# Java Programming

## Unit – 2

## Classes, Inheritance, Polymorphism

Dr. Y. J. Nagendra Kumar
Professor of IT
Dean Technology and Innovation Cell - GRIET

# TABLE OF CONTENTS

# Classes and Objects

# Classes

A class is a *template* *for an object, and an object is an instance of a class.*

```
class classname
{
        type instance-variable1;
        type instance-variable2;
        // ...
        type instance-variableN;

type methodname1(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

# Classes contd.,

- The data, or variables, defined within a class are called ***instance variables***.
- *The code is* contained within ***methods***.
- *Collectively, the methods and variables defined within a class are* called ***members of the class***.
- Variables defined within a class are called instance variables because each instance of the Class contains its own copy of these variables.

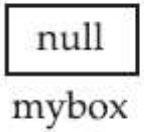
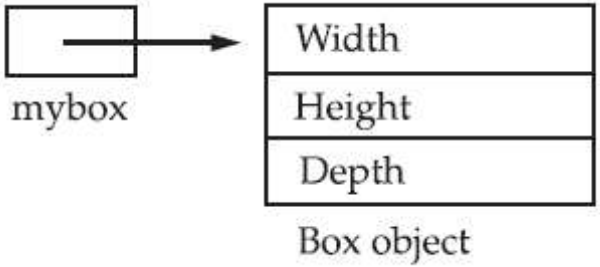
# Declaring Objects

- Objects of a class is a two-step process.
- First, we must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer to an object.*
- *Second, we must acquire an actual,* physical copy of the object and assign it to that variable. We can do this using the **new operator.**
- The new operator dynamically allocates (that is, allocates at run time) memory for an object

# Declaring Objects Contd.,

- Box mybox; // declare reference to object
- After this line executes, mybox contains the value null.
- mybox = new Box(); // allocate a Box object
- This line allocates an actual object and assigns a reference to it to mybox.
- We can combine the above statements into a singe one as follows:
- Box mybox = new Box();

# Declaring Objects Contd.,

| Statement | Effect |
| --- | --- |
| Box mybox; |   |
| mybox = new Box(); |   |

**A class is a logical construct. An object has physical reality.**

# Declaring Objects Contd.,

```
class Box
{
    double width;
    double height;
    double depth;
}
class BoxDemo
{   public static void main(String args[])
    {       Box mybox = new Box();
            double vol;
            mybox.width = 10;
            mybox.height = 20;
            mybox.depth = 15;

        vol = mybox.width * mybox.height *
        mybox.depth;

        System.out.println("Volume is " + vol);
        }
}
```

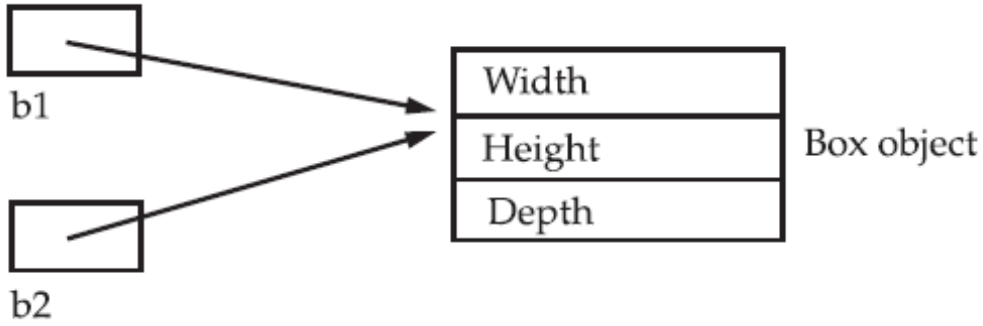
# Assigning Object Reference Variables

Box b1 = new Box();

Box b2 = b1;

- b1 and b2 will both refer to the *same object.*
- *The assignment of b1 to b2 did not allocate any memory.*
- It simply makes b2 refer to the same object as does b1.
- Thus, any changes made to the object through b2 will affect the object to which b1 is referring

# Assigning Object Reference Variables contd.,




```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1 has been set to null, but b2 still points to the original object.**

# Introducing Methods

- Classes usually consist of two things:

instance variables and

methods.

