Unit-2

Inheritance

- It is one of the object oriented principles
- Definition: Deriving new class from existing class
- Existing class is called super class/parent class/base class
- Derived class is called sub class/child class/derived class
- Child class inherits all of the instance variables and methods of the super class and adds its own instance variables and methods

Inheritance Basics

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

Syntax:-

class *subclass-name extends superclass-name*

{

// body of class

}

Member Access rules

A subclass inherits all the members of its super class except ***private*** *members*

Example:

```
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));
    }
}
```

```
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("Contents of
      superOb:");
       superOb.showij();
       subOb.i = 7;
       subOb.j = 8;
       subOb.k = 9;
       System.out.println("Contents of
      subOb:");
       subOb.showij();
       subOb.showk();
       System.out.println("Sum of i, j and k in
      subOb:");
       subOb.sum();
    }
}
```

Output:

```
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
```

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

single inheritance

- subclass can be derived from one super class

A Superclass Variable Can Reference a Subclass Object

- The type of the ***reference variable*** determines what members can be accessed, not the type of the object

- That is, whenever 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 x1=10;
    void f1()  { System.out.println("Superlass A"); }
}
class B extends A
{
    int x2=20;
    void f2()  { System.out.println("Subclass B");  }
}
class RefDemo
{
    public static void main(String args[ ])
    {
      A a1;      // a1 is reference variable
      a1=new A();
      System.out.println(a1.x2);   //wrong
      a1.f2();   //wrong
    }
}
```

In Which Order Constructors Are Called

--In a class hierarchy constructors are called in the order of derivation, from super class to sub class

```
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

- **Default constructor**
  - A parameter less constructor is a default constructor
    - Programmer can provide explicitly
    - Automatically inserted by a compiler
  - Default constructor(created by a compiler) initializes all instance variables to default values
  - Compiler inserts a default constructor only if you don’t define any constructor (parameter / parameter less) explicitly

Ex1:- user default constructor

```
class A
{
    int i,j;
    A() {
          i=10; j=20;
          System.out.println("A default constructor");
          }
}
class B extends A
{
    int k;
    B() {
            k=20;
            System.out.println("B default constructor");   }
```

```
 void sum()      {
                    System.out.println("i+j+k:" + (i+j+k));
                 }
}
class SimpleInheritance
{
    public static void main(String args[])
    {
        B subOb = new B();
        subOb.sum();
     }
}
```

Output:

A default constructor

B default constructor

i+j+k:50

Ex2:- compiler default constructor

```
class A {
            A() {
                  System.out.println("Inside A's constructor.");
                }
    }
class B extends A {          //compiler inserts default 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();
        B b = new B();
     }
}
```

Ex3:- compiler default constructor

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

```
class SimpleInheritance1
{
    public static void main(String args[])
    {
        B subOb = new B();
        subOb.sum();
     }
}
```

Output:

i+j+k: 0

super uses

super has two uses

- To call super class constructor
- To access a member of the super class that is hidden by a member of a sub class

Using super to Call Super class Constructors

super(*parameter-list*);

- *parameter-list* specifies any parameters needed by the super class constructor

- **super( )** must be the first statement executed inside a subclass constructor

- Compiler automatically inserts default form of super (super ()) in each constructor

- Compiler does not insert default form of super if you define super explicitly

Ex1:- Parameter constructor with super

```
class A
{
    int i,j;
    A(int ii) {  i=ii; j=ii; System.out.println("A parameter constructor");  }
}
class B extends A
{
    int k;
    B(int kk) {
                    super(kk);
                    k=kk;
                    System.out.println("B parameter constructor");
                  }
void sum()      {
       System.out.println("i+j+k:" + (i+j+k));
    }
}
```

```
class SimpleInheritance2
{
    public static void main(String args[])
    {
        B subOb = new B(10);
        subOb.sum();
    }
}
```

Output:

A parameter constructor

B parameter constructor

i+j+k:30

Ex2:- constructors with out super

```
class A
{
    int i,j;
    A(){ i=10;j=20; System.out.println("A default constructor"); }
    A(int ii) { i=ii; j=ii; System.out.println("A parameter
    constructor");}
}
class B extends A
{
    int k;
    B() { k=20; System.out.println("B default constructor"); }
    B(int kk) { k=kk; System.out.println("B parameter
    constructor");}
```

```
void sum()     {
      System.out.println("i+j+k:" + (i+j+k));
   }
}
class SimpleInheritance3
{
   public static void main(String args[])
   {
        B subOb = new B(10);
        subOb.sum();
   }
}
```

