UNIT III

Arrays: Introduction, Declaration and Initialization of Arrays, Storage of Array in Computer Memory, Accessing Elements of Arrays, Operations on Array Elements, Assigning Array to Another Array, Dynamic Change of Array Size, Sorting of Arrays, Search for Values in Arrays, Class Arrays, Two-dimensional Arrays, Arrays of Varying Lengths, Three-dimensional Arrays, Arrays as Vectors.

Inheritance: Introduction, Process of Inheritance, Types of Inheritances, Universal Super Class-Object Class, Inhibiting Inheritance of Class Using Final, Access Control and Inheritance, Multilevel Inheritance, Application of Keyword Super, Constructor Method and Inheritance, Method Overriding, Dynamic Method Dispatch, Abstract Classes, Interfaces and Inheritance.

Interfaces: Introduction, Declaration of Interface, Implementation of Interface, Multiple Interfaces, Nested Interfaces, Inheritance of Interfaces, Default Methods in Interfaces, Static Methods in Interface, Functional Interfaces, Annotations.

Array:

- Group of similar elements referred with a single name.
- In the Java programming language *arrays* are objects, are dynamically created, and may be assigned to variables of type Object.
- All methods of class Object may be invoked on an array.
- There are two types of arrays.
  - Single Dimensional Array
  - Multidimensional Array

Advantages

1. Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.
2. Random access: We can get any data located at an index position.

Disadvantages

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

Declaration and Initialization of single dimensional Arrays:

Syntax to Declaration an Array:

Syntax: - datatype variableName[size];

```
                 (or)
        datatype[size] variableName
                 (or)
        datatype []variableName;
```

Ex: - int arr[10];

(or)
int[10] arr;
(or)
int [10]arr;

Note: The array arr is initially set to null. *new* is a special operator that allocate memory.

Instantiation or initialization of an Array in Java

Syntax: arrayVariableName=new datatype[size];

Example:

```
int arr[10];   // decalration
Arr=new int[10]  //instantiation or initialization of array without values
```

(Or)

```
int  arr[]=new int[10];  //declaration and instantiation/ initialization of array without values in single line
```

Initialization of an Array with values:

```
int  arr[]=new int[10]{10,20,30,40,50,60,70,80,90,100};
```

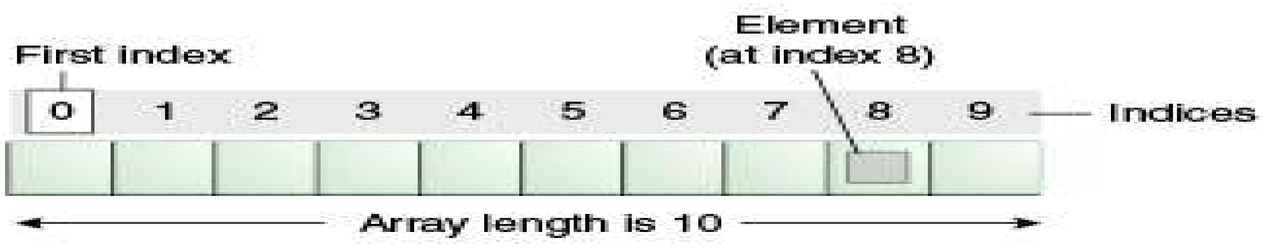
(or)

```
int  arr[]={10,20,30,40,50,60,70,80,90,100};
```

Note: If the array is initialized at the time of declaration then new key word is not required.

Storage of Array in Computer Memory:

- The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements.
- Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on and index ends with array-size-1.



Accessing Elements of Arrays:

- Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on and index ends with array-size-1.
- we can get the length of the array using the length member.
- With help of **arrayName[index]** syntax we can access individual elements of an array.

  Example:

arrayName=a	10	20	30	40	50	60	70	80	90	100
index	0	1	2	3	4	5	6	7	8	9

a [3] =40
a [6] =70
a [9] =100

Operations on Array Elements

- reading data into the array and accessing array data (Traversing)

  Example:

```
        import java.util.Scanner;
        class Array2
        {
                public static void main(String[] args)
                {
                        int a[]=new int[10]; // declaring and instatiation of an array
                        Scanner s=new Scanner(System.in);
                        for(int i=0;i<a.length;i++) //reading element into an
array(a.length gives no of elements in the array)
        {
                System.out.println("Enter Value at index "+i);
                a[i]=s.nextInt();
        }
        int i=0;
        for(int element:a) // accessing or traversing array using for each loop
        {
                System.out.println("Element at index "+i+" is: "+element);
                i=i+1;
        }
    }
}
```

- finding the largest and smallest element in the array.
- find the sum and average of the elements of the array.
- Sorting
- Searching
- Insertion − Adds an element at the given index.

- Deletion − Deletes an element at the given index.
- Update − Updates an element at the given index.