- This is the general form of a method:

*type name(parameter-list) {*

// body of method

}

- Here, *type specifies the type of data returned by the method.*

*This can be any valid type,* including class types

# Introducing Methods Contd.,

- If the method does not return a value, its return type must be void.
- The name of the method is specified by *name. This can be any legal identifier.*
- The *parameter-list is a* sequence of type and identifier pairs separated by commas.
- return *value;*
- Here, *value is the value returned.*

# Introducing Methods Example

```
class Box
{
    double width;
    double height;
    double depth;
    double volume()
    {
return width * height * depth;
    }
}
```

```
class BoxDemo1
{   public static void main(String args[])
    {       Box mybox1 = new Box();
            double vol;
            mybox1.width = 10;
            mybox1.height = 20;
            mybox1.depth = 15;
            vol = mybox1.volume();
 System.out.println("Volume is " + vol);
    }
}
```

# Adding a Method that takes parameters

```
class Box
{    double width;
     double height;
     double depth;
void set(double w, double h, double d)
{ width=w; height=h; depth=d; }
     double volume()
{ return width * height * depth;
     }
}
```

```
class BoxDemo2
{    public static void main(String args[])
     {       Box mybox1 = new Box();
             double vol;
             mybox1.set(20,30,40);
             vol = mybox1.volume();
 System.out.println("Volume is " + vol);
     }
}
```

# Constructors

- It can be tedious to initialize all of the variables in a class each time an instance is created.
- Even when we add convenience functions like **set( )**
- Java allows objects to initialize themselves when they are created.
- This automatic initialization is performed through the use of a constructor.
- A *constructor initializes an object immediately upon creation.*
- *It has the same name as the* class

# Constructors contd.,

- The constructor is automatically called immediately after the object is created.
- Constructors look a little strange because they have no return type, not even **void.**
- This is because the implicit return type of a class' constructor is the class type itself.
- We can construct **Box** objects of various dimensions.
- The easy solution is to add parameters to the constructor. It is called "Parameterized Constructors".

# Parameterized Constructors Example

```
class Box
{    double width;
     double height;
     double depth;
Box(double w,double h,double d)
{ width=w; height=h; depth=d; }
double volume()
{ return width * height * depth;
}
}
```

```
class Constructor
{    public static void main(String args[])
     {
      Box mybox1 = new Box(20,30,40);
            double vol;
            vol = mybox1.volume();
 System.out.println("Volume is " + vol);
     }
}
```

# The this Keyword

- Sometimes a method will need to refer to the object that invoked it.
- To allow this, Java defines the this keyword.
- this can be used inside any method to refer to the *current object.*
- *That is,* this is always a reference to the object on which the method was invoked.

# The this Keyword

```
class Box
{    double width;
     double height;
     double depth;

     Box(double width, double height,
     double depth)
     {       this.width=width;
             this.height=height;
             this.depth=depth;
     }
     double volume()
     {       return width * height * depth;
     }
}
class Constructor
{    public static void main(String args[])
     {       Box mybox1 = new Box(20,30,40);
             double vol;
             vol = mybox1.volume();
             System.out.println("Volume is " + vol);
     }
}
```

# Garbage Collection

- Since objects are dynamically allocated by using the **new operator**
- When objects are destroyed and the memory released is used for later reallocation.
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete operator.**
- Java takes a different approach; it handles de allocation for us automatically.
- The technique that accomplishes this is called ***garbage collection.***

# The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed.
- For example, if an object is holding some non-Java resource such as a file handle or network connections, data base connections, then we might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called *finalization.*
- To add a finalizer to a class, you simply define the **finalize( ) method.**

# The finalize( ) Method Contd.,

- The Java run time calls that method whenever it is about to recycle an object of that class.
- Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed.
- The garbage collector runs periodically, checking for objects that are no longer referenced.

```
protected void finalize( )
{
        // finalization code here
}
```

# Method Overloading

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. The methods are said to be *overloaded,* and the process is referred to as *method overloading.*

# Method Overloading

- Method overloading is one of the ways that Java implements Polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

# Automatic Type Conversion

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- However, this match need not always be exact. In some cases Java's automatic type conversions can play a role in overload resolution.

