UNIT-3

Introduction toArrays

- An Array is a collection of elements that share the same type andname.
- The elements from the array can be accessed by theindex.
- All array indices start atzero.
- You can access a specific element in the array by specifying its index within square brackets.

Advantages

Code Optimization: It makes the code optimized, we can retrieve or sort the data easily.

Random access: We can get any data located at any index position.

Disadvantages

Size Limit: We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used injava.

Types of Array

There are two types of array.

1. Single Dimensional Array
2. Multidimensional Array

Single Dimensional Array

To create an array, we must first create the array variable of the desired type. The general form of the One Dimensional array is as follows:

```
type array-var[] = new type[size];
```

- Here *type* declares the base type of the array. This base type determine what type of elements that the array willhold.
- *array-var* is the array variable that is linked to thearray.
- *size* specifies the number of elements stored in thearray
- **new** to allocate memory for an array. The elements in the array allocated by **new will automatically be initialized tozero**.

Example: String months[] = new int[12];

Here

- type is *String*, the variable name is *months*. All the elements in the months are *Strings*. Since, the base type is*String*.
- This example allocates 12-element array of *Strings* and links them to **months**. The elements in the array allocated by **new will automatically be initialized tonull**.
-

months

Index	0	1	2	3	4	5	6	7	8	9	10	11

Element	null	null	null	null	null	null	null	null	null	null	null

For example, this statement assigns the value *"February"* to the second element of **months**.
months[1] = "February";

Index	0	1	2	3	4	5	6	7	8	9	10	11
Element	null	February	null	null	null	null	null	null	null	null	null	null

Example Program: Write a Java Program to read elements into array and display them?

```
ArrayTest.javaimport
java.io.*; import
java.util.*; class
ArrayTest

{
        public static void main(String args[])
        {
                Scanner s=new Scanner(System.in);
                //declaring and allocating memory to array and all the elements are set zero

                int a[] = new int[5];

                //read the elements into array System.out.println("Enter
                the elements into Array:"); for(int i=0;i<5;i++)
                {
                        a[i]=s.nextInt();
                }

                //displaying the elements System.out.println("The
                elements of Array:"); for(int i=0;i<n;i++)
                {
                        System.out.print(a[i]+",");
                }
        }
}
```

Output

```
Enter the elements into Array :12 45 56 78 30 The
elements of Array: 12,45,56,78,30
```

Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two dimensional array variable called**twoD**

```
int twoD[][] = new int[4][4];
```

This allocates a 4 by 4 array and assigns it to **twoD**. Internally this matrix is implemented as an *array* of *arrays* of **int**.

	Right Index Determines the Columns			
Left index determines theRows	[0,0]	[0,1]	[0,2]	[0,3]
	[1,0]	[1,1]	[1,2]	[1,3]
	[2,0]	[2,1]	[2,2]	[2,3]
	[3,0]	[3,1]	[3,2]	[3,3]

Example Program for Matrix Addition

```
import java.io.*;
import java.util.Scanner;
class Testarray2
{
        public static void main(String args[])
        {
                int m, n, i, j;
                Scanner s=new Scanner(System.in);
                System.out.println("Enter the no. of rows and columns of matrix"); m =
                s.nextInt());
                n = s.nextInt());
                int m1[][] = newint[m][n];
                int m2[][] = newint[m][n];

                int result[][] = new int[m][n];
                System.out.println("Enter the elements of first matrix:"); for (
                i = 0 ; i < m ; i++ )
                        for ( j = 0 ; j< n ; j++ )
                                m1[i][j] = s.nextInt(); System.out.println("Enter
                the elements of second matrix:"); for ( i = 0 ; i < m ; i++ )
                        for ( j = 0 ; j < n ; j++ )
                                m2[i][j] = s.nextInt();

                System.out.println("Sum of entered matrices:"); for (
                i = 0 ; i < m ; i++ )
                {
                        for ( j = 0 ; j < n ; j++ )
                        {
                               result[i][j] = m1[i][j] + m2[i][j];
                               System.out.print( result[i][j]+"\t");
                        }
                       System.out.println();
                }
        }
}
```

Output

```
Enter the elements of first matrix: 1 2 3 4
Enter the elements of second matrix: 1 2 3 4
Sum ofenteredmatrices:      2       4
                            6       8
```

2.8.1 Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

type[ ] *var-name;*

Here, the square brackets follow the type specifier, and not the name of the array variable. This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

```
int[] nums, nums2, nums3; // create three arrays
```

creates three array variables of type **int**. It is the same as writing int nums[], nums2[], nums3[]; // create three arrays.