Example:

```
import java.util.ArrayList;
import java.util.Scanner;
class ArrayDynamic
{
        public static void main(String args[])
        {
                ArrayList<Integer> a=new ArrayList<Integer>();
                Scanner s=new Scanner(System.in);
                char ch;
                do
                {
                        System.out.println("********Dynamic Array********");
                        System.out.println("Choose any option");
                        System.out.println("1.insert element into array");
                        System.out.println("2.remove element from array");
                        System.out.println("3.clear all element of an array ");
                        System.out.println("4.do you wanyt to know size?");
                        System.out.println("5.get element from the array");
                        System.out.println("6.Print element of the dynamic array");
                        int option=s.nextInt();
                        switch(option)
                        {
                                case 1:System.out.println("Enter element to insert:");
                                    int ele=s.nextInt();
                                    a.add(ele);
                                    break;
                                case 2:System.out.println("Enter index of the element to remove:");
                                    int index=s.nextInt();
                                    a.remove(index);
                                    break;
                                case 3:a.clear();
                                    break;
                                case 4:System.out.println("Size of the array is-"+a.size());
                                    break;
                                case 5:System.out.println("Enter the index to know the element:");
                                    int index1=s.nextInt();
                                    System.out.println("Element at index-"+index1+"is"+a.get(index1));
                                    break;
                                case 6:System.out.println("Element of the dynamic array:");
                                    System.out.println(a);
                                    break;
                                default:System.out.println("Invalid/Wrong Option choosen");
                                        break;

                        }
                        System.out.println("do you want to continue(Y/y or N/n)");
                        ch=s.next().charAt(0);
                }while(ch=='Y' || ch=='y');
        }
}
```

Assigning Array to Another Array

It is nothing but copying one array element to another array.

Example:

Assigning(copying) one array element to another array:

```
import java.util.Scanner;
class ArrayCopy
{
        public static void main(String args[])
        {
                int a1[]=new int[10];
                int a2[]=new int[a1.length];
                Scanner s=new Scanner(System.in);
                for(int i=0;i<a1.length;i++)
                {
                        System.out.println("Enter Value at index "+i);
                        a1[i]=s.nextInt();
                }
                for(int i=0;i<a1.length;i++)
                {
                        a2[i]=a1[i];
                }
                System.out.println("Array 1 Elements: ");
                for(int element:a1)
                {
                        System.out.print(element+"\t");
                }
                System.out.println("");
                System.out.println("Array 2 Elements: ");
                for(int element:a2)
                {
                        System.out.print(element+"\t");
                }

        }
}
```

Dynamic Change of Array Size / dynamic array:

- An array is a fixed size, homogeneous **data structure**. The limitation of arrays is that they're fixed in size. It means that we must specify the number of elements while declaring the array.
- The dynamic array is a **variable size** list data structure. It grows automatically when we try to insert an element if there is no more space left for the new element. It allows us to add and remove elements. It allocates memory at run time using the heap. It can change its size during run time.
- In Java, **ArrayList** is a resizable implementation. It implements the List interface and provides all methods related to the list operations.

Example:

```
import java.util.ArrayList;
import java.util.Scanner;
class ArrayDynamic
{
        public static void main(String args[])
        {
                ArrayList<Integer> a=new ArrayList<Integer>();
                Scanner s=new Scanner(System.in);
                char ch;
                do
                {
                        System.out.println("********Dynamic Array********");
                        System.out.println("Choose any option");
                        System.out.println("1.insert element into array");
                        System.out.println("2.remove element from array");
                        System.out.println("3.clear all element of an array ");
                        System.out.println("4.do you wanyt to know size?");
                        System.out.println("5.get element from the array");
                        System.out.println("6.Print element of the dynamic array");
                        int option=s.nextInt();
                        switch(option)
                        {
                                case 1:System.out.println("Enter element to insert:");
                                    int ele=s.nextInt();
                                    a.add(ele);
                                    break;
                                case 2:System.out.println("Enter index of the element to remove:");
                                    int index=s.nextInt();
                                    a.remove(index);
                                    break;
                                case 3:a.clear();
                                    break;
                                case 4:System.out.println("Size of the array is-"+a.size());
                                    break;
                                case 5:System.out.println("Enter the index to know the element:");
```

```
                                        int index1=s.nextInt();
                                        System.out.println("Element at index-"+index1+"is"+a.get(index1));
                                        break;
                                    case 6:System.out.println("Element of the dynamic array:");
                                        System.out.println(a);
                                        break;
                                    default:System.out.println("Invalid/Wrong Option choosen");
                                            break;

                            }
                            System.out.println("do you want to continue(Y/y or N/n)");
                            ch=s.next().charAt(0);
                    }while(ch=='Y' || ch=='y');
        }
}
```

Sorting of Arrays:

Example 1:

//sorting of an array using sort() methodin Ascending order

```
import java.util.Arrays;
class SortByMethod
{
        public static void main(String args[])
        {
                int a[]={20,5,34,789,123,3,5};
                System.out.println("Array Before Sorting: ");
                for(int i=0;i<a.length;i++)
                        System.out.print(a[i]+"\t");
                Arrays.sort(a); //sort() is a static method of Arrays class and it perform sorting
                System.out.println();
                System.out.println("Array After Sorting: ");
                for(int i=0;i<a.length;i++)
                        System.out.print(a[i]+"\t");
        }
}
```