Output:

```
A default constructor
B parameter constructor
i+j+k:40
```

A Second Use of super

- To access a member of the super class that is hidden by a member of a sub class

general form:

super.*member*

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

- This form is used to resolve name collisions that might occur between *super* and *subclass* member names

Ex1:-

```
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();
 }
}
```

Ex2:-

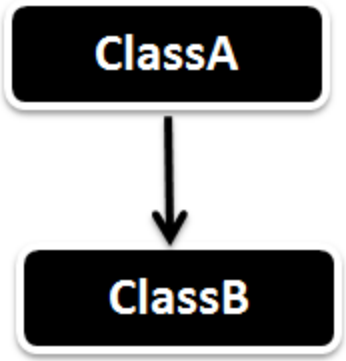
```
class A  {
 int i;
 void show()  {    System.out.println("i in superclass: " + i);     }
 }
class B extends A  {
 int i;                        // this i hides the i in A
B(int a, int b) {
 super.i = a;                 // i in A
 i = b;                       // i in B
}
void show()  {        System.out.println("i in superclass: " + super.i);
                       System.out.println("i in subclass: " + i);     }
void print() {          super.show();   }
}
class UseSuper {
 public static void main(String args[]) {
  B subOb = new B(1, 2);
  subOb.show();
  subOb.print();
 }
}
```

Types of inheritance

- Single Inheritance
- Multiple Inheritance (Through interface)
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance (Through Interface)

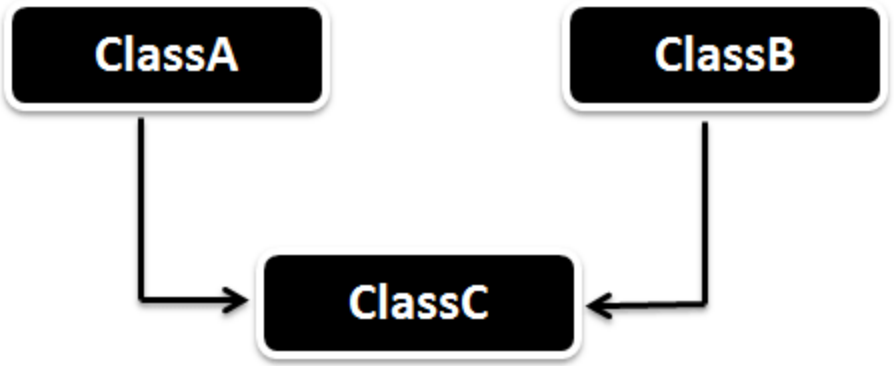
Single Inheritance

- Sub class is derived from a single parent class



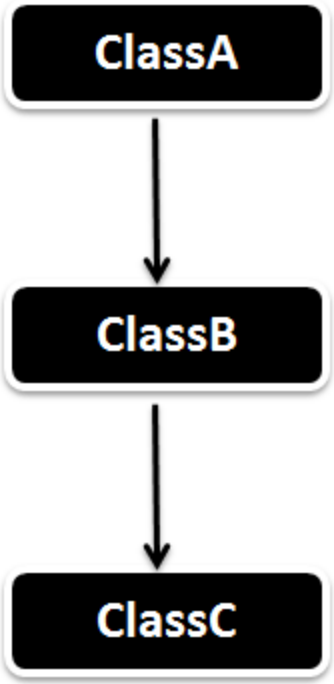
Multiple Inheritance

- Sub class is derived from two or more parent classes
- It is not supported by Java
- But, it can be achieved using interfaces