Inheritance Basics

Inheritance is the process by which one class acquires the properties of another class. This is important because it supports the concept of hierarchical classification. The class that is inherited is called ***super class***. The class that is inheriting the properties is called ***subclass***. Therefore the subclass is the specialized version of the super class. The subclass inherits all the instance variable and methods using the **extends** keyword, and adds its own code. Super class is also known as Parent class, and Base class. The sub class is also known as Child class and Derived class.

Aggregation is the process of making an object combining number of other objects. The behavior of the bigger object is defined by the behavior of its component objects. For example, cars contain number of other components such as engine, clutches, breaks, starter etc.

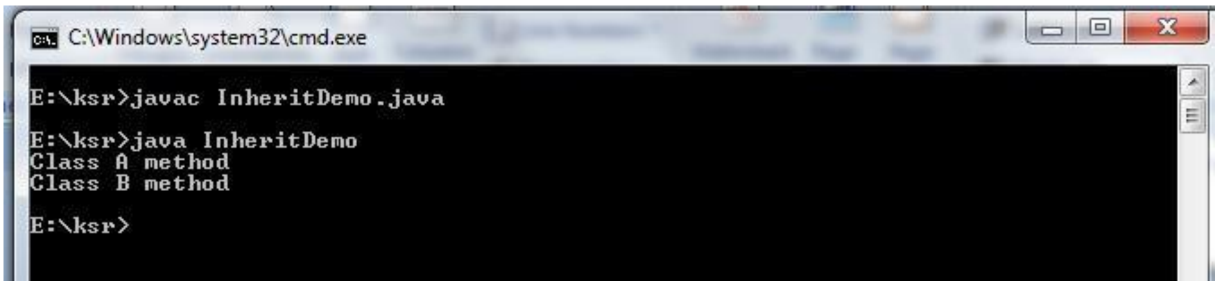
To inherit a class, we simply incorporates the definition one class into another class using the **extends** keyword.

Syntax:

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

(a) Write a JAVA program to implement Single Inheritance (Lab Exercise – 5 (a))

```
class A
{
        //body of the class A
        void methodA()
        {
                System.out.println("Class A method");
        }
}
class B extends A
{
        //body of the class B
        void methodB()
        {
                System.out.println("Class B method");
        }
}
class InheritDemo
{
        public static void main(String args[])
        {
        B b=new B();
        b.methodA();
        b.methodB();
        }
}
```



```
E:\ksr>javac InheritDemo.java

E:\ksr>java InheritDemo
Class A method
Class B method

E:\ksr>
```

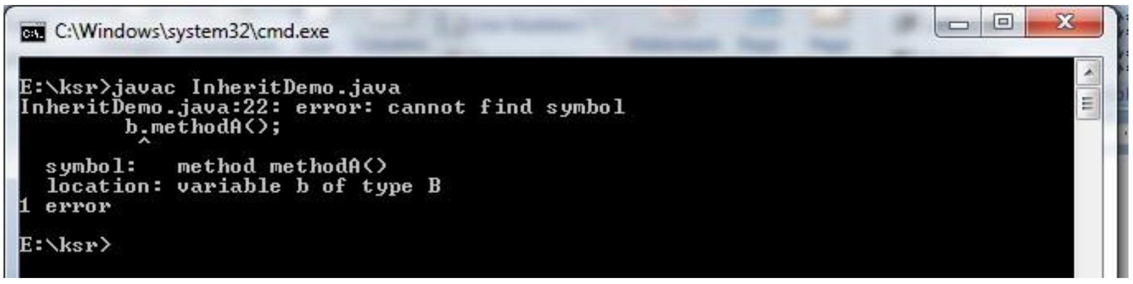
Private member access and Inheritance

Although sub class has the right to access members of the super class, but it cannot access private members of the super class.