Example 2:

//sorting of an array using sort() methodin Descending order

```
import java.util.Arrays;
import java.util.Collections;
class SortByMethod1
{
        public static void main(String args[])
```

```
	{
		Integer a[]={20,5,34,789,123,3,5};
		System.out.println("Array Before Sorting: ");
		for(int i=0;i<a.length;i++)
			System.out.print(a[i]+"\t");
		//Arrays.sort(a,Collections.reverseOrder());
		Arrays.sort(a, Collections.reverseOrder());
		System.out.println();
		System.out.println("Array After Sorting: ");
		System.out.println(Arrays.toString(a));

	}
}
```

<u>Example 3:</u>

```
//sorting of an array in ascending and descending oder without using sort() method.
class SortingAscDes
{
	public static void main(String args[])
	{
		int a[]={10,345,2,789,45,34};
		System.out.println("Array Before Swapping: ");
		for(int element:a)
			System.out.print(element+"\t");
		for(int i=0;i<a.length;i++)
		{
			for(int j=i+1;j<a.length;j++)
			{
				if(a[i]>a[j])
				{
					int temp=a[i];
					a[i]=a[j];
					a[j]=temp;
				}
			}
		}
		System.out.println("\nArray After Swapping in Ascending Order: ");
		for(int element:a)
			System.out.print(element+"\t");
		for(int i=0;i<a.length;i++)
		{
			for(int j=i+1;j<a.length;j++)
			{
```

```
                    if(a[i]<a[j])
                    {
                        int temp=a[i];
                        a[i]=a[j];
                        a[j]=temp;
                    }
                }
            }
            System.out.println("\nArray After Swapping in decsending Order: ");
            for(int element:a)
                System.out.print(element+"\t");
        }
}
```

Search for Values in Arrays:

Example 1: using binarySearch() Method

```
/*
public static int binarySearch(int[] a, int key)
        or
public static int binarySearch(int[] a, int fromIndex, int toIndex, int key)
*/
import java.util.Scanner;
import java.util.Arrays;
class ArraysClassMethods
{
        public static void main(String args[])
        {
            Scanner s=new Scanner(System.in);
            System.out.println("Enter size of the array: ");
            int n=s.nextInt();
            int a[]=new int[n];
            for(int i=0;i<a.length;i++)
            {
                System.out.println("Enter element at index-"+i);
                a[i]=s.nextInt();
            }
            Arrays.sort(a);
            for(int i=0;i<a.length;i++)
            {
                System.out.print(a[i]+"\t");
            }
            System.out.println("\nEnter which element you want to search : ");
```

```
                int search=s.nextInt();
                int index=Arrays.binarySearch(a,search);
                System.out.println("Element found at index-"+index);
                int index1=Arrays.binarySearch(a,4,6,search);
                System.out.println("Element found at index-"+index1);

        }
}
```

Example 2: without using binarySearch() Method

```
//Binary Search in java
import java.util.Scanner;
import java.util.Arrays;
class SearchBinary
{
        public static void main(String args[])
        {
                int a[]={2,40,7,5,75,1,20};
                Arrays.sort(a);
                Scanner s=new Scanner(System.in);
                System.out.println("Enter which element you want to search: ");
                int search=s.nextInt();
                int first=0,last=a.length-1;
                int mid=(first+last)/2,status=0;
                while(first<=last)
                {
                        if(search>a[mid])
                                first=mid+1;
                        else if (search<a[mid])
                                last=mid-1;
                        else if(search==a[mid])
                        {
                                status=1;
                                break;
                        }
                        mid=(first+last)/2;
                }
                if(status==1)
                        System.out.println("Element found in the array at index-"+mid);

                else
```

```
                        System.out.println("Element not found in the array");
        }
}
```

Class Arrays:

The **java.util.Arrays** class contains a static factory that allows arrays to be viewed as lists.Following are the important points about Arrays −

- This class contains various methods for manipulating arrays (such as sorting and searching).
- The methods in this class throw a NullPointerException if the specified array reference is null.

Array class methods:

- asList()
- binarySearch()
- copyOf()
- deepEquals()
- equals()
- hashCode()
- fill()
- sort()
- tostring