# Method Overloading Example

```
class overloading
{    void show()
     {System.out.println("No arguments");
     }
     void show(int x,int y)
{ System.out.println(“2 arguments"+x+y);
     }
     void show(double x)
     {
System.out.println("one argument :"+x);
     }
}
```

```
class methodoverloading
{
     public static void main(String ar[])
     {
     overloading m=new overloading();
               m.show();
               m.show(33,44);
               m.show(3.14);
               m.show(22);
     }
}
```

# Overloading Constructors

- In addition to overloading normal methods, we can also overload constructor methods.

## Using Objects as Parameters

- So far we have only been using simple types as parameters to methods.
- However, it is both correct and common to pass objects to methods.



# Overloading Constructors Example

```
class rect
{    double l,b;
     rect()
     {  l=10;b=20;  }
     rect(int x,int y)
     {  l=x; b=y;  }
     rect(rect n)
     {  l=n.l;
        b=n.b;    }
     rect(double x)
     {  l=b=x;    }
     void area()
     { System.out.println("Area :"+l*b);       }
}
```

```
class rectarea
{    public static void main(String ar[])
     {
     rect r1=new rect();
     rect r2=new rect(22,33);
     rect r3=new rect(r2);
     rect r4=new rect(55.6);

     r1.area();
     r2.area();
     r3.area();
     r4.area();
     }
}
```

# Returning Objects Example

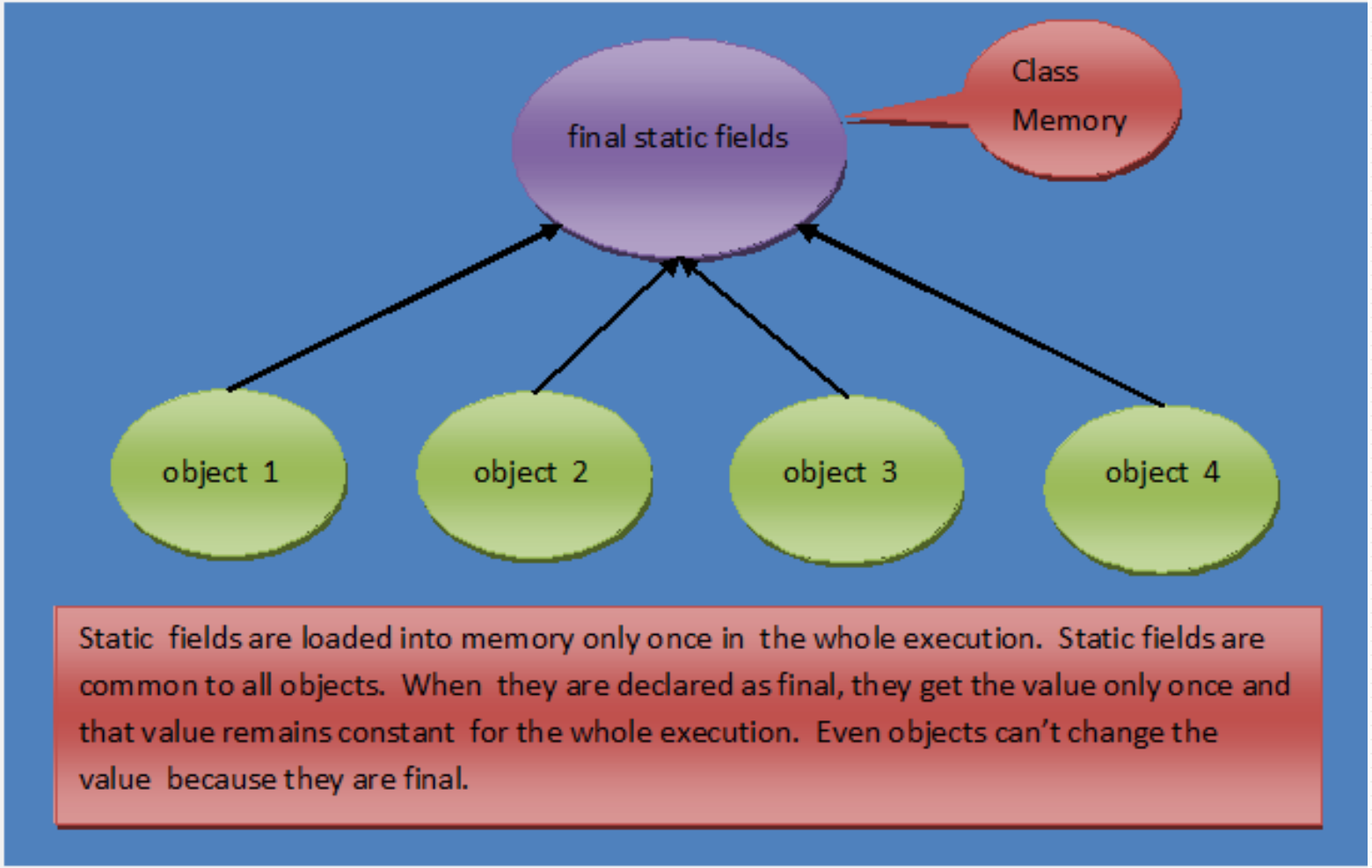
- A method can return any type of data, including class types that we create.

```
class Test
{
    int a;

    Test(int i)
    {
        a=i;
    }
    Test incr()
    {
        Test temp=new Test(a+10);
        return temp;
    }
}
```

```
class retobjects
{
    public static void main(String args[])
    {
        Test ob1 = new Test(5);
        Test ob2;

        ob2=ob1.incr();
    System.out.println("ob1.a : "+ob1.a);
    System.out.println("ob2.a : "+ob2.a);

    }
}
```