Example Program:

```
class A
{
        //body of the class A
        private void methodA()      //private member of the super class

        {
                System.out.println("Class A method");
        }
}
class B extends A
{
        //body of the class B
        void methodB()
        {
                System.out.println("Class B method");
        }
}
class InheritDemo
{
        public static void main(String args[])
        {
        B b=new B();
        b.methodB();
        }
}
```

Output:



C:\Windows\system32\cmd.exe

```
E:\ksr>javac InheritDemo.java
InheritDemo.java:22: error: cannot find symbol
        b.methodA();
         ^
  symbol:   method methodA()
  location: variable b of type B
1 error

E:\ksr>
```

Another example:

// Create a superclass.

```
                                class A
                                   {
         int i; // public by default
       private int j; // private to A
          void setij(int x, int y)
                                       {
              i = x;
              j = y;
                                       }
                                   }
// A's j is not accessible here.
      class B extends A
 {       int total;
       void sum()
                                       {
                      total = i + j; // ERROR, j is not accessible here
                                       }
                                   }
                              class Access
                                   {
                     public static void main(String args[])
                                   {
            B b = new B(); // creating the object b
                  b.setij(10, 12);
                   b.sum(); // Error, private members are not accessed
                     System.out.println("Total is " + subOb.total);
                                   }
                                   }
```

Note: This program will not compile because the reference to **j** inside the **sum( )** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

Types of Inheritance

There are five different types of Inheritances:

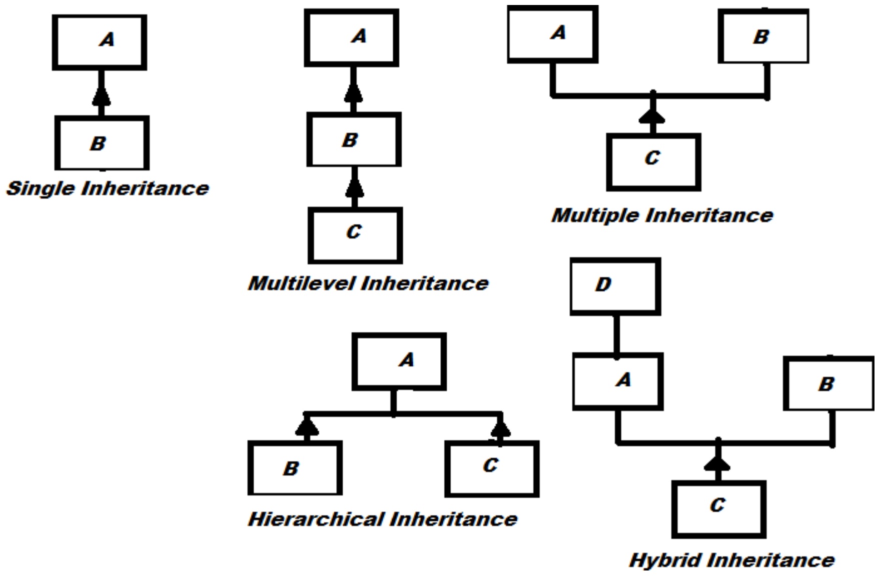
i. Single Inheritance

ii. Multiple Inheritance

iii. Multilevel Inheritance

iv. Hierarchical Inheritance

v. Hybrid Inheritance



Single Inheritance:

In this one class acquires the properties from another class and adds its own code to it. Example Program is shown below.

A more practical example to illustrate the Inheritance (Single Inheritance)

```
class Box
{
	double width,height,depth;
	Box(double w,double h,double d)
	{
		width=w;
		height=h;
		depth=d;
	}
}
class BoxVolume extends Box
{
	BoxVolume(double w,double h,double d)
	{
		super(w,h,d); //calling the super class constructor
	}
	void boxVolume()
	{
		double v=width*height*depth;
		System.out.println("The volume of the Box is "+v);
	}
}
class BoxTest
{
	public static void main(String args[])
```

```
        {
                BoxVolume bv=new BoxVolume(12.3,13.2,14.3);
                bv.boxVolume();
        }
}
```