Examples: fill () method

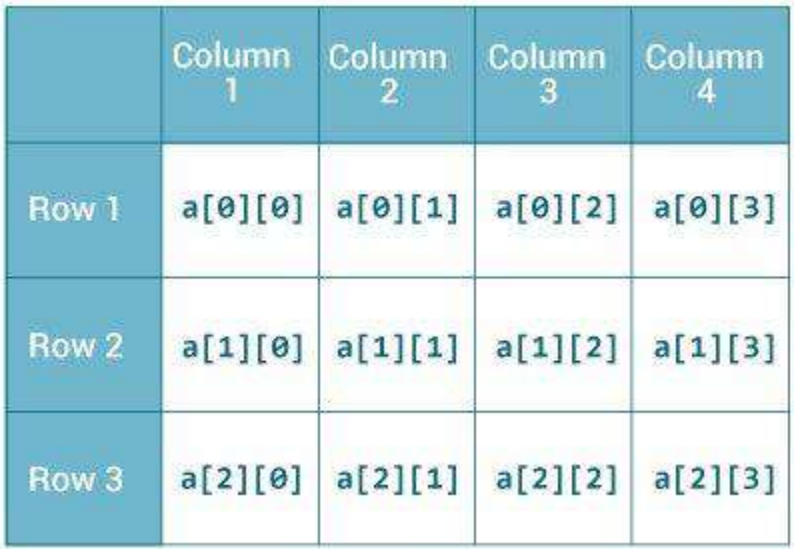
```
/*
public static void fill(array_name,value_to_fill);
public static void fill(array_name,from_index,to_index,value_to_fill);
*/
import java.util.Arrays;
class ArraysFill
{
        public static void main(String args[])
        {
                int a[]={10,20,30,40,50,60};
                int a1[]=Arrays.copyOf(a,6);
                System.out.println("Array elements before fill operation:");
                for(int ele:a)
                        System.out.print(ele+"\t");
                Arrays.fill(a,2);
                System.out.println("\nArray elements after fill operation:");
                for(int ele:a)
                        System.out.print(ele+"\t");
                Arrays.fill(a1,2,4,100);
                System.out.println("\nArray elements after fill operation:");
                for(int ele:a1)
                        System.out.print(ele+"\t");
        }
}
```

Examples: asList () method

```
import java.util.Arrays;
import java.util.List;
class ArraysDemo {
  public static void main (String args[]) {
    // create an array of strings
    String a[] = {"abc","klm","xyz","pqr"};

    List list1 = Arrays.asList(a);

    // printing the list
    System.out.println("The list is:" + list1);
  }
}
```

Two-dimensional Arrays:

- It is a collection of rows and columns
- Declaration and instantiation :
  - Synatx: data_type arrayname[][]=new data_type[row_size][column_size];
  - Example: int a[][] =new int[3][4];



	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Example:

```
        import java.util.Scanner;
        class TwoDArray
        {
                Public static void main(String[] args)
                {
                        int[][] arr = newint[3][3];
                        Scanner sc = new Scanner(System.in);
                        for (inti =0;i<3;i++)
                        {
                           for(intj=0;j<3;j++)
                           {
                              System.out.print("Enter Element");
                              arr[i][j]=sc.nextInt();
                              System.out.println();
                           }
                        }
                        System.out.println("Printing Elements...");
                        for(inti=0;i<3;i++)
                        {
                           System.out.println();
                           for(intj=0;j<3;j++)
                           {
                              System.out.print(arr[i][j]+"\t");
                           }
                        }
                }
        }
```

Arrays of Varying Lengths:

```
//Arrays of Varying Lengths

import java.util.Scanner;

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

        {

                System.out.println("Enter the size of the array:");

                Scanner s=new Scanner(System.in);

                int n=s.nextInt();

                int a[]=new int[n];

                System.out.println("Enter the"+n+"array elements");

                for(int i=0;i<a.length;i++)

                {

                        System.out.println("Enter element at index-"+i);

                        a[i]=s.nextInt();

                }
                System.out.println("array contain following elements");

                for(int ele:a)

                        System.out.print(ele+"\t");

        }

}
```

Three-dimensional Arrays:

```
import java.util.Scanner;
class Three
{
        public static void main(String args[])
        {
                Scanner s=new Scanner(System.in);
                int a[][][]=new int[3][3][3];   //3-D array declaration and intialization
                for(int i=0;i<3;i++)     // reading data into 3-D array
                {
                        for(int j=0;j<3;j++)
                        {
                                for(int k=0;k<3;k++)
                                {
                                        System.out.println("Enter element at index:"+i+j+k);
                                        a[i][j][k]=s.nextInt();
                                }
                        }
                }
                for(int i=0;i<3;i++)     //printing data of 3-D array
                {
                        for(int j=0;j<3;j++)
                        {
                                for(int k=0;k<3;k++)
                                {
                                        System.out.print(a[i][j][k]+"\t");
                                }
                        }
                }

        }
}
```

Arrays as Vectors:

Arrays	Vectors
The length of an array is fixed once it is created, and elements cannot be added or removed before its creation.	A Vector is a resizable-array that works by reallocating storage and copying the old array elements to a new array.
In arrays are retrieval and assignment operations will be fast.	Vector is relatively slow.
An array does not reserve any additional storage.	An vector reserve any additional storage.
Java arrays can hold both primitive data types (int, char, long, etc.) and Java objects (Integer, Character, Long, etc.),	A Vector can hold only Java objects. (Integer, Character, Long, etc.)
An array has a length property that stores its length.	To find the size of the Vector, we can call its size() method.
The dimension of an array is the total number of indices needed to select an element. Arrays in Java supports single-dimensional array as well as multidimensional arrays.	A Vector has no concept of dimensions, but we can easily construct a Vector of Vectors.