# Static

- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- We can declare both methods and variables to be static.
- The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.
- Instance variables declared as static are, essentially, global variables.


final static fields
Class Memory
object 1
object 2
object 3
object 4
Static fields are loaded into memory only once in the whole execution. Static fields are common to all objects. When they are declared as final, they get the value only once and that value remains constant for the whole execution. Even objects can't change the value because they are final.

# Static

- All instances of the class share the same static variable.
  - Ex: static int a = 3;
- Methods declared as static have several restrictions:
  - ■ They can only call other static methods.
  - ■ They must only access static data.
  - ■ They cannot refer to this or super in any way

  - *Syntax: classname.method*( )

Ex: static void callme()
{ System.out.println("a = " + a); }

# Static Example

```
class StaticDemo
{
    static int a = 42;
    static int b = 99;

    static void callme()
    {
    System.out.println("a = " + a);
    }
}
```

```
class statics
{
    public static void main(String args[])
    {
            StaticDemo.callme();

            System.out.println("b = " + StaticDemo.b);
    }
}
```

# final

- “final” keyword prevents its contents from being modified. This means that we must initialize a final variable when it is declared.
- It is a common coding convention to choose all uppercase identifiers for final variables.
- Variables declared as final do not occupy memory on a per-instance basis.
- Thus, a final variable is essentially a constant.

*Ex: final int x=22;*

# Access specifiers

- A member can be accessed is determined by the access specifier.
- Java's access specifiers are public, private, and protected. Java also defines a default access level.
- public: member can be accessed by any other code
- private: member can only be accessed by other members of its class
- Protected: applies only when inheritance is involved
- defalut: When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

# Access Specifiers Example

```
class Test
{
    int a;              // default access
    public int b;       // public access
    private int c;      // private access
    // methods to access c
    void setc(int i)
    {       c = I;      }
    int getc()
    {       return c; }
}
```

```
class Access
{   public static void main(String args[])
    {                   Test ob = new Test();
            ob.a = 10;
            ob.b = 20;
            // ob.c = 100; // Error!
            ob.setc(100); // OK
    System.out.println("a, b, and c: " + ob.a +
    " " +ob.b + " " + ob.getc());
    }
}
```

# Nested Classes

- Nested Class: define a class within another class
- The scope of a nested class is bounded by the scope of its enclosing class.
- Ex: if class B is defined within class A, then B is known to A, but not outside of A.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

# Static Nested Classes

- There are two types of nested classes:
  - static
  - non-static.
- A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object.
- That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

# Static Nested Classes Example

```
class Outer
{    int outer_x = 100;
     void test()
     {       Inner in=new Inner();
             in.display();
     }
     static class Inner
     {       void display()
             {   Outer out=new Outer();
     System.out.println("display: outer_x = " +
     out.outer_x);
             }
     }}
```

```
class Innerrr
{
     public static void main(String args[])
     {
             Outer outer = new Outer();
             outer.test();
     }
}
```

# Non Static Nested Classes (Inner Classes)

- An inner class is a non-static nested class.
- It has access to all of the variables and methods of its outer class and may refer to them directly.