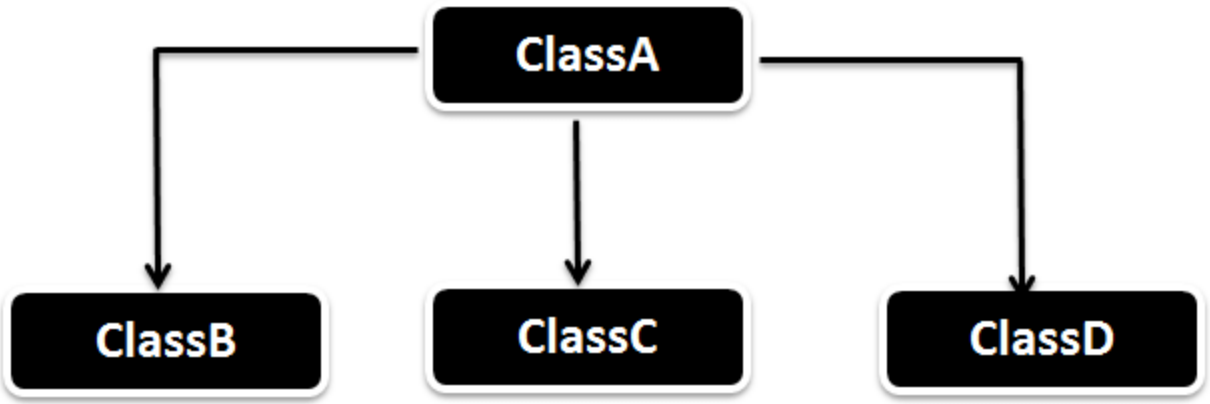
Multilevel Inheritance

• Sub class is derived from a parent class and this sub class is used to derive another sub class.



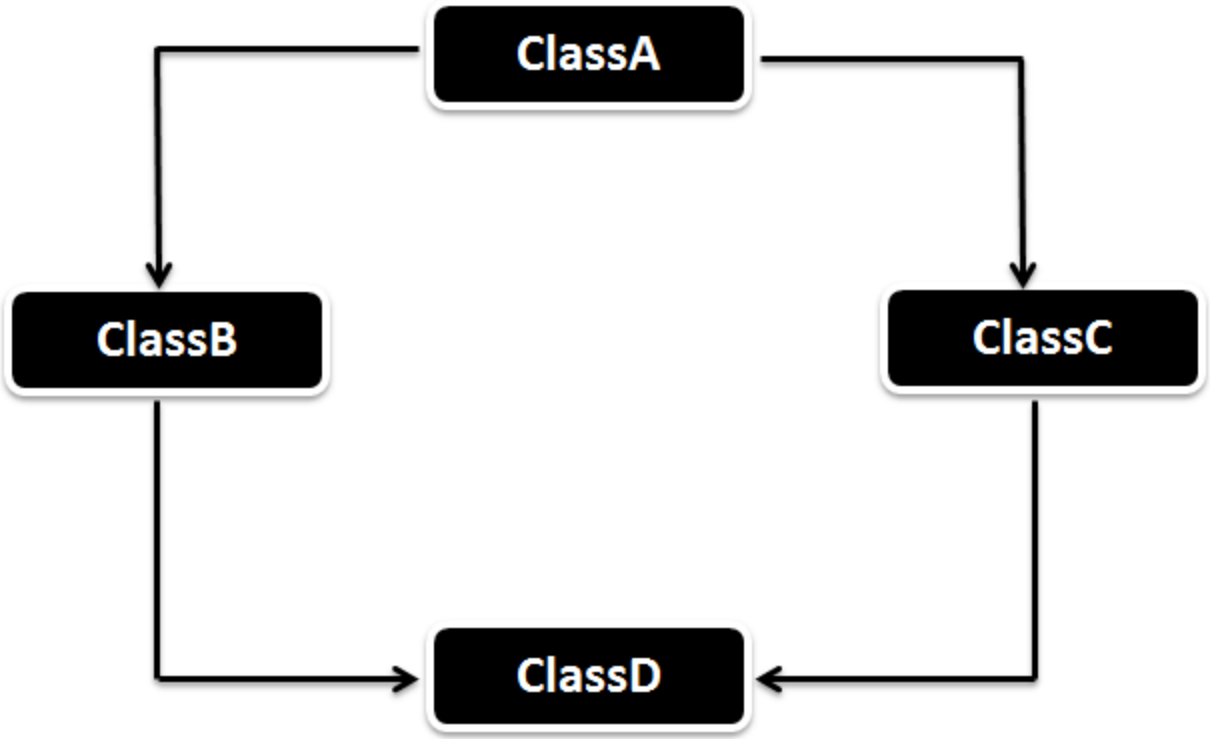
Hierarchical Inheritance

- Sub classes are derived from a single parent class



Hybrid Inheritance

- It is a combination of single inheritance and multiple inheritance
- It is not supported by Java
- But, it can be achieved using interfaces