Inheritance:

Introduction

- The mechanism of deriving a new class from an old one is called inheritance.
- When a class is written by a programmer and another programmer wants the same features(members) in his class also, then the other programmer can go for inheritance.
- Reusability is another feature of OOP.
- The old class is referred as the super class and the new class is sub class.
- A sub class inherits some or all of the data from the super class.
- *extends* is the keyword which is used for inheritance.

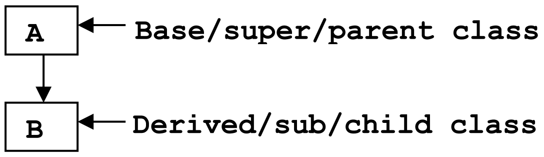
Process of Inheritance:

```
Syntax:
class <sub class name> extends <super class name>
{
 body
}
```

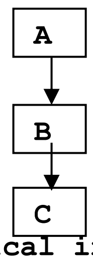
Types of Inheritances:

Java Supports 4 types of inheritance:

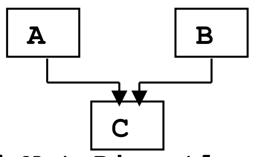
1. Single inheritance



3. Multilevel inheritance

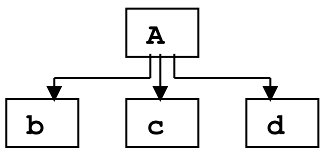


2. Multiple inheritance



(Not Directly)

4. Hierarchical inheritance



Single inheritance
Examples:

```
class Parent
{
  int a=10;
}
class Child extends Parent
{
  int b=20;
}
class Single
{
  public static void main(String args[])
  {
    Child c=new Child();
    System.out.println("Value of Parent:"+c.a);
    System.out.println("Value of Child:"+c.b);
  }
}
```

Multi-Level Inheritance:

```
class Animal
{
  void eat()
  {
    System.out.println("Eating...");
  }
}
class Dog extends Animal
{
 void bark()
 {
   System.out.println("Barking...");
 }
}
class BabyDog extends Dog
{
 void weep()
 {
   System.out.println("weeping...");
 }
}
class MultiLevel
{
 public static void main(String args[])
 {
   BabyDog b=new BabyDog();
   b.weep();
   b.bark();
   b.eat();
 }}
```

Hierarchical inheritance:

```
class Animal
{
  void eat()
  {
    System.out.println("Eating");
  }
}
class Cat extends Animal
{
  void meow()
  {
    System.out.println("meowingggg");
  }
}
class Dog extends Animal
{
  void bark()
  {
    System.out.println("barking");
  }
}
class Tiger extends Animal
{
  void roar()
  {
    System.out.println("roaringgg");
  }
}
class Hier
{
  public static void main(String args[])
  {
    Tiger t=new Tiger();
    t.eat();
    t.roar();
    Dog d=new Dog();
    d.eat();
    d.bark();
    Cat c=new Cat();
    c.eat();
    c.meow();
  }
}
```

Note: super class and sub class are independent classes, they can be used separately also.
Note: Multiple inheritance is also known as hybrid inheritance. Directly we can’t implement by extend keyword. By using interfaces we can implement multiple inheritance.

<u>Application of Keyword Super</u>:

In the method overriding examples, Data members of <u>super class</u> are accessed by sub class directly because they are public (default access), but when the data members of the super class are declared as private then they cannot be accessed/initialized by sub class. Since encapsulation is the primary attribute of OOP. So this problem can be solved by using the keyword <u>super.</u>

<u>super</u> keyword is used by subclass to refer to its immediate super class.

<u>Super keyword is used in two ways:</u>

- To call super class Constructors.
- To call super class members.

<u>Invoking super class constructors in Subclass:</u>

- A sub class constructor is used to construct instance variables of both sub class and super class.
- The sub class constructor uses the keyword super to invoke the constructor of the super class.
- Conditions for invoking super keyword
  - Super must be used with in a subclass constructor method.
  - Super must be the first statement in subclass constructor.
  - The parameters in the super call(subclass) should match the order and type of the constructor of the superclass.

Syntax:- super(parameter-list);

Example:

```
class Room
 {
   private int length;
   private int breadth;

   Room(int l,int b)
     {
      length=l;
      breadth=b;
     }
   int area()
     {
      return(length*breadth);
     }
 }
 class BedRoom extends Room // inheriting room
 {
  int height;
  BedRoom(int x,int y,int h) // private members are not inherited so
  {                          // accessed through super
   super(x,y);
   height=h;
  }
  int volume()
  {
   return( (area()*height) );
   }
 }
 class SuperConsDemo
 {
  public static void main(String args[])
  {
   BedRoom r1=new BedRoom(10,20,5);

   System.out.println("Area is :"+r1.area());
   System.out.println("Volume is:"+r1.volume());
   }
  }
```

Invoking super class members:

The super keyword is used when sub class members hides super class members (same name in super class and sub class).

Syntax: - *super.member;*

Here member is either method or instance variable.

```
class A
{
 int i;
 void disp()
  {
   System.out.println("Super class Disp....");
  }
}
class B extends A
 {
  int i; // this i hides super class i
  B(int a,int b)
  { super.i=a;
    this.i=b;
  }
  void disp()
  {  super.disp();
     System.out.println("Sub class Disp...");
     System.out.println("I in Super class..:"+super.i);
     System.out.println("I in Sub class....:"+this.i);
     }
  }
 class SuperMemberDemo
  {
   public static void main(String args[])
   { B sob=new B(10,20);
     sob.disp();
   }
  }
```

Inhibiting Inheritance of Class Using Final:

- Classes declared as final cannot be inherited.
- A final class implicitly declares all of its methods as final.
- It is illegal to declare a class as abstract and final.

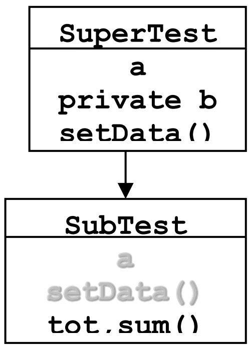
Ex:

```
final class A
{
 ----
 ----
 }

class B extends A // cannot be inherited
{ ---
  ----
}
```

Access Control and Inheritance:

Access specifier in inheritance: Although a subclass includes all of the members of its super class, it cannot access those members of the super class that have been declared as private.

```
class SuperTest
{
 int a; // public by default
 private int b; //private
 void setData(int x,int y)
 {
  a=x;
  b=y;
  }
}
class SubTest extends SuperTest
{
 int tot;
 void sum()
 {
  tot=a+b; // b is private, not avalible in sub class
 }
};
class Access
{
 public static void main(String args[])
 {
  SubTest s=new SubTest();
  s.setData(10,20);
  s.sum();
```



```
  System.out.println("Total is :"+s.tot);
 }
};
```

The above program gives the following error:

```
Access.java:18: b has private access in SuperTest
                tot=a+b; // b is private, not avalible in sub class
                      ^
1 error
```

Note: Variable b is declared as private, it is only accessible by other members of its own class. Sub classes have no access.

The Protected Specifier:

The private members of the super class are not available to sub classes directly. But sometimes, there may be a need to access the data of super class in the sub class. For this purpose, protected specifier is used.

```
class Access
{
 protected int a;
 protected int b;
}

class Sub extends Access
{
 public void get()
 {
  System.out.println(a);
  System.out.println(b);
 }
}

class ProtectedTest
{
 public static void main(String args[])
 {
  Sub s=new Sub();
  s.get();
}
}
```

Universal Super Class-Object Class:

- The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Methods of Object class

The Object class provides many methods. They are as follows:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Constructor Method and Inheritance:

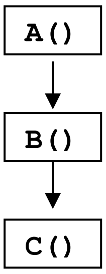
Constructors are called in the order in which they created.

```
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();
            }
}
```

Note:

- To execute super class constructor super() must be the first statement executed in a subclass constructor.
- But if super() is not used, then the default or parameter less constructor of each superclass will be executed.



Method/Function overriding:

It is the procedure of defining a functionality in the sub-class with the same signature of a function that is already existing in the super class.

```
class Super
{
             int i,j;
             void disp1()
             {
                 System.out.println("Inside disp1 of Super..");
             }
             void disp2()
            {
                System.out.println("Inside disp2 of Super...");
             }
}
class Sub extends Super
{
             int x,j;
            void disp1()
           {
                System.out.println("Inside disp1 of Sub..");
            }
            void disp3()
            {
```

```
            System.out.println("Inside disp1 of Sub..");
         }
         public static void main(String args[])
        {
            Sub s=new Sub();
            s.i=10; // i of Super class is changed
            s.j=20; // j of Sub class is changed
            s.disp1();
            s.disp2();
        }
}
```

Note:
In the above program disp1() is defined in both Super and Sub classes, when s.disp1() is called disp1() function present in Sub class is executed. This is because local member have highest priority.

Even though same variables/functions are present in both the object, they are not available with the same priority. So there is no ambiguity.

Dynamic Method Dispatch: Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

```
class A
{
   void m1()
   {
      System.out.println("Inside A's m1 method");
   }
}

class B extends A
{
   // overriding m1()
   void m1()
   {
      System.out.println("Inside B's m1 method");
   }
}

class C extends A
{
   // overriding m1()
   void m1()
   {
      System.out.println("Inside C's m1 method");
   }
}

// Driver class
class Dispatch
{
    public static void main(String args[])
    {
        // object of type A
        A a = new A();

        // object of type B
        B b = new B();

        // object of type C
        C c = new C();

        // obtain a reference of type A
        A ref;

        // ref refers to an A object
        ref = a;

        // calling A's version of m1()
        ref.m1();

        // now ref refers to a B object
        ref = b;

        // calling B's version of m1()
        ref.m1();

        // now ref refers to a C object
        ref = c;

        // calling C's version of m1()
        ref.m1();
    }
}
```

Abstract Classes:

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract (method with declaration only) and non-abstract methods (method with the body).

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Program:

```
abstract class MyClass
{
 abstract void calculate(double x);
}

class Square extends MyClass
{
 void calculate(double x)
 {
  System.out.println("Square value = "+(x*x));
 }
}

class SquareRoot extends MyClass
{
 void calculate(double x)
 {
  System.out.println("Square root value = "+Math.sqrt(x));
 }
}

class Cube extends MyClass
{
 void calculate(double x)
 {
  System.out.println("Cube value = "+(x*x*x));
 }
}
 class Different
 {
  public static void main(String args[])
  {
   Square s=new Square();
   SquareRoot st=new SquareRoot();
   Cube c=new Cube();
   s.calculate(3);
   st.calculate(4);
   c.calculate(5);
  }
 }
```

Interfaces and Inheritance :

Category	Inheritance	Interface
Description	Inheritance is the mechanism in java by which one class is allowed to inherit the features of another class.	Interface is the blueprint of the class. It specifies what a class must do and not how. Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
Use	It is used to get the features of another class.	It is used to provide total abstraction.
Syntax	class subclass_name extends superclass_name { }	interface <interface_name>{ }
Number of Inheritance	It is used to provide 4 types of inheritance. (multi-level, simple, hybrid and hierarchical inheritance)	It is used to provide 1 types of inheritance (multiple).
Keywords	It uses extends keyword.	It uses implements keyword.
Inheritance	We can inherit lesser classes than Interface if we use Inheritance.	We can inherit enormously more classes than Inheritance, if we use Interface.
Method Definition	Methods can be defined inside the class in case of Inheritance.	Methods cannot be defined inside the class in case of Interface (except by using static and default keywords).
Overloading	It overloads the system if we try to extend a lot of classes.	System is not overloaded, no matter how many classes we implement.
Functionality Provided	It does not provide the functionality of loose coupling	It provides the functionality of loose coupling.
Multiple Inheritance	We cannot do multiple inheritance (causes compile time error).	We can do multiple inheritance using interfaces.

Interfaces:

Introduction, Declaration and Implementation of Interface

- Abstract class contains abstract and concrete methods also. But which contains only abstract methods is called Interface.
- We can't create objects for interfaces.

Declaration syntax:

Syntax:-

```
interface InterfaceName
 {
   variables declaration;
   methods declaration;
 }
```

Here interface is the keyword which declares an interface, InterfaceName is any java variable name(just like class name).

Ex:-

```
Interface Item
{
 static final int code=101;
 static final String name=”Pen”;
 void display();
}
```

- Variables declared in interface are implicitly static and final.
- The variables can not be changed by implementing classes.
- The class that implements this interface must define code for the method.
- All the members(variables/methods) are implicitly public, if the interface, itself, is declared as public.

Implementing interfaces:

- interfaces are used as “superclasses” whose properties are inherited by classes.
- The methods that implement an interface must be declared as public.

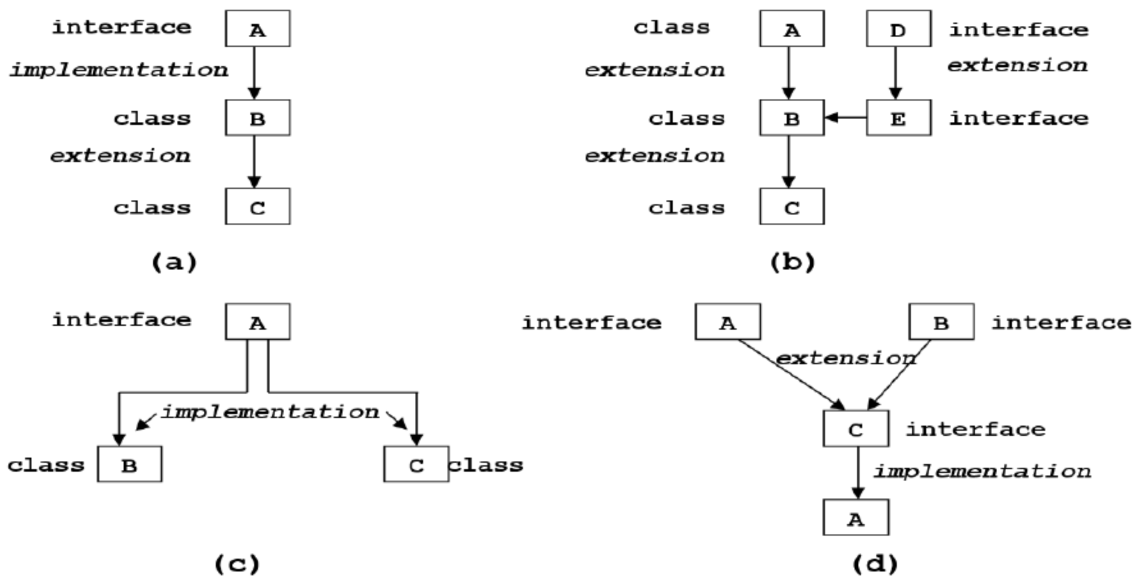
Ex:1

```
class classname implements interfacename
{
  body of the class
 }
```

Ex:2

```
Class classname extends superclass implements interface1,interface2
 {
 body of the class
 }
```

Various forms of interface implementation:



Example:

//**interface example**

//implementing interface

//implementing multiple interfaces

```
interface AreaData
{
 final static float pi=3.14F;
}
interface AreaMethod
{
 float compute(float x,float y);
}

class Rectangle implements AreaMethod  //implementing interface
{
 public float compute(float x,float y)
 {
  return(x*y);
  }
}

class Circle implements AreaData,AreaMethod //implementing multiple interfaces
{
public float compute(float x,float y)
 {
  return(pi*x*x);
  }
}
class Interface
{
 public static void main(String args[])
 {
  Rectangle r1=new Rectangle();
  Circle c1=new Circle();
  System.out.println("Area of rectangle....:"+r1.compute(10,20));
  System.out.println("Area of circle..:"+c1.compute(10,0));
  }
 }
```

Nested Interfaces: We can declare one interface inside another.

Example:

```
interface Show
{
  void disp();
```

```
 interface Message
 {
  void msg();
 }
}
class NestedInterface implements Show.Message
{
 public void disp()
 {
   System.out.println("Hello  interface");
 }
 public void msg()
 {
   System.out.println("Hello nested interface");
 }
}
class NestedInterface1
{
  public static void main(String args[])
  {
      NestedInterface message=new NestedInterface();
       message.msg();
       message.disp();
   }
}
```

<u>Inheritance of Interfaces</u>

- Like classes interfaces can be extended(inherited).
- An interface can be subinterfaced from other interfaces.

<u>Ex:1</u>

```
interface name2 extends name1
 {
    Body of name2
   }
```

<u>Ex:2</u>

We can put all constants in one interface and all the methods in another interface.

```
interface ItemConstants
 {
  int code=101;
  String name=”Fan”;
 }
interface Item extends ItemConstants
{
 void display();
 }
```

- We can also combine several interfaces into single interface

  Ex:

```
 interface ItemConstants
 {
  int code=101;
  String name=”Fan”;
 }

interface ItemMethods
{
 void display();
 }

interface Item extends ItemConstans,ItemMethods
{
 --------
 --------
 }
```

Notes:

- Subinterfaces cannot define the methods declared in the superinterfaces(they are also interfaces only).
- Interfaces are implemented by classes to define the methods.
- Interfaces cannot extends classes, this will violate rule that an interface can have only abstract methods.
- An class can extend an abstract class, because the class can have abstract and concrete methods also.

Default Methods in Interfaces

Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

Example:

```
interface Sayable
{
        // Default method
         default void say()
        {
              System.out.println("Hello, this is default method");
         }
           // Abstract method
           void sayMore(String msg);
}
public class DefaultMethods implements Sayable
{
           public void sayMore(String msg)
           {       // implementing abstract method
              System.out.println(msg);
           }
           public static void main(String[] args)
          {
```

```
        DefaultMethods dm = new DefaultMethods();
        dm.say();   // calling default method
        dm.sayMore("Work is worship");  // calling abstract method
    }
 }
```

Static Methods in Interface

- Static Methods in Interface are those methods, which are defined in the interface with the keyword static. Unlike other methods in Interface, these static methods contain the complete definition of the function and since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.
- Similar to Default Method in Interface, the static method in an interface can be defined in the interface, but cannot be overridden in Implementation Classes. To use a static method, Interface name should be instantiated with it, as it is a part of the Interface only.

Example:

```
// Java program to demonstrate
// static method in Interface.

interface NewInterface
{

   // static method
   static void hello()
   {
      System.out.println("Hello, New Static Method Here");
   }

   // Public and abstract method of Interface
   void overrideMethod(String str);
}

// Implementation Class
public class InterfaceDemo implements NewInterface
{
   public static void main(String[] args)
   {
      InterfaceDemo interfaceDemo = new InterfaceDemo();

      // Calling the static method of interface
      NewInterface.hello();

      // Calling the abstract method of interface
      interfaceDemo.overrideMethod("Hello, Override Method here");
   }

   // Implementing interface method
```

```
  @Override    //annotation
  public void overrideMethod(String str)
  {
     System.out.println(str);
  }
}
```

Functional Interfaces

- An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method.
- Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

Example:

```
@FunctionalInterface
interface sayable
{
   void say(String msg);
}
public class FunctionalInterfaceExample implements sayable
{
   public void say(String msg){
      System.out.println(msg);
   }
   public static void main(String[] args) {
      FunctionalInterfaceExample fie = new FunctionalInterfaceExample();
      fie.say("Hello there");
   }
}
```

Annotations

- Java Annotation is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
- Annotations in Java are used to provide additional information
- Built-In Java Annotations used in Java code
  - @Override
  - @SuppressWarnings
  - @Deprecated

Built-In Java Annotations used in other annotations

- @Target
- @Retention
- @Inherited
- @Documented

Write a same example program that is written in static methods in interface concept.