# Inner Classes Example

```
class Outer
{     int outer_x = 100;
      void test()
      {         Inner in=new Inner();
                in.display();
      }
      class Inner
      {         void display()
                {
    System.out.println("display: outer_x = " + outer_x);
                }
      }
}
```

```
class InnerClass
{
      public static void main(String args[])
      {
                Outer outer = new Outer();
                outer.test();
      }
}
```

# Strings

# Strings

- String constants are actually String objects
- Strings are immutable; once a String object is created, its contents cannot be altered.
- If we need to change a string, we can always create a new one that contains the modifications.
- Java defines a peer class of String, called StringBuffer, which allows strings to be altered.

# String Methods

- Java defines one operator for String objects is +. It is used to concatenate two strings.
- For ex: String myString = "I" + " like " + "Java.";
- boolean equals(String object) → test two strings for equality
- int length( ) → obtain the length of a string
- char charAt(int index)→ obtain the character at a specified index within a string.

# String Methods Example

```
class StringMethods
{
    public static void main(String args[])
    {
        String s1 = "Pulse";
        String s2 = "AnnualDay";
        String s3 = s1;

System.out.println("Length of strOb1: " +s1.length());

System.out.println("Char at index 3 in strOb1: "
    +s1.charAt(3));

if(s1.equals(s2))
    System.out.println("s1 == s2");
else
    System.out.println("s1 != s2");

if(s1.equals(s3))
    System.out.println("s1 == s3");
else
    System.out.println("s1 != s3");
    }
}
```

# String Buffer

- String represents fixed length, immutable character sequence.
- String Buffer represents extensible character sequence
- In String Buffer object, we can insert or append substrings at any position.
- Array of Strings:
- String s[]={“IT”, “CSE”,”MCA”};

# String Buffer Example

```
class StringBuf
{
	public static void main(String a[])
	{
		StringBuffer sb=new StringBuffer("Happy Pongal");
		System.out.println("Length :"+sb.length());
		System.out.println("Capacity :"+sb.capacity());
		sb.insert(6,"Bogi, ");
		System.out.println(sb);
		sb.append(" and Kanuma");
		System.out.println(sb);
	}
}
```

# String Tokenizer

- The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

| Constructor | Description |
| --- | --- |
| StringTokenizer(String str) | creates StringTokenizer with specified string. |
| StringTokenizer(String str, String delim) | creates StringTokenizer with specified string and delimeter. |

# Methods of String Tokenizer Class

- **The 6 useful methods of StringTokenizer class are as follows:**

| Public method | Description |
|---|---|
| boolean hasMoreTokens() | checks if there is more tokens available. |
| String nextToken() | returns the next token from the StringTokenizer object. |
| String nextToken(String delim) | returns the next token based on the delimeter. |
| boolean hasMoreElements() | same as hasMoreTokens() method. |
| Object nextElement() | same as nextToken() but its return type is Object. |
| int countTokens() | returns the total number of tokens. |

