are Called

- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- super(…) must be the first statement executed in a subclass' constructor.
- If super(…) is not used, the default constructor of each superclass will be executed.
  - Implicitly default form of super (super()) will be invoked in each subclass to call default constructor of superclass.

```
class A {
            A() {
            System.out.println ("Inside A's constructor.");
            }
    }
class B extends A {
            B() {
            System.out.println("Inside B's constructor.");
            }
    }
class C extends B {
            C() {
            System.out.println("Inside C's constructor.");
    }
}
class CallingCons {
    public static void main(String args[]) {
    C c = new C();
    }
}
```

```
Output:
Inside A’s constructor
Inside B’s constructor
Inside C’s constructor
```

Member access rules

A subclass includes all of the members (default, public, protected) of its superclass except private members.

```
class A{
        private int v=10;
        int d=20;
        public int b=30;
        protected int p=40;
}
class B extends A{
        void disp(){
                //System.out.println(“v value : "+v);
                System.out.println(“d value : "+d);
                System.out.println(“b value : "+b);
                System.out.println("p value : "+p);
        }
}
class C extends B{
        void show(){
                System.out.println("p value : "+p);
        }
}
```

```
class Protected{
        public static void main(String args[]){
                B b=new B();
                b.disp();
                C c=new C();
                c.show();
        }
}
```

```
Output:
d value : 20
b value : 30
p value : 40
p value : 40
```

Polymorphism

- Assigning multiple meanings to the same method name
- Implemented using late binding or dynamic binding (run-time binding):
- It means, method to be executed is determined at execution time, not at compile time.
- Polymorphism can be implemented in two ways
  - Overloading
  - Overriding
- When a method in a subclass has the same name, signature and return type as a method in its superclass, then the method in the subclass is said to be overridden the method in the superclass.
- By method overriding, subclass can implement its own behavior.

//Overriding example

```
class A{
     int i,j;
     A(int a,int b){
          i=a;
          i=b;
     }
     void show(){
        System.out.println(“i and j :”+i+” “+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(3,4,5);
                              subob.show();
               }
}
```

Output:

K: 5

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an **overridden** method is resolved at run time, rather than compile time.

- When an **overridden** method is called through a superclass reference, the method to execute will be based upon the type of the object being referred to at the time the call occurs. Not the type of the reference variable.

//Dynamic Method Dispatch

```
class A{
    void callme(){
    System.out.println(“Inside A’s callme method”);
    }
}
class B extends A{
    void callme(){
    System.out.println(“Inside B’s callme method”);
    }
}
class C extends A{
    void callme(){
    System.out.println(“Inside C’s callme method”);
    }
}
```

```
 class Dispatch{
public static void main(String args[]){
                A a=new A();
                B b=new B();
                C c=new C();

                A r;

                 r=a;
                 r.callme();

                 r=b;
                 r.callme();

                 r=c;
                 r.callme();
 }
 }
```

Output:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

```
// Using run-time polymorphism.
class Figure {
    double dim1;
    double dim2;
    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }
    double area() {
  System.out.println("Area for Figure is undefined.");
      return 0;
    }
}
class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a,b);
    }
    // override area for rectangle
    double area() {
      System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
```

```
class Triangle extends Figure {
    Triangle(double a, double b) {
            super(a, b);
    }
    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
            return dim1 * dim2 / 2;
    }
}

class FindAreas {
    public static void main(String args[]) {
        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref;
        figref = r;
        System.out.println("Area is " + figref.area());
        figref = t;
        System.out.println("Area is " + figref.area());
        figref = f;
        System.out.println("Area is " + figref.area());
    }
}
```

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Abstract Classes

- A method that has been declared but not defined is an abstract method.

- Any class that contains one or more abstract methods must also be declared abstract.

- You must declare the abstract method with the keyword abstract:

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

- You must declare the class with the keyword abstract:

```
abstract class MyClass{
        ......
}
```

- An abstract class is incomplete, It has “missing” method bodies.

- You cannot instantiate (create a new instance of) an abstract class but you can create reference to an abstract class.

- Also, you cannot declare abstract constructors, or abstract static methods.

- You can declare a class to be abstract even if it does not contain any abstract methods. This prevents the class from being instantiated.

- An abstract class can also have concrete methods.