Output:

```
E:\DHK>javac
BoxTest.java
E:\DHK>java  BoxTest
```

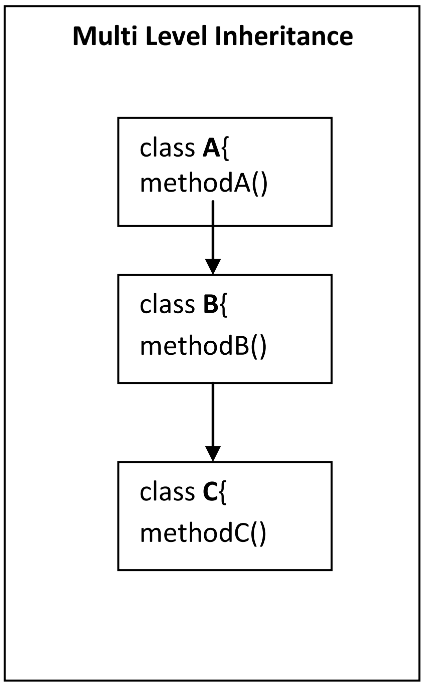
Multiple Inheritance:

In this one class acquires the properties from two or more classes at a time and adds its own code to it. This is not supported by Java among the classes, but is supported among the Interfaces. We will see this example in the Interface section.

Multilevel Inheritance:

In this one class acquires the properties from another class, which in turn has acquired the properties from another class. Hence, in this there are many levels in the process of Inheritance. The example program is below.

```
class A
        {
        public void methodA()
        {
                System.out.println("class A method");
        }
        }
class B extends A
        {
        public void methodB()
        {
                System.out.println("class B method");
        }
        }
class C extends B
        {
        public void methodC()
        {
                System.out.println("class C method");
        }
        }
class MLI
{
  public static void main(String args[])
  {
```



```
            C obj = new C();
            obj.methodA(); //calling grand parent class method
            obj.methodB(); //calling parent class method
            obj.methodC(); //calling child class method
  }
}
```

Output:

```
E:\DHK>javac
MLI.java
E:\DHK>java  MLI
class A method
class B method
```

Hierarchical Inheritance:

In this two or more classes will acquire the properties from only one same class and add their own code. The example program is given below.

```
class A
            {
            public void methodA()
            {
                        System.out.println("class A method");
            }
            }
class B extends A
            {
            public void methodB()
            {
                        System.out.println("class B method");
            }
            }
class C extends A
            {
            public void methodC()
            {
                        System.out.println("class C method");
            }
            }
class MLI
{
   public static void main(String args[])
   {
            B b=new B();
            C c = new C();
            System.out.println("calling the methodA() and methodB() with B's object");
            b.methodA(); // calling the methodA() and methodB() with B's object
```

```
        b.methodB();
        System.out.println("calling the methodA() and methodC() with C's object");
        c.methodA(); // calling the methodA() and methodC() with C's object
        c.methodC();
  }
}
```

Output:

```
E:\DHK>javac MLI.java
E:\DHK>java  MLI
calling the methodA() and methodB() with B's object
class A method
class B method
calling the methodA() and methodC() with C's object
class A method
class C method
```

Hybrid Inheritance:

In this different types of Inheritances are used to acquire the properties from number of classes to a single class. We will discuss this example in the interface section.

Note: Java Supports Single and Multilevel Inheritances between classes and Multiple Inheritance among the Interfaces.

A Superclass Reference Variable Can be assigned a Subclass Object

Any sub class reference variable can be assigned to the super class reference variable. When a reference to a subclass object is assigned to a super class reference variable, we will have access only to those parts of the object defined by the super class.
For example,

```
class Box
{
        double width,height,depth;
}
class Boxweight extends Box
{
        double weight;
        Boxweight(double x,double y,double z,double z)
        {
                width=x; height=y; depth=z; weight=a;
        }
        void volume()
        {
                System.out.println("The volume is :"+(width*height*depth));
        }
```

```
}
//main class

class BoxDemo
{
        //creating super class object
        public static void main(String args[])
        {
        Box b=new Box();
        //creating the subclass object
        Boxweight bw=new Boxweight(2,3,4,5);
        bw.volume();
        //assigning the subclass object to the superclass object

        b=bw;   // b has been created with its own data, in which weight is not a member

        System.out.println ("The weight is :"+b.weight);
```

Super keyword

There are three uses of the ***super*** keyword.

- ***i.*** super keyword is used to call the super class ***constructor***
- ***ii.*** super keyword is used to access the super class ***methods***
- ***iii.*** super keyword is used to access the super class ***instance variables***.