# String Tokenizer Example

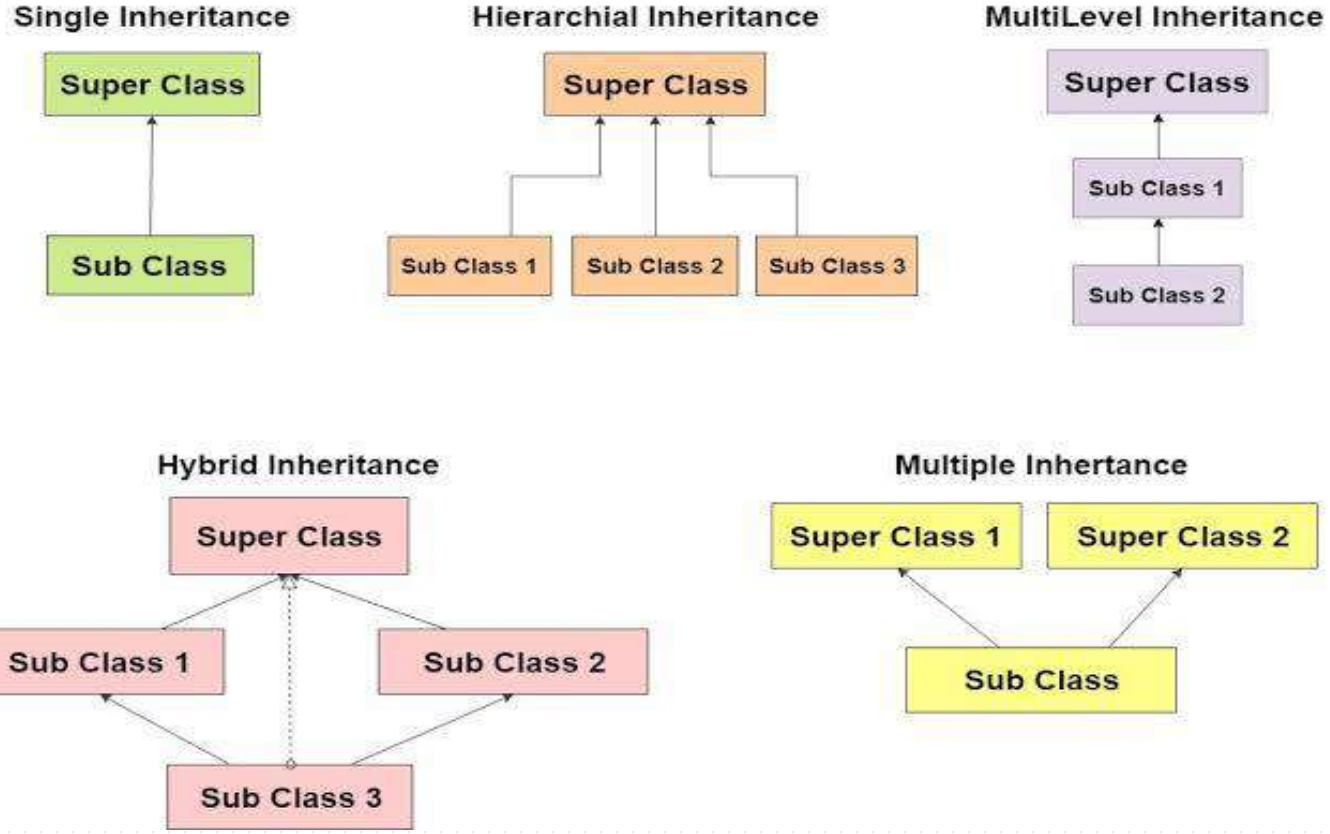
```
import java.util.StringTokenizer;
import java.util.Scanner;
 public class Tokenizer1
{    public static void main(String[] args)
     {        String s;
              Scanner sc=new Scanner(System.in);
              s=sc.nextLine();
              StringTokenizer st = new StringTokenizer(s);
              //iterate through tokens
              while(st.hasMoreTokens())
              System.out.println(st.nextToken());
     }
}
```

# Inheritance

# Inheritance

- The mechanism of deriving a new class from an old one is called Inheritance.
- The old class is known as “base” or “super” or “parent” class.
- The new one is called the “derived” or “sub” or “child” class.
- The forms of Inheritance
  - Single
  - Multiple
  - Multi level
  - Hierarchical
  - Hybrid

# Types of Inheritance



# Single Inheritance

- Subclass inherits all of the instance variables and methods defined by the super class and adds its own unique elements.
- To inherit a class we simply incorporate the definition of one class into another by using the “extends” keyword.

- Syn:

```
class subclassname extends superclassname
{
    //body
}
```

- Subclass cannot access those members of super class that have been declared as “private”

# Single Inheritance Example

```
class rect
{       double length;
        double breadth;
        rect()
        {  length=-1; breadth=-1;          }
        rect(double l,double b)
        {
                length=l; breadth=b;
        }
        double area()
        {
                return length*breadth;
        }
}
```

```
class box extends rect
{
    double height;
      box(double l,double b,double h)
      {         length=l;
                breadth=b;
                height=h;
      }
      void volume()
      {
                System.out.println("Volume :
      "+length*breadth*height);
      }
}
```

# Single Inheritance Example Contd.,

```
class inhert1
{
    public static void main(String ar[])
    {
        rect r=new rect(20,10);
        System.out.println("Area : "+r.area());
        box b=new box(20,10,5);
        b.volume();
    }
}
```

- In the previous example, the constructor for **box** explicitly initializes the length and breadth fields of rectangle. This duplicate code makes the program inefficient.

- Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword "**super**".

# ‘super’ forms

- super has two general forms.
  - The first calls the superclass’ constructor.
  - The second is used to access a member of the superclass

# Using super to Call Superclass Constructors

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

**super(*parameter-list*);**

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

# Super First form Example

```
class rect
{     double length;
      double breadth;
      rect()
      {  length=-1; breadth=-1;          }
      rect(double l,double b)
      {
                  length=l; breadth=b;
      }
      double area()
      {
                  return length*breadth;
      }
}
```

```
class box extends rect
{
     double height;
        box()
      {          super();
                 height=-1;      }
      box(double l,double b,double h)
      {          super(l,b);
                 height=h;       }
      void volume()
      {
System.out.println("Volume : “+length*breadth*height);
      }
}
```

# Super First form Example Contd.,

```
class inhert2
{
    public static void main(String ar[])
    {
        rect r=new rect(20,10);
        System.out.println("Area : "+r.area());

        box b=new box(30,40,5);
        b.volume();
    }
}
```

# A Second Use for super

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

  Syn: 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.

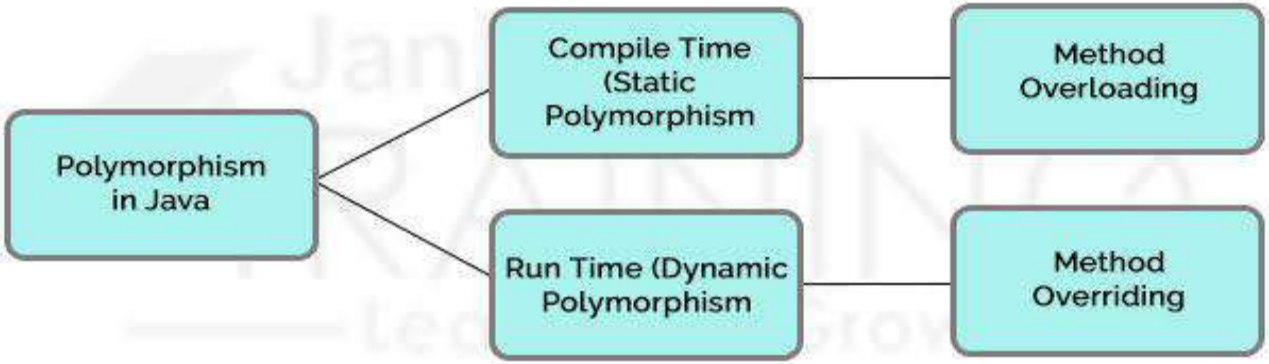
# Super Second form Example

```
class A
{
        int i;
}

class B extends A
{
        int i;

        B(int a,int b)
        {
                super.i=a;
                i=b;
        }
void display()
        {
                System.out.println(" Super i : "+super.i);
                System.out.println(" Sub i : "+i);
        }
}
class inhert3
{
        public static void main(String ar[])
        {
                B sub=new B(20,30);
                sub.display();
        }
}
```

# Polymorphism

# Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- The word "poly" means many and "morphs" means forms. So polymorphism means many forms.



# Method Overloading

- In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called ***"Method Overloading".***
- Method Overloading is used when objects are required to perform similar tasks but using different input parameters. This process is known as ***Polymorphism.***



# Method Overloading Example

```
class A
{     int i,j;
      A(int a, int b)
      {          i=a;j=b;       }
      void display()
      {
      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 display(String s)
      {          System.out.println(s+k);
      }
}
class overload
{
      public static void main(String ar[])
      {
                 B sub=new B(10,20,30);

                 sub.display(" this is :");
                 sub.display();
      }
}
```

# 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.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

# Method Overriding Example

```
class A
{     int i,j;
      A(int a, int b)
      {          i=a;j=b;       }
      void display()
      {
      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 display()
      {
                 System.out.println("k: " + k);
      }
}
class Override
{
      public static void main(String args[])
      {
                 B subOb = new B(1, 2, 3);
      subOb.display();   // this calls show() in B
      }
}
```

# 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.**
- Dynamic method dispatch is also known as **run-time polymorphism.**

# Dynamic Method Dispatch

- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.