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 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
- 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
  - The goal is to permit the development of new applications that require little or no actual coding
  - The java library offers a rich collection of software components for use in the development of applications

- 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

- Information Hiding
  - The programmer who reuses a software component needs only to understand the nature of the component and its interface
  - He does not have to know the techniques used to implement the component

Polymorphism

- Poly means “many” and morphism means “forms”

➢ There are two types

1. Method overloading
2. Method overriding

Method Overriding

- Method in a subclass has the same *signature* as a method in its super class
- In method overloading, method to be executed is determined at compile time. This process is called early binding or compile-time binding
- In method overriding, method to be executed is determined at execution time. This process is called late binding or dynamic binding (run-time binding)

Ex:-

```
class A {
   int i, j;
   A(int a, int b) { i=a; j=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(1, 2, 3);
     subOb.show(); }
}
```

Output: k: 3

Dynamic Method Dispatch

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

Ex:-

```
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 is reference variable
   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
```

Abstract Classes

- A ***method*** that has been declared but not defined is an abstract method
- Any class that contains at least one abstract method is an abstract class
- 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

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

Ex:-

```
abstract class Shape {
        abstract void draw();
}
class Rectangle extends Shape {
        void draw() {
                System.out.println("drawing rectangle");
        }
}
class Circle extends Shape {
        void draw() {
                System.out.println("drawing circle");
        }
}
class TestAbstraction {
        public static void main(String args[]) {
                Rectangle r = new Rectangle();
                r.draw();
                Circle c = new Circle();
                c.draw();
        }
}
```

Ex:-

```
abstract class Bank{
        abstract int getRateOfInterest();
}
class SBI extends Bank{
        int getRateOfInterest() { return 7;}
}
class PNB extends Bank{
        int getRateOfInterest() { return 8;}
}
class TestBank
{
    public static void main(String args[]){
    SBI s = new SBI();
    System.out.println("SBI Rate of Interest is: "+s.getRateOfInterest()+" %");
    PNB p = new PNB();
    System.out.println("PNB Rate of Interest is: "+p.getRateOfInterest()+" %");
        }
}
```

Ex:-

```
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.
```

Ex:-

```
abstract class A
{
        void callme() { System.out.println("This is A -callme method."); }
        void callmetoo() { System.out.println("This is A-callmetoo method."); }
}
class B extends A {
                void callme() {
                System.out.println("This is B-callme method.");
        }
}

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

Output:

```
This is B-callme method.
 This is A-callmetoo method.
```

- The keyword ***final*** has three uses
  - To create constant
  - To prevent overriding
  - To prevent inheritance

Using final to create constant

- A variable which is declared as **final** can not be modified latter
- This means that you must initialize a **final** variable when it is declared

Ex:-

```
final   float   PI = 3.14f;
```

Using final to Prevent Overriding

- 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()
    {
            System.out.println("final method  is overridden ");
     }
  }                                          // ERROR! Can't override
```

Using final to Prevent Inheritance

- Classes declared as **final** cannot be inherited
- 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, because** an abstract class is incomplete by itself and depends upon its subclasses to provide complete implementations

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

Base Class **Object**

- There is one special class, **Object**, defined by Java
- In Java, all classes use inheritance
- If no parent class is specified explicitly, the base class **Object** is implicitly inherited
- All classes defined in Java, is a child of **Object** class, which provides minimal functionality guaranteed to be common to all objects

➢**Object** defines the following methods, which means that they are available in every object

`boolean equals(Object object)`

--Determines whether one object equals to another object

`Class getClass( )`

--Obtains the class name of an object at run time

`int hashCode( )`

-- Returns the hash code associated with the invoking object

`String toString( )`

--Returns a string that describes the object (name of the class@hexadecimal representation of the hashcode)

Ex:- base class object

```
class BaseClass
{
   int i;
   char c;
   double d;
   BaseClass() { i=1234; c='z'; d=3.14; }
   void display() { System.out.println(i+"-"+c+"-"+d); }
}
class ObjectDemo
{
  public static void main(String args[])
  {
     BaseClass b=new BaseClass();
     BaseClass b1=new BaseClass();
     b1=b;
     b.display();
     System.out.println(b.hashCode());
     System.out.println(b.getClass());
     System.out.println(b.toString());
     System.out.println(b);
     System.out.println(b.equals(b1));
  }
}
```

Output:

```
1234-z-3.14
1671711
class BaseClass
BaseClass@19821f
BaseClass@19821f
true
```

Interfaces

- An ***interface***, is a way of describing *what* classes should do, without specifying *how* they should do it.
- Interfaces are syntactically similar to classes, but their methods are declared without any body.
- Any number of classes can implement an **interface**.
- One class can implement any number of interfaces.