Using the super to call the super class constructor

A subclass can call the constructor of the super class by the using the super keyword in the following form:

```
super(arg_list);
```

Here, the arg_list, is the list of the arguments in the super class constructor. This must be the first statement inside the subclass constructor. For example,

```
// BoxWeight now uses super to initialize its Box attributes.

class Box
{
        double width,height,depth;

        //superclass constructor
        Box(double x,double y,double z)
        {
                width=x;height=y;depth=z;
        }
class BoxWeight extends Box
{
double weight; // weight of box

// initialize width, height, and depth using super()
        BoxWeight(double w, double h, double d, double m)
        {
                super(w, h, d); // call super class constructor
                weight = m;
        }
}
```

Using the super to access the super class members (methods or instance variable)

The second form of **super** acts somewhat like **this**, except that it always refers to the super class of the subclass in which it is used. This usage has the following general form:

```
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 super class. Consider this simple class hierarchy:

Write a JAVA program give example for "super" keyword. (Lab Exercise - 6 (a))

```
class A
{
        int i;
        void show()
        {
                System.out.println(" i in A is: "+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()
        {
                super.show(); //calling the super class method

                System.out.println("i in A : " + super.i); // accessing the super class variable

                System.out.println("i in B : " + i);
        }
}
class UseSuper
{
public static void main(String args[])
{
B subOb = new B(1, 2);
subOb.show();
```

Output:

```
E:\DHK>javac
UseSuper.java
E:\DHK>java UseSuper
i in A is: 1
i in A : 1
i in B : 2
```

Note: In the above program the super and sub classes have common names for variables and methods. When we want to execute the super class method, at run time actually the sub class method is executed, because of ***method overriding***. To overcome this problem and ***hide*** the sub class members from the super class members, the keyword super is used with help of .(dot) operator along with member.

final keyword

The "***final***" keyword is used for the following purposes.

- to declare constants
- to prevent method overriding
- to prevent inheritance.

When a variable is declared as final through the program its value should not be changed by the program statement. If any modification is done on the final variable, that can lead to error, while compiling the program.

Example program: demonstrating (1) and (2)

FinalTest.java

```
import java.io.*;
class A
{
        final int MAX=100; // (1) constant declaration

final   void disp() //(2) prevents overriding

        {
                // MAX++ or MAX-- operations are illegal
        System.out.println("The super class disp method MAX is:"+MAX);

        }
}
class B extends A
{
        void disp()
        {
        System.out.println("The SUB class disp method :");
        }

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

Output:

```
E:\DHK>javac FinalTest.java
FinalTest.java:14: disp() in B cannot override disp() in A; overridden method is
 final
      void disp()
           ^
1 error
```

Explanation:

In the above program, the super class method is declared as final, and hence the sup class cannot override this. So we should not redefine the same method in the sub class. If we do so, it leads to an error. If we do the same program without disp() method in the sub class, it will produce the following output.

Example Program: removing the disp() method in the sub class

```
import java.io.*;
class A
{
        final int MAX=100; // (1) constant declaration
        final   void disp() //(2) prevents overriding

        {
                // MAX++ or MAX-- operations are illegal
        System.out.println("The super class disp method MAX is:"+MAX);

        }
}
class B extends A
{       /*void disp()   multiple comments

        {
        System.out.println("The SUB class disp method :");
        }*/
}
class FinalTest
{       public static void main(String args[])
        {       B b=new B();
                b.disp();
```

Output:

```
E:\DHK>javac
FinalTest.java
E:\DHK>java  FinalTest
```

<u>Example program: Demonstrating "final" to Prevent Inheritance</u>

Sometimes you 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. As you might expect, 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.
Here is an example of a **final** class:

```
final class A

{
        // ...
}
// The following class is illegal.

class B extends A
{
        // ERROR! Can't subclass A
        // ...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

Method overriding

In the Inheritance, when a method in the sub class has the same name as the super class method name and signature, then the method in the sub class is executed. The method in the sub class is said to be override the method in the super class. The super class version of the method is hidden. This is called "***method overriding***".

Example Program on method overriding

```
class A
{
        void disp()
        {
                System.out.println("Method of class A");
        }
}
class B extends A
{
        void disp()
        {
                System.out.println("Method of class B");
        }
}
class MOTest
{
        public static void main(String args[])
        {
                B b=new B();
                b.disp();
        }
}
```

Output:

```
E:\DHK>javac
MOTest.java
E:\DHK>java MOTest
```

Using Abstract Classes

Sometimes it may be need by the super class to define the structure of the every method without implementing it. The subclass can fill or implement the method according to its requirements. This kind of situation can come into picture whenever the super class unable to implement the meaningful implementation of the method. For example, if we want to find the area of the Figure given, which can be Circle, Rectangle, and Triangle. The **class Figure** defines the method area(), when subclass implements its code, it implements its own version of the method. The Java's solution to this problem is **abstract method.**

To declare anabstract method, use this general form:

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

As you can see, no method body is present.

To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keywordat 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 wouldbe useless, because an abstract class is not fully defined.

Example Program:

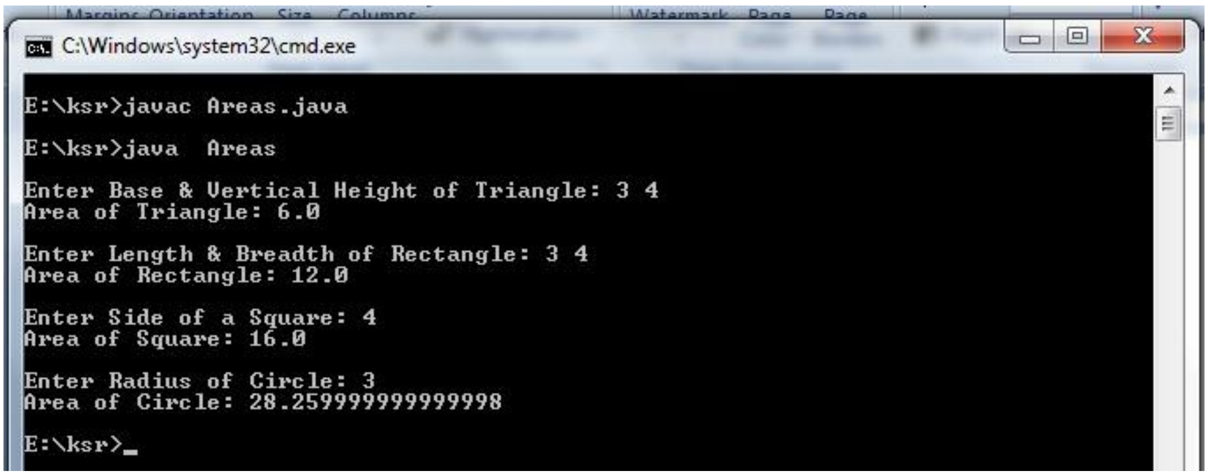
FigureDemo.java

c). Write a java program for abstract class to find areas of different shapes (Exercise 5 (c))

```
import java.util.Scanner;

abstract class CalcAreas
{
    abstract void findTriangle(double b, double h);
    abstract void findRectangle(double l, double b);
    abstract void findSquare(double s);
    abstract void findCircle(double r);
}

class FindArea extends CalcAreas
{
     void findTriangle(double b, double h)
    {
        double area = (b*h)/2;
        System.out.println("Area of Triangle: "+area);
    }
```

```
     void findRectangle(double l, double b)
    {
       double area = l*b;
       System.out.println("Area of Rectangle: "+area);
    }
     void findSquare(double s)
    {
       double area = s*s;
       System.out.println("Area of Square: "+area);
    }
     void findCircle(double r)
    {
       double area = 3.14*r*r;
       System.out.println("Area of Circle: "+area);
    }
}
class Areas
{
    public static void main(String args[])
    {
       double l, b, h, r, s;
       FindArea area = new FindArea();
       Scanner get = new Scanner(System.in);
       System.out.print("\nEnter Base & Vertical Height of Triangle: ");
       b = get.nextDouble();
       h = get.nextDouble();
       area.findTriangle(b, h);
       System.out.print("\nEnter Length & Breadth of Rectangle: ");
       l = get.nextDouble();
       b = get.nextDouble();
       area.findRectangle(l, b);
       System.out.print("\nEnter Side of a Square: ");
       s = get.nextDouble();
       area.findSquare(s);
       System.out.print("\nEnter Radius of Circle: ");
       r = get.nextDouble();
       area.findCircle(r);
    }
}
```

Output:



```
E:\ksr>javac Areas.java

E:\ksr>java  Areas

Enter Base & Vertical Height of Triangle: 3 4
Area of Triangle: 6.0

Enter Length & Breadth of Rectangle: 3 4
Area of Rectangle: 12.0

Enter Side of a Square: 4
Area of Square: 16.0

Enter Radius of Circle: 3
Area of Circle: 28.259999999999998

E:\ksr>_
```

<u>Dynamic Method Dispatch (Runtime Polymorphism) (Topic Beyond Syllabus)</u>

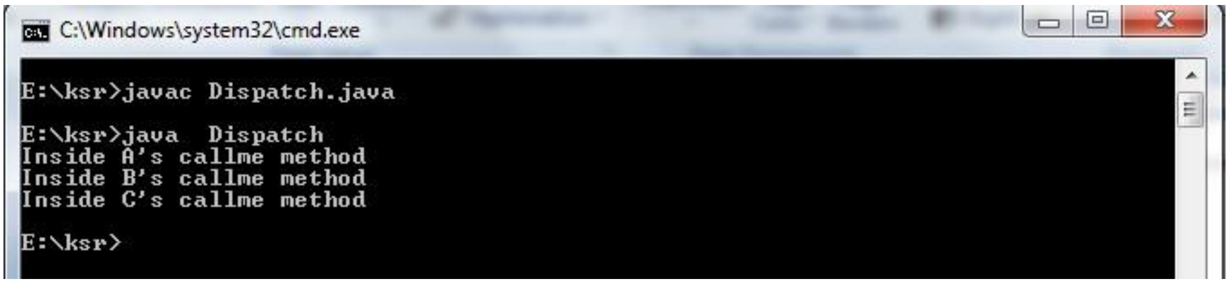
Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements **run-time polymorphism.**

Let's begin by restating an important principle: a super class reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a super class 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.
Here is an example that illustrates dynamic method dispatch:
Write a JAVA program that implements Runtime polymorphism class A (Exercise 8 (a))

```
{
        void callme()
        {
                System.out.println("Inside A's callme method");
        }
}
class B extends A
{
        // override callme()
        void callme()
        {
                System.out.println("Inside B's callme method");
        }
}
class C extends A
{
        // override callme()
        void callme()
        {
                System.out.println("Inside C's callme method");
        }
}
```

```
class Dispatch
{
        public static void main(String args[])
        {
        A a = new A(); // object of type A
        B b = new B(); // object of type B
        C c = new C(); // object of type C
        A r; // obtain a reference of type A
        r = a; // r refers to an A object
        r.callme(); // calls A's version of callme
        r = b; // r refers to a B object
        r.callme(); // calls B's version of callme
        r = c; // r refers to a C object
        r.callme(); // calls C's version of callme
        }
}
```

The output from the program is shown here:



```
E:\ksr>javac Dispatch.java

E:\ksr>java  Dispatch
Inside A's callme method
Inside B's callme method
Inside C's callme method

E:\ksr>
```

Introduction to Interfaces

Java supports the concept of Inheritance, that is acquiring the properties from one class to other class. The class that acquires properties is called **"subclass"**, and the class from which it acquire is called "**super class**". Here, one class can acquire properties from other class using the following statement:

```
class A extends B

{

---------

---------

}
```

But, Java does not allow to acquire properties from more than one class, which we call it as "**multiple inheritance**". We know that large number of real-life applications require the use of multiple inheritance. Java provides an alternative approach known as "***interface***" to support the concept of multiple inheritance.

Defining Interface

An interface is basically a kind of class. Like classes, interfaces contain the methods and variables but with major difference. The difference is that interface define only abstract methods and final fields. This means that interface do not specify any code to implement these methods and data fields contain only constants. Therefore, it is the responsibility of the class that implements an interface to define the code for these methods.

The syntax of defining an interface is very similar to that of class. The general form of an interface will be as follows:

```
        interface Intyerface_name
{
        //variables inside interface are by default final, publicand static
        type id=value;

        //by default abstract methods

        return_type method_name(paprameter_list);
        return_type method_name(paprameter_list);
        return_type method_name(paprameter_list);
```

Note: 1) ***variables inside interface are by default final, public and static***

2) by default methods are public and abstract

These methods must be implemented any class that want to acquire the properties

Here is an example of an interface definition that contain two variable and one method

```
interface Calculator
{
        //variables inside interface are by default final, public and static
        double PI=3.14;

        //by default abstract methods
        int add(int a,int b);

        int sub(int a,int b);
        int mul(int a,int b);
```

Implementing the interface

Interfaces are used as "**superclasses**" whose properties are inherited by the classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

```
class Class_Name      implements     Interface_Name

          //Body of the class
}
```

Here, Class_Name class, implements the interface "Interface_Name". A more general form of implementation may look like this:

```
class     Class_Name       extends    Superclass_Name implements interface1,
interface2,interface3..