# Dynamic Method Dispatch Example

```
class draw
{
        double d1,d2;

        draw(double a, double b)
        {
                d1=a;d2=b;
        }
        double area()
        {
System.out.println(" Area is undefined");
                return 0;
        }
}
```

```
class triangle extends draw
{
        triangle(double a,double b)
        {
                super(a,b);
        }

        double area()
        {
System.out.println(" Inside Area for Triangle : ");
        return d1*d2/2;  // base * height / 2
        }
}
```

# Dynamic Method Dispatch Example contd.,

```
class rectangle extends draw
{
        rectangle(double a,double b)
        {
                super(a,b);
        }

        double area()
        {
        System.out.println(" Inside Area for Rectangle : ");
        return d1*d2;  // length * breadth
        }
}
```

```
class dynamic
{       public static void main(String ar[])
        {       draw d=new draw(10,20);
                triangle t=new triangle(12,8);
                rectangle r=new rectangle(6,4);
                draw ref;
                ref=d;
        System.out.println("Area is :"+ref.area());
                ref=t;
        System.out.println("Area is :"+ref.area());
                ref=r;
        System.out.println("Area is :"+ref.area());
        }
}
```

# Using “final” 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.**

# Final method Example

```
class A
{
      final void display()
      {         System.out.println(" Inside A  ");
      }
}
class B extends A
{
     // Error cannot override becoz of final
      void display()
      {
              System.out.println(" Inside B  ");
      }
}
```

```
class finalkeyword
{     public static void main(String ar[])
      {         A a=new A();
                a.display();
                B b=new B();
                b.display();    }
}
```

**Output:**

```
D:\>javac finalkeyword.java
finalkeyword.java:12: display() in B cannot override
      display() in A; overridden method is final
     void display() // Error cannot override becoz of
      final
         ^  1 error
```

# Using final to Prevent Inheritance

- Sometimes we will want to **prevent a class from being inherited.** To do this, 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 Example

```
final class A
{
      void display()
      {   System.out.println(" Inside A  ");
      }
}
class B extends A
// Error cannot Inherit becoz of final
{
      void display()
      {
      System.out.println(" Inside B  ");
      }
}
```

```
class finalclass
{     public static void main(String ar[])
      {          A a=new A();
                 a.display();
                 B b=new B();
                 b.display();
      }
}
```

**Output:**

```
D:\>javac finalclass.java
finalclass.java:10: cannot inherit from final A
class B extends A // Error cannot Inherit becoz of final
                  ^   1 error
```

# Abstract Classes

# Abstract Classes

- A superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- A superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- This situation can occur is when a superclass is unable to create a meaningful implementation for a method.



# Abstract Methods

- Java's solution to this problem is the *abstract method.*
- To declare an abstract method, use this general form:

abstract *type methodname(parameter-list)*;

*** no method body is present.

# Abstract Classes

- Any class that contains one or more abstract methods must be declared abstract.
- To declare a class abstract, simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator.
- Such objects would be useless, because an abstract class is not fully defined.

# Abstract Classes

```
abstract class draw
{
      double d1,d2;
      draw(double a, double b)
      {
            d1=a;d2=b;
      }
      abstract double area();
}
```

```
class triangle extends draw
{
      triangle(double a,double b)
      {
            super(a,b);
      }
      double area()
      {
      System.out.println(" Inside Area for Triangle : ");
      return d1*d2/2;  // base * height / 2
      }
}
```

# Abstract Classes

```
class rectangle extends draw
{
        rectangle(double a,double b)
        {
                super(a,b);
        }

        double area()
        {
        System.out.println(" Inside Area for
        Rectangle : ");
        return d1*d2;  // length * breadth
        }
}
```

```
class abstractclasses
{       public static void main(String ar[])
        {
//      draw d=new draw(10,20);    is illegal
                triangle t=new triangle(12,8);
                rectangle r=new rectangle(6,4);
                draw ref;

                ref=t;
        System.out.println("Area is :"+ref.area());
                ref=r;
        System.out.println("Area is :"+ref.area());
        }
}
```

# 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.**

# End of Unit II