UNIT-3

I. Arrays:

1. Introduction
2. Declaration and Initialization of Arrays
3. Storage of Array in Computer Memory
4. Accessing Elements of Arrays
5. Operations on Array Elements
6. Assigning Array to Another Array
7. Dynamic Change of Array Size
8. Sorting of Arrays
9. Search for Values in Arrays
10. Class Arrays
11. Two-dimensional Arrays
12. Arrays of Varying Lengths
13. Three-dimensional Arrays
14. Arrays as Vectors.

II. Inheritance:

1. Introduction
2. Process of Inheritance
3. Types of Inheritances
4. Universal Super Class- Object Class
5. Inhibiting Inheritance of Class Using Final
6. Access Control and Inheritance
7. Multilevel Inheritance
8. Application of Keyword Super
9. Constructor Method and Inheritance
10. Method Overriding
11. Dynamic Method Dispatch
12. Abstract Classes
13. Interfaces and Inheritance.

III. Interfaces:

1. Introduction
2. Declaration of Interface
3. Implementation of Interface
4. Multiple Interfaces
5. Nested Interfaces
6. Inheritance of Interfaces
7. Default Methods in Interfaces
8. Static Methods in Interface
9. Functional Interfaces
10. Annotations.

I. Arrays:

1. Introduction

An array is a structure consisting of a group of elements of the same type. When a large number of data values of the same *type* are to be processed, it can be done efficiently by declaring an array of the data *type*.

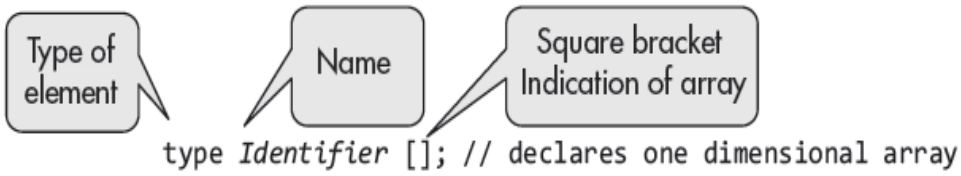
The complete data gets represented by a **single object with a single name** in the computer memory. **An array is a sequence of objects of the same data type**. The type of data that the array holds becomes the type of the array, which is also called *base type* of the array.

If the array elements have values in whole numbers, that is, of type int, the type of array is also int. If it is a sequence of characters, the type of array is char;

If it is an array of floating point numbers of type float, the type of array is also float.
An array can hold objects of a class but cannot be a mixture of different data types.

Syntax:

```
        datatype arrayName[];
or
        type identifier[];
```



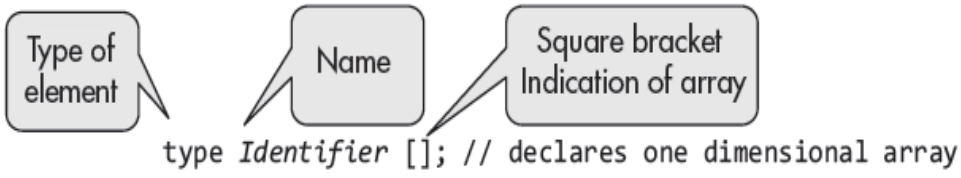
Examples:

```
int numbers []; // an array of whole numbers
char name []; // A name is an array of characters
float priceList[]; // An array of floating point numbers.
```

2. Declaration and Initialization of Arrays

Declaration of Array:

```
        datatype arrayName[];
or
        type identifier[];
```



Examples:

```
int numbers []; // an array of whole numbers
char name []; // A name is an array of characters
fl oat priceList []; // An array of floating point numbers.
```

Initialization of Arrays:

An array may be initialized by mentioning the values in braces and separated by commas.
For example, the array pencils may be initialized as below:

```
int pencils [] = {4, 6, 8, 3};
```

3. Storage of Array in Computer Memory

- The operator *new* allocates memory for storing the array elements.
- For example, with the following declaration