- You can extend (subclass) an abstract class.
  - If the subclass defines all the inherited abstract methods, it is “complete” and can be instantiated.
  - If the subclass does not define all the inherited abstract methods, it is also an abstract class.

```
// A Simple demonstration of abstract.
abstract class A {
    abstract void callme();
    // concrete methods are still allowed in abstract classes
    void callmetoo() {
            System.out.println("This is a concrete method.");
    }
}
class B extends A {
    void callme() {
    System.out.println("B's implementation of callme.");
    }
}
class AbstractDemo {
    public static void main(String args[]) {
            B b = new B();
            b.callme();
            b.callmetoo();
    }
}
```

Output:

```
B's implementation of callme.
This is a concrete method.
```

Using final with Inheritance

The keyword **final** has three uses:

To create a constant variable

To prevent overriding

To prevent inheritance

To create a constant variable:

– A variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared.

```
class FinalDemo{
        public static void main(String sree[]){
                final int i=20;
                System  em.out.println(i);
                //i=i+1;   can't assign a value to final variable i
        //System.out.println(i);   cannot assign a value to final variable i
        }
}
```

To prevent overriding

To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

```
class A {
        final void meth() {
                System.out.println("This is a final method.");
        }
}

class B extends A {
        void meth() {                   // ERROR! Can't override.
                System.out.println("Illegal!");
        }
}
```

To prevent inheritance

→To prevent a class from being inherited precede the class declaration with **final**.

→Declaring a class as **final** implicitly declares all of its methods as **final**, too.

→It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

```
final class A {
        // ...
}
// The following class is illegal.
class B extends A {          // ERROR! Can't subclass A
        // ...
}
```

- 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.

The Object Class

- **Object** is a special class, defined by Java.

- **Object** is a superclass of all other classes.

- This means that a reference variable of type **Object** can refer to an object of any other class.

- **Object** defines the following methods:

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( )	Waits on another thread of execution.
void wait(long *milliseconds*)	
void wait(long *milliseconds*, int *nanoseconds*)	

```
import java.io.*;
import java.util.Scanner;
class CharDemo{
    static char c[]=new char[10];
    public static void main(String sree[])throws Exception{
     //BufferedReader d=new BufferedReader(new InputStreamReader(System.in));
     System.out.println("Enter Characters:");
     for(int i=0;i<10;i++){
           //c[i]=(char)d.read();
           c[i]=(char)System.in.read();
     }
     System.out.println("Entered Characters:");
     for(int i=0;i<10;i++){
           System.out.println(c[i]);
     }
    }
}
```

- Defining an interface
- Implementing an interface
- Differences between classes and interfaces
- Implements and extends keywords
- An application using an interfaces and uses of interfaces

- Defining Package
- Creating and Accessing a Package
- Types of packages
- Understanding CLASSPATH
- importing packages

Interface

- It defines a standard and public way of specifying the behavior of classes.

- It defines a contract of a class.

- Using interface, you can specify what a class must do, but not how it does it.

- All methods of an interface are abstract methods. That is it defines the signatures of a set of methods, without the body.

- A concrete class must implement the interface (all the abstract methods of the Interface).

- Interface allows classes, regardless of their locations in the class hierarchy, to implement common behaviors.

- Once an interface is defined, any number of classes can implement an interface.

- Also, one class can implement any number of interfaces.

- Using the keyword interface, you can fully abstract a class' interface from its implementation.

- Using the keyword implements, you can implement any number of interfaces.

- The methods in interface are abstract by default.

- The variables in interface are final by default.

Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface interfacename {
                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;
      }
```

Example:

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

Here, access is either **public or not used.**

When **no access specifier** is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code.

‘name’ is the name of the interface, and can be any valid identifier.

Notice that the methods which are declared have no bodies. They are, essentially, abstract methods.

Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class.
They **must** also be initialized with a **constant value**.

All methods and variables are implicitly **public if** the interface, itself, is declared as **public.**

Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface.

To **implement** an interface, include the **implements clause** in a class definition, and then create the methods defined by the interface.

The general form of **a class that includes the implements clause** looks like this:

```
access class classname [extends superclass]   [implements interface [,interface...]] {
                    // class-body
}
```

- Here, access is either public or not used.

- If a class implements more than one Interface, the interfaces are separated with a comma.

- If a class implements two interfaces that declare the same method, then the same method will be **used by clients of** either interface.

- The methods that implement an interface **must be declared** public.

- Also, the **type signature of the implementing method** must match **exactly the type signature specified in the interface definition.**

Here is a small example class that implements the **Callback interface.**