Defining an Interface

- The general form of an interface:

```
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;
}
```

- Variables can be declared inside of an interface, They are implicitly **static** and **final**.
  - They cannot be changed.
  - They can be directly accessed by using interface name or class name that implements interface.

- Variables must be initialized with a constant value.

- Methods can not be declared as ***static*** or ***final*.**

Ex:-

```
interface I
{
        void callback(int param);

}
```

Implementing Interfaces

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

- The general form of a class that **implements** an interface.

class *classname* [**extends** *superclass*]
[**implements** *interface* [,*interface*...]]
{

// class-body

}

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

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

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

Example class that implements the **Callback** interface

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

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

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

Example:

```
interface I {
  int x=100;
  void display();
}
class A implements I {
            public void display()  { System.out.println("class A method"); }
}
class B implements I {
            public void display() { System.out.println("class B method"); }
}
class MainClass {
             public static void main(String args[]) {
             A a = new A();
             B b = new B();
            System.out.println(a.x+"-"+b.x+"-"+I.x);
             a.display();
             b.display();
      }
}
```

- Classes that implement an interfaces can define additional members of their own

Ex:-

```
class Client implements I
{           // Implement Callback's interface
    public void callback(int p)
    {
        System.out.println("callback called with " + p);
    }
    void nonIfaceMeth()
    {
        System.out.println("Classes that implement interfaces " +
                                "may also define other members, too.");
    }
}
```

Partial Implementations

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

Ex:-

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

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

Variables in Interfaces

- You can use interfaces to share constants among classes by simply declaring an interface that contains variables which are initialized to the desired values.

Ex:-

```
interface SharedConstants
{
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}
```

Methods in Interfaces

- Methods are Just declared.
- Methods are not defined.
- Class which is implementing Interface has to define the methods

Interfaces Can Be Extended

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

- The syntax is similar to 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, otherwise that class should be declared as abstract class.

Ex:-An Application using interfaces

```
interface A
{
 void meth1();
 void meth2();
}
interface B extends A
{
 void meth3();
}
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();
 }
}
```

O/P:

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

Accessing Implementations Through Interface References

- Any instance of any class that implements the interface can be referred by an interface variable.
- Through an interface variable, only interface members can be accessed
- This is one of the ***key features*** of interfaces.

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

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

Note : **iface** can be used to access the **callback( )** method, but it cannot be used to access any other members of the **Client** class.

Example:

```
interface I {
  int x=100;
  void display();
}
class A implements I {
 public void display() { System.out.println(" A display method"); }
 void print() { System.out.println("A print method"); }
}
class MainClass {
            public static void main(String args[]) {
               I iface;
               A a = new A();
               iface=a;
               iface.display();
               System.out.println(iface.x);
               iface.print();
            }  }
```

Error

Use of implements and extends keyword

- **implements** is used to implement the interface (interface methods) by the class.

- **extends** is used to inherit one interface from another interface (inherit one class from another class).

Differences between classes and interfaces

- Interfaces has method declarations with out any body
- Interfaces has instance variables which are static and final
- Classes has to implement the interfaces
- Instances can not be created for interfaces

Use of Interfaces

- It acts as APIs (Application Programming Interface)
- It means users can implement interfaces in its own way (depends on the application)
- It is easy to add new features (like members) to the interfaces
- It is used in multiple inheritance

- Java does not support multiple inheritance through classes, but Java supports multiple inheritance through interfaces (one class can implement more than one interface)

Example:

```
interface X
{
   void methodX( );
}
interface Y
{
   void methodY( );
}
class MI implements X, Y  {
   public void methodX( )
   {
       System.out.println("Implementaion of methodX");
   }
```

```
 public void methodY( )
   {
      System.out.println("Implementaion of methodY");
   }
}
class MIDemo {
   public static void main(String args[ ]) {
    MI m =  new MI( );
    m.methodX( );
    m.methodY( );
 }
}
```