{

[illegible]
```

This shows that a class can extend another class while implementing interfaces. When a class implements more than one interface they are separated by a comma.

Example program using interface

InterfaceTest.Java

```
interface Calculator

{

        //variables inside interface are by default final, publi and static
        double PI=3.14;

        //by default abstract methods
        int add(int a,int b);

        int sub(int a,int b);
        int mul(int a,int b);
        int div(int a,int b);
        double area(int r);

}

class NormCal implements Calculator

{public int add(int x,int y)

        {

        return(x+y);

        }

        public int sub(int x,int y)

        {

        return(x-y);

        }

        public int mul(int x,int y)

        {
```

```
        System.out.println("The sum is:"+nc.add(2,3));
        System.out.println("The sum is:"+nc.sub(2,3));
        System.out.println("The sum is:"+nc.mul(2,3));
        System.out.println("The sum is:"+nc.div(4,2));
        System.out.println("area of Circle is:"+nc.area(5));
```

OutPut:

```
E:\DHK>javac
NormCal.java
E:\DHK>java NormCal

The sum is:5
The sum is:-1
The sum is:6
```

Extending the Interfaces

Like classes interfaces also can be extended. That is, an interface can be sub interfaced from other interface. The new sub interface will inherit all the members from the super interface in the manner similar to the subclass. This is achieved using the keyword extends as shown here:

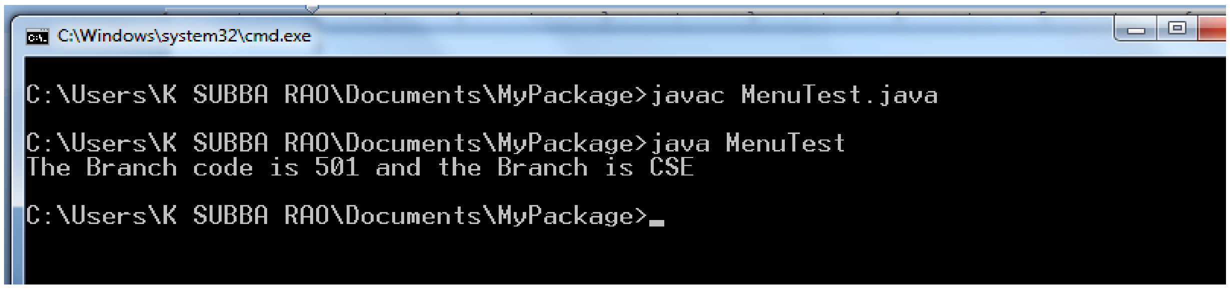
```
        interface       Interface_Name1         extendsInterface_Name2

                //Body of the Interface_name1
        }
```

For example,   **MenuTest.java**

```
interface Const

{

        static final int code=501;

        static final String branch="CSE";

}

interface Item extends Const

{

        void display();

}

class Menu implements Item
```

```
                Menu m=new Menu(); // this contains the interfaces
                Item and Const m.display();

        }
}
```

Output:



```
C:\Users\K SUBBA RAO\Documents\MyPackage>javac MenuTest.java

C:\Users\K SUBBA RAO\Documents\MyPackage>java MenuTest
The Branch code is 501 and the Branch is CSE

C:\Users\K SUBBA RAO\Documents\MyPackage>_
```

Interfaces Vs Abstract classes

Sl	Interface	Abstract
1	Multiple Inheritance possible	Multiple Inheritance not possible
2	***implements*** keyword is used	***extends*** keyword is used
3	By default all the methods are public, and abstract. No need to tag as public and abstract	Methods have to be tagged as public and abstract.
4	All methods of interface need to be overridden	Only abstract methods need to be overridden
5	All variable declared in interface are By default public, final and static	Variable if required, need to be declared in interface as public, final and static
6	Methods cannot be static	Non-abstract methods can be static