```
class Client implements Callback {
            // Implement Callback's interface
            public void callback(int p) {
                  System.out.println("callback called with " + p);
            }
}
```

Notice that **callback( ) is declared using the public access specifier.**

When you implement an interface method, it must be declared as **public.**

It is both permissible and common for classes that implement interfaces to define **additional members of their own.**

For example, the following version of **Client** implements **callback( )** and adds the method **nonIfaceMeth( ):**

```
//Example for a class which contain both interface and non interface methods
        class Client implements Callback {
                // Implement Callback's interface
                public void callback(int p) {
                        System.out.println("callback called with " + p);
                }

                void nonIfaceMeth() {
                        System.out.println(“Non Interface Method…."); 
                }
        }
```

Accessing Implementations Through Interface References

- You can declare variables as object references that use an interface rather than a class type.

- Any instance of any class that implements the declared interface can be referred to by such a variable.

- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.

- This is one of the key features of interfaces.

- The calling code can dispatch through an interface without having to know anything about the "callee."

The following example calls the **callback( )** via **an interface reference** variable:

```
class TestIface {
                    public static void main(String args[]) {
                              Callback c = new Client();
                               c.callback(42);
                              //Callback cb;
                              //Client c=new Client();
                              //cb=c;
                              //cb.callback(42);
                    }
          }
```

Output:

```
                              callback called with 42
```

- Notice that variable **c is declared to be of the interface type Callback, yet it was** assigned an instance of **Client.**

- **Although c can be used to access the callback( )** method, it cannot access any other members of the **Client class.**

- **An interface reference** variable only has knowledge of the methods declared by its **interface declaration.**

- Thus, **c could not be used** to access **nonIfaceMeth( )** since it is defined by **Client** but not **Callback.**

- While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference.

```
// Another implementation of Callback.
    class AnotherClient implements Callback {
        // Implement Callback's interface
        public void callback(int p) {
                System.out.println("Another version of callback");
                System.out.println("p squared is " + (p*p));
        }
    }
```

```
class TestIface2 {
        public static void main(String args[]) {
            Callback c = new Client();
            AnotherClient ob = new AnotherClient();
            c.callback(42);
            c = ob; // c now refers to AnotherClient object
            c.callback(42);
        }
}
```

Output:

```
callback called with 42
Another version of callback
p squared is 1764
```

Partial Implementations

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

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

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

- Here, the class **Incomplete does not implement callback( )** and must be declared as **abstract**.

- Any class that inherits **Incomplete must** implement **callback( ) or** be declared **abstract itself.**

Variables in Interfaces

You can define variables in an interface but implicitly they are final variables.

That is you can’t modify them.

FinalDemo.java

```
interface FinalDemo{
      int i=100;
      void show();
}
```

FinalTest.java

```
class FinalImpl implements FinalDemo{
      public void show(){
            System.out.println("FinalTest :Show()");
      }
}
class FinalTest{
      public static void main(String sree[]){
            FinalImpl fi=new FinalImpl();
            fi.show();
            //fi.i=200; can’t assign a value to variable i
            System.out.println("FinalDemo Varaible i :"+fi.i);

      }
}
```

Output:

```
            FinalTest :Show()
            FinalDemo Varaible i :100
```

Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends.**

The syntax is the same as for inheriting classes.

When a **class** implements an **interface** that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
// One interface can extend another.
interface A {
    void meth1();
    void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
    void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
    public void meth1() {
    System.out.println("Implement meth1().");
}
    public void meth2() {
     System.out.println("Implement meth2().");
    }
    public void meth3() {
     System.out.println("Implement meth3().");
    }
}
class IFExtend {
    public static void main(String arg[]) {
     MyClass ob = new MyClass();
     ob.meth1();
     ob.meth2();
     ob.meth3();
    }
}
```

Output:

```
    Implement meth1().
    Implement meth2().
    Implement meth3().
```

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

```
class Client implements Callback {
        // Implement Callback's interface
        public void callback(int p) {
                System.out.println("callback called with " + p);
        }
        void nonIfaceMeth() {
                System.out.println(“NonInterface
Method…."); 
        }
}
```

```
// Another implementation of Callback.
class AnotherClient implements Callback {
        // Implement Callback's interface
        public void callback(int p) {
                System.out.println("Another version of callback");
                System.out.println("p squared is " + (p*p));
        }
}
```

```
class TestIface2 {
        public static void main(String args[]) {
        Callback c = new Client();
        AnotherClient ob = new AnotherClient();
        c.callback(42);
        c = ob; // c now refers to AnotherClient object
        c.callback(42);
}       }
```

Output:

```
callback called with 42
Another version of callback
p squared is 1764
```

Class Vs Interface

- The methods of an Interface are all abstract methods. They cannot have bodies.
- An interface can only define constants.
- You cannot create an instance from an interface.
- An interface can only be **implemented** by classes or extended by other interfaces.
- Interfaces have no direct inherited relationship with any particular class, they are defined **independently.**
- Interfaces themselves have **inheritance** relationship among themselves.
- A class can implement **more than one** interface. By contrast, a class can only inherit a single superclass (abstract or otherwise).

Abstract Class Vs Interface

- An abstract class is written when there are some common features shared by all the objects.
- An interface is written when all the features are implement differently in different objects.

- When an abstract class is written, it is the duty of the programmer to provide sub classes to it.
- An interface is written when the programmer wants to leave the implementation to the third party vendors.

- An abstract class contains some abstract methods and also some concrete methods.
- An interface contains only abstract methods.

- An abstract class can contain instance variables also.
- An interface can not contain instance variables. It contains only constants.

- All the abstract methods of the abstract class should be implemented in its sub classes.
- All the (abstract) methods of the interface should be implemented in its implementation classes.

- Abstract class is declared by using the keyword abstract.
- Interface is declared using the keyword interface.

- An abstract class can only inherit a single super class (abstract or otherwise).
- A class can implement **more than one** interface.

- Interfaces have no direct inherited relationship with any particular class, they are defined **independently.** Interfaces themselves have **inheritance** relationship among themselves.

- An abstract methods of abstract class have abstract modifier.
- A method of interface is an abstract method by default.

Uses of Interface

- To reveal an object's programming interface (functionality of the object) without revealing its implementation.
  - This is the concept of encapsulation.
  - The implementation can change without affecting the caller of the interface.

- To have unrelated classes implement similar methods (behaviors).
  - One class is not a sub-class of another.

- To model multiple inheritance.
  - A class can implement multiple interfaces while it can extend only one class.

Packages

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.

The package is both a naming and a visibility control mechanism.

A package represents a directory that contains related group of classes and interfaces.

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

You can also define class members that are only exposed to other members of the same package.

Pre-defined packages

1. java.applet
2. java.awt
3. java.beans
4. java.io
5. java.lang
6. java.lang.ref
7. java.math
8. java.net
9. java.nio
10. java.sql
11. java.text
12. java.util
13. java.util.zip

14.javax.sql

15.javax.swing

Defining a Packages

To create a package is quite easy: simply include a **package command as the first** statement in a Java source file.

Any classes declared within that file will belong to the specified package.

The **package statement defines a name space in which classes are** stored.
If you omit the **package statement, the class names are put into the default** package, which has **no name**.

This is the general form of the **package statement:**

Syntax: **package pkg;**

Example: **package MyPackage;**

Java uses file system directories to store packages.

More than one file can include the same **package statement.**

You can create a hierarchy of packages.

To do so, simply separate each package name from the one above it by use of a period.

The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system.

For example, a package declared as package java.awt.image; needs to be stored in **java\awt\image on your** Windows.

Finding Packages and CLASSPATH

How does the Java run-time system know where to look for packages that you create?

The answer has two parts:

First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found.

Second, you can specify a directory path or paths by setting the **CLASSPATH environmental variable.**

For example, consider the following package specification.

```
package MyPack;
```

In order for a program to find **MyPack, one of two things must be true.**

Either the program is executed from a directory immediately above **MyPack, or CLASSPATH** must be set to include the path to **MyPack.**

```
// A simple package
package MyPack;
class Balance {
    String name;
    double bal;
    Balance(String n, double b){
      name = n;
      bal = b;
    }
    void show() {
      if(bal<0)
      System.out.print("--> ");
      System.out.println(name + ": $" + bal);
    }
}
```

```
//AccountBalance.java
class AccountBalance {
        public static void main(String args[]) {
                Balance current[] = new Balance[3];
                current[0] = new Balance("K. J. Fielding", 123.23);
                current[1] = new Balance("Will Tell", 157.02);
                current[2] = new Balance("Tom Jackson", -12.33);
                for(int i=0; i<3; i++)
                        current[i].show();
                }
}
```

//To compile

```
javac AccountBalance.java
```

//To run

```
java MyPack.AccountBalance
```

//java AccountBalance **invalid**

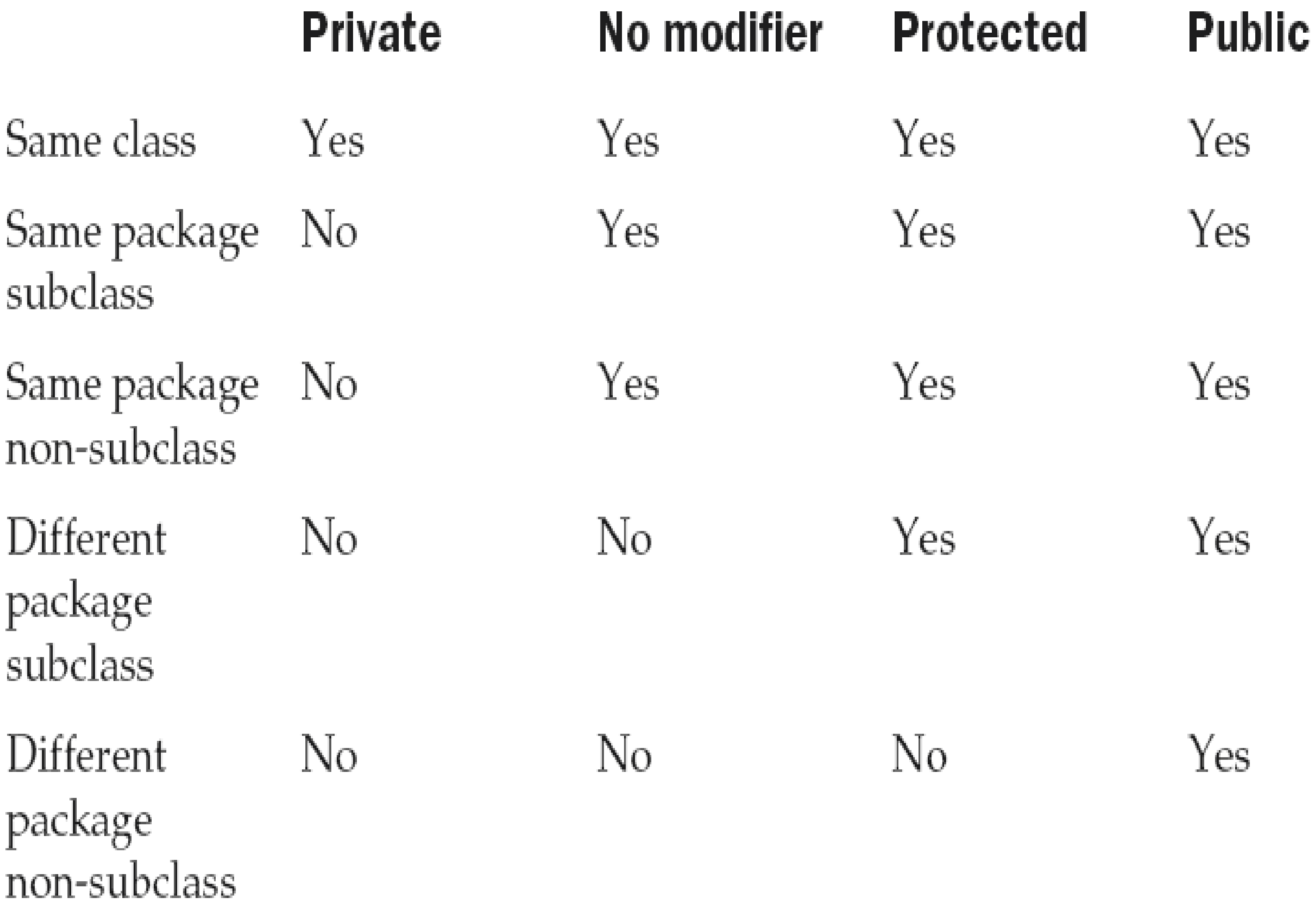
Access Control

Java 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.

A **class** has only two possible access levels: **default** and **public**.

Class Member Access



	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

```
//VarProtection.java
   package pack1;
   public class VarProtection {
    int n = 1;
    private int pri = 2;
    protected int pro = 3;
    public int pub = 4;
    public VarProtection() {
           System.out.println("Individual class constructor");
           System.out.println("default value is: " + n);
           System.out.println("private value is: " + pri);
           System.out.println("protected value is: " + pro);
           System.out.println("public value is: " + pub);
     }
}
```

To Compile:

```
d:\>javac –d . VarProtection.java
```

```
//SameSub .java:
package pack1;
class SameSub extends VarProtection{
    SameSub(){
     System.out.println("subclass constructor");
     System.out.println("default value is: " + n);
     // System.out.println("private value is: " + pri);
     System.out.println("protected value is: " + pro);
     System.out.println("public value is: " + pub);
    }
}
```

To Compile:

```
d:\>javac –d . SameSub.java
```

```
// SameDiff.java
package pack1;
class SameDiff{
    SameDiff(){
     VarProtection v1 = new VarProtection();
     System.out.println("Delegationclass constructor");
     System.out.println("default value is: " +v1. n);
     // System.out.println("private value is: " +v1. pri);
     System.out.println("protected value is: " +v1. pro);
     System.out.println("public value is: " + v1.pub);
    }
}
```

To Compile:

```
d:\>javac –d . SameDiff.java
```

```java
//OtherSub.java
package pack2;
import pack1.*;
class OtherSub extends VarProtection{
    OtherSub(){
     System.out.println("Different Package subclass constructor");
     //System.out.println("default value is: " + n);
     // System.out.println("private value is: " + pri);
     System.out.println("protected value is: " + pro);
     System.out.println("public value is: " + pub);
    }
}
```

To Compile:

```
d:\>javac –d . OtherSub.java
```

```
// OtherDiff.java
package pack2;
import pack1.*;
class OtherDiff{
     OtherDiff(){
       VarProtection v2=new VarProtection();
       System.out.println("Different Package non-subclass constructor");
       // System.out.println("default value is: " +v2. n);
       // System.out.println("private value is: " + v2.pri);
       // System.out.println("protected value is: " + v2.pro);
       System.out.println("public value is: " + v2.pub);
     }
}
```

To Compile:

```
d:\>javac –d . OtherDiff.java
```

```
// Demo package p1.
package pack1;
class MainTest{
    public static void main(String args[]){
     VarProtection v=new VarProtection();
     SameDiff s2=new SameDiff();
     SameSub s1=new SameSub();
    }
}
```

To Compile:

```
d:\>javac –d . MainTest.java
```

To Run:

```
d:\>java pack1.MainTest
```

```
package pack2;
import pack1.*;
class OtherMainTest{
    public static void main(String args[]){
     OtherSub os=new OtherSub();
     OtherDiff od=new OtherDiff();
    }
}
```

To Compile:

```
d:\>javac –d . OtherMainTest.java
```

To Run:

```
d:\>java pack2.OtherMainTest
```

Importing Packages

There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package.

Java includes the **import statement to bring certain classes, or entire** packages, into visibility.

Once imported, a class can be referred to directly, using only its name.

In a Java source file, **import statements occur immediately following the package** statement (if it exists) and before any class definitions.

This is the 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 outer package separated by a dot **(.).**

There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.

Finally, you specify either an explicit classname or a star **(*), which indicates that the Java compiler** should import the entire package.

This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

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.**

Normally, you have to import every package or class that you want to use, but **java.lang** is implicitly imported by the compiler for all programs.

This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code. For example, if you want the Balance class of the package MyPack shown earlier to be available as a stand-alone class for general use outside of MyPack, then you will need to declare it as public and put it into its own file, as shown here:

```
package MyPack;
public class Balance {
        String name;
        double bal;
        public Balance(String n, double b) {
                name = n;
                bal = b;
        }
        public void show() {
                if(bal<0)
                System.out.print("--> ");
                System.out.println(name + ": $" + bal);
        }
}
```

```
import MyPack.*;
class TestBalance {
    public static void main(String args[]) {
        /* Because Balance is public, you may use Balance
            class and call its constructor. */
        Balance test = new Balance("J. J. Jaspers", 99.88);
        test.show(); // you may also call show()
    }
}
```

As an experiment, remove the public specifier from the Balance class and then try compiling TestBalance. As explained, errors will result.