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

  here 4 elements will be created and assigned to the array "**numbers"**
- The declaration and initialization may as well be combined as:

```
int numbers [] = {20,10,30,50};
```

- A **two-dimensional** array may be declared and initialized as,

```
int [][] array2d = new int [][] {{1, 2, 3}, {4, 5, 6}};
```

  or as

```
int [][] array2D = {{1, 2, 3}, {4, 5, 6}};
```

4. Accessing Elements of Arrays

- The individual member of an array may be accessed by its index value.
- The index value represents the place of element in the array.
- The first element space is represented by numbers [0], and index value is 0.

  Note that the value of an array element is different from its index value.

  Example:

```
int numbers [] = {20,10,30,50};
```

  Here
  number[0] is the first element
  number[1] is the second element
  number[2] is the third element and so on…

  the value at number[0] is 20,
  value at number[1] is 10,
  value at number[2] is 30,
  value at number[3] is 20,
  value at number[4] is 50.

Determination of Array Size

- The size or length of an array may be determined using the following code:

```
int arraySize = array_identifier.length;
```

- The size of array numbers is determined as:

```
int size = numbers.length;
```

- The elements of a large array may be accessed using a *for* loop.

For example, the elements of array numbers may be accessed as

```
for (int i = 0; i<size; i++)
     System.out.println(x);
```

Example-1:

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

   int numbers[] = new int[4];

   numbers[0] = 10;
   numbers[1] = 20;
   numbers[2] = 30;
   numbers[3] = 40;

   for(int i =0 ;i<numbers.length; i++)
           System.out.println(numbers[i]);

   }

}
```

Output:

```
C:\>javac NumArray.java

C:\>java NumArray
10
20
30
40
```

Example-2: One Dimensional Array - NumArray2.java

```
class NumArray2
{
   public static void main(String[] args)
   {
    int numbers [] = {20,10,30,50};

    for(int i=0 ; i<numbers.length; i++ )
           System.out.println(numbers[i]);
   }
}
```

Output:

```
C:\>javac NumArray2.java

C:\>java NumArray2
10
20
30
40
```

Example-3: One Dimensional Array - NumArray3.java

```
class NumArray3
{
      public static void main(String[] args)
      {

      int numbers[] = new int[]{100,200,300,400} ;

      for(int i =0 ;i<numbers.length; i++)
            System.out.println(numbers[i]);

      }
}
```

Output:

```
C:\>javac NumArray3.java

C:\>java NumArray3
100
200
300
400
```

Example-4: One Dimensional Array - StringArray.java

```
class StringArray
{
      public static void main(String[] args)
      {
      String names[] = {"Red", "Blue", "Green", "Black", "White"} ;

      for(String i :names)
            System.out.println(i);

      }
}
```

Output:

```
C:\>javac StringArray.java

C:\>java StringArray
Red
Blue
Green
Black
White
```

Use of *for–each* Loop

- the *for–each* loop may be used to access each element of the array.

```
for (int x: numbers)
   System.out.println(x);
```

- For a two-dimensional array the nested *for–each* loops are used.

Example : Two Dimensional Array – TwoDimArray.java

```
class TwoDimArray
{
    public static void main(String[] args)
    {
   int pArray[][]= {{1,2,3},{4,5,7}};
   for(int[] y : pArray)
        {
          for(int x : y)
             System.out.print( x + " ");
          System.out.println();
        }
    }
}
```

Output:

```
C:\>javac TwoDimArray.java

C:\>java TwoDimArray
1 2 3
4 5 7
```

5. Operations on Array Elements

i. Arithmetic Operations on Arrays:

Arithmetic operations can be applied on Array elements.

Example:

```
int[] array1 = new int []{1,2,3,4,5};
array1[0] = array1[0] + 10;
```

Here the value of the first element is added 10. Now its value is changed from 1 to 11.

ii. Arrays as Parameters of Methods

Arrays can be passed to methods just like variables.

Example:

```
display(array1);  // Method calling
  . . . . ..
void display(int[] array)  //Method definition
{
     for (int x : array)
         System.out.println(x + " ");
}
```

Example: ArrayOperations.java

```
class ArrayOperations
{
  public static void main(String args[])
  {
   int[] array1 = new int []{1,2,3,4,5};

   System.out.println("Before Adding - Array elements are :");
   display(array1); //Passing an array to display() method
   System.out.println();

   // Operations on Arrays
   for(int i =0; i< array1.length; i++)
     array1[i] = array1[i] + 10;  // Adding 10 to each element

    System.out.println("After Adding - Array elements are :");
    display(array1);
    }

    static void display(int[] array)  //Method definition
    {
      for (int x : array)
           System.out.print(x + "   ");
    }
   }
```

Output:

```
C:\>javac ArrayOperations.java

C:\>java ArrayOperations
Before Adding - Array elements are :
1   2   3   4   5

After Adding - Array elements are :
11   12   13   14   15
```

6. Assigning Array to Another Array

- In Java, an array may be assigned to another array of same data type.
- In this process, the second array identifier is the reference to the first array.
- The second array is not a new array, instead only a second reference is created.
- This is illustrated in this program, array1 is assigned to array2. Then, array1 is modified array2 also gets modified, which shows is not an independent array.

```
class ArrayAssignment
{
     public static void main(String args[])
     {
      int[] array1 = new int []{1,2,3,4,5};
      int[] array2 = new int[array1.length];


      System.out.println("Array-1 elements are :");
        display(array1);
        System.out.println();

      array2 = array1; // Array Assignment

        System.out.println("Array-2 elements are :");
        display(array2);  // Method calling
        System.out.println();

      //Modification of array2 elements
        for(int i=0;i<array2.length;i++)
          array2[i]+=100;  //adding 100 to each element of array2

      System.out.println("After Modification of Array2 \n Array-1
elements are :");
        display(array1);
        System.out.println();
       }

     static void display(int[] array)  //Method definition
     {
       for (int x : array)
           System.out.print(x + "   ");
     }
}
```

```
C:\ >javac  ArrayAssignment.java

C:\ >java ArrayAssignment
Array-1 elements are :
1  2  3  4  5
Array-2 elements are :
1  2  3  4  5
After Modification of Array2
 Array-1 elements are :
101  102  103  104  105
```

7. Dynamic Change of Array Size

Java allows us to change the array size dynamically during the execution of the program. In this process the array destroyed along with the values of elements. In the following program, the array contains 5 elements. It is again defined with 10 elements with the same array name.

Example:

```
class DyanamicArraySize
{
     public static void main(String args[])
     {
      int[] array1 = new int []{1,2,3,4,5};

      System.out.println("Before Changing Array Size: array1 = ");
      display(array1);

      //Changing array size
      array1 = new int[10];

      System.out.println("\nAfter Changing Array Size: array1 = ");
      display(array1);

      //adding values to the array elements
      for(int i=0;i<10;i++)
        array1[i] = 5*(i+1);

      System.out.println("\nAfter Modification : array1 = ");
      display(array1);
     }
     static void display(int[] array)  //Method definition
     {
       for (int x : array)
           System.out.print(x + "   ");
     }
}
```

Output:

```
C:\>javac DynamicArraySize.java

C:\>java DyanamicArraySize
Before Changing Array Size: array1 =
1   2   3   4   5
After Changing Array Size: array1 =
0   0   0   0   0   0   0   0   0   0
After Modification : array1 =
5   10   15   20   25   30   35   40   45   50
```

8. Sorting of Arrays

Sorting of arrays is often needed in many applications of arrays. For example, in the preparation of "examination results" , "order of grades acquired by students" or "Student names in alphabetical order of dictionary style". The arrays may be sorted in ascending or descending order. Several methods are used for sorting the arrays that include the following:

1. Bubble sort

2. Selection sort

3. Sorting by insertion method

1. Bubble Sort

This method of sorting is the simplest to understand but the most inefficient one; however, it can be use fully employed for short arrays.

In the process, if the sorting is done in ascending order, the first element is compared with the second element. If the value of the second is smaller than the first, the elements are inter changed, that is, the second is made first and first is made second.

However, if the second is higher than the first, then no action is taken. The second element is then compared with the third element and the aforementioned procedure is repeated. The third is then compared with the fourth.

The process is repeated till the last . This process places the largest value as the last element. The process is again element repeated for the next largest value from the remaining elements until the last element of the array is reached. Thus, the process is repeated (n-1) times to completely sort the array.

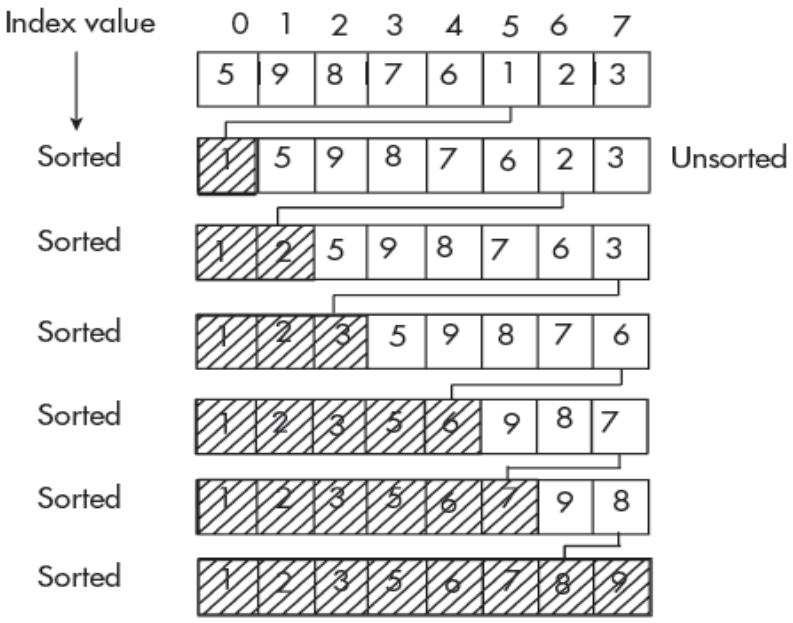


Fig. 7.6 Bubble sort method

Example:

```
class BubbleSort
{
     public static void main(String args[])
     {
      int[] array1 = new int []{5,8,9,2,4,1,7,6};

      System.out.println("Before Sorting : array1 = ");
      display(array1);

      // Buble Sorting
      int t;
      for(int i=0; i<array1.length; i++)
      {
         for(int j=array1.length-1; j>0 ; j--)
          {
            if(array1[j-1]>array1[j])
            {
            t=array1[j];
            array1[j] = array1[j-1];
            array1[j-1] =t;
            }
          }
      }
      System.out.println("\nAfter Sorting : array1 = ");
      display(array1);
```

```
    }
    public static void display(int[] array)  //Method definition
    {
      for (int x : array)
          System.out.print(x + "   ");
    }
}
```

Output:

```
C:\ >javac BubbleSort.java

C:\ >java BubbleSort
Before Sorting : array1 =
5   8   9   2   4   1   7   6
After Sorting : array1 =
  1    2   4   5   6   7   8   9
```

2. Selection Sort

- Let us take an array with elements 9, 7, 4, 3, 6, and 8.
- If the array is being sorted in ascending order, pick the element with the lowest value, that is 3, and place at the first place as shown in the second line of Figure.
- From the remaining elements that are 9, 7, 4, 6, 8, again pick the lowest value, that is, 4 and place it next to 3, as shown in the third line of the figure.
- Then, from the remaining elements with values 9, 7, 6, 8, again pick the lowest value, that is, 6 and place it next to 4.
- From the remaining three values, 9, 7, 8, again pick the lowest value, that is, 7 and place it next to 6 on its right side.
- Then, from the remaining two values 9 and 8, pick the lowest, that is, 8 and place it next to 7.
- The highest value goes to the last place. The array is sorted with the lowest value at the first place and the largest value at the end.
- The process is better than the bubble sort but still not the most efficient.

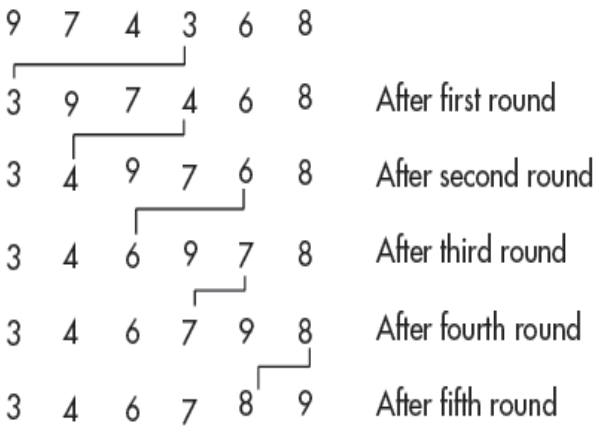


Fig. 7.7 Selection sort

Example:

```
class SelectionSort
{
     static int min=0;
     public static void main(String args[])
     {
      int[] array1 = new int []{9,7,4,3,6,7};

      System.out.println("Before Sorting : array1 = ");
      display(array1);

      // Selection Sort

      int minIndex=0;

      for(int i=0; i<array1.length; i++)
      {
         min=array1[i];
         for(int j=i+1; j<array1.length ; j++)
          {
              if(min>array1[j])
           {
              min=array1[j];
                 minIndex=j;
           }
          }
         for(int j=minIndex;j>i;j--)
            array1[j]=array1[j-1];
           array1[i]=min;
```

```
      }
     System.out.println("\nAfter Sorting : array1 = ");
     display(array1);
     }

     public static void display(int[] array)  //Method definition
     {
       for (int x : array)
           System.out.print(x + "   ");
     }
}
```

Output:

```
C:\ >javac SelectionSort.java

C:\ >java SelectionSort
Before Sorting : array1 =
9   7   4   3   6   7
After Sorting : array1 =
3   4   6   7   7   9
```

3. Insertion Sort

- Sorting algorithm builds a final sorted array one item at a time.
- In this method, the value at any index is compared to all the prior elements.
- The input data is inserted into the correct position in the sorted list and the process is repeated until no input element remains.

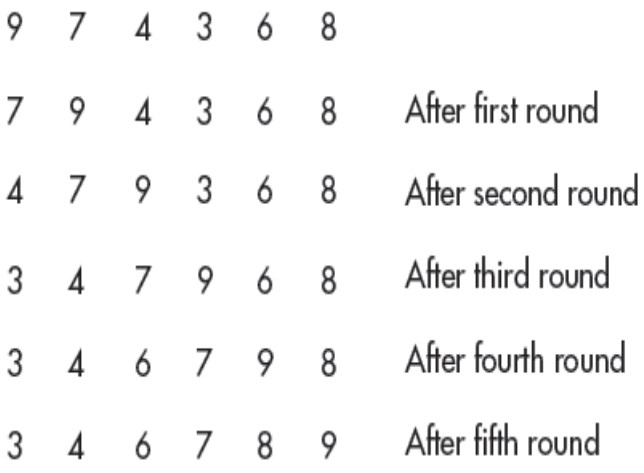


Fig. 7.8 Insertion sort

Example: InsertionSort.java

```
class InsertionSort
{
     static int min=0;
     public static void main(String args[])
     {
   int[] array1 = new int []{9,7,4,3,6,7};

   System.out.println("Before Sorting : array1 = ");
   display(array1);

   // Insertion Sort
   int len = array1.length;
   int key =0;

   int i=0;
   for(int j=1; j<len;j++)
   {
       key= array1[j];
       i=j-1;
       while(i>=0 && array1[i]>key)
       {
         array1[i+1] = array1[i];
         i=i-1;
         array1[i+1]=key;
       }

           /* System.out.println("\nAfter "+(i+1)+" Iteration
array1=");
       display(array1);*/
    }

    System.out.println("\nAfter Sorting : array1 = ");
    display(array1);
     }

     public static void display(int[] array)  //Method definition
     {
       for (int x : array)
           System.out.print(x + "   ");
     }
}
```

Ouput:

```
C:\ >javac InsertionSort.java

C:\ >java InsertionSort
Before Sorting : array1 =
9   7   4   3   6   7
After Sorting : array1 =
3   4   6   7   7   9
```

9. Search for Values in Arrays

Searching an array for a value is often needed. Let us consider the example of searching for your name among the reserved seats in a rail reservation chart, etc. Two methods are used in searching, They are :

1. Linear search
2. Binary search for sorted arrays.

1. Linear search

The method may be applied to any array.
The key value is compared to the value of the elements of the array successively.
If a match is found, it is noted, and the program ends there.
Otherwise, the complete array is searched, and if no match is found, it is reported that the value is not there in the array.

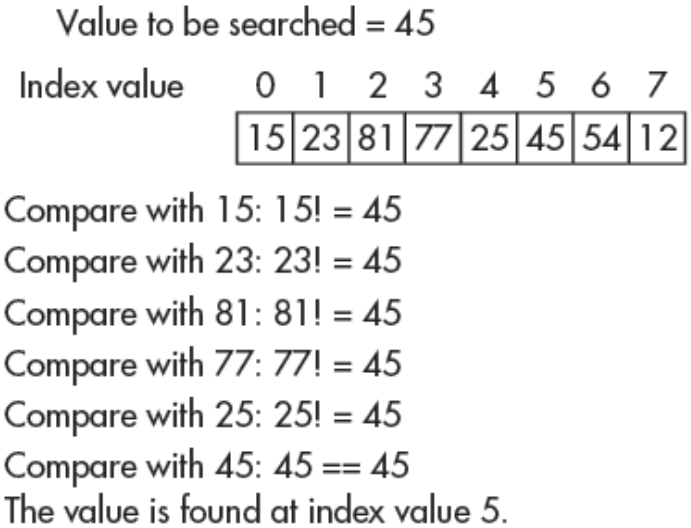


Fig. 7.9 Illustration of linear search

Example:

```
import java.util.Scanner;
class LinearSearch
{
  public static void main(String args[])
  {
   boolean b =false;
   int[] array = {67,78,85,44,25,65,36};
   Scanner in = new Scanner(System.in);
   System.out.println("Enter the number which you want to
search");
   int key = in.nextInt();

   for(int i=0; i<array.length; i++)
   {
            if(array[i] == key)
      {
         System.out.println("Your number is at index = " + i);
         b=true;
            }
   }
         if(b!=true)
    System.out.println("Your number is not in the array");
   }
}
```

Output:

```
C:\>javac LinearSearch.java

C:\>java LinearSearch
Enter the number which you want to search
44
Your number is at index = 3

C:\>java LinearSearch
Enter the number which you want to search
65
Your number is at index = 5

C:\>java LinearSearch
Enter the number which you want to search
888
Your number is not in the array
```

2. **Binary search for sorted arrays.**

- It is a very efficient method of search but it is applicable only to the sorted arrays.
- The beginning, end, and the midpoint of the array are defined first.
- The key value is compared with the value at midpoint.
- If it does not match, then it is checked in which half of the array the key value lies by checking whether the key value is more or less than the value at midpoint.
- The array is truncated to the half in which the key value lies.
- This process is repeated on that half, that is, it is again divided into two halves where the value is compared with the midpoint;
- if match is not found, it is determined in which half of the truncated array the value lies.
- The process is very useful for searching large sorted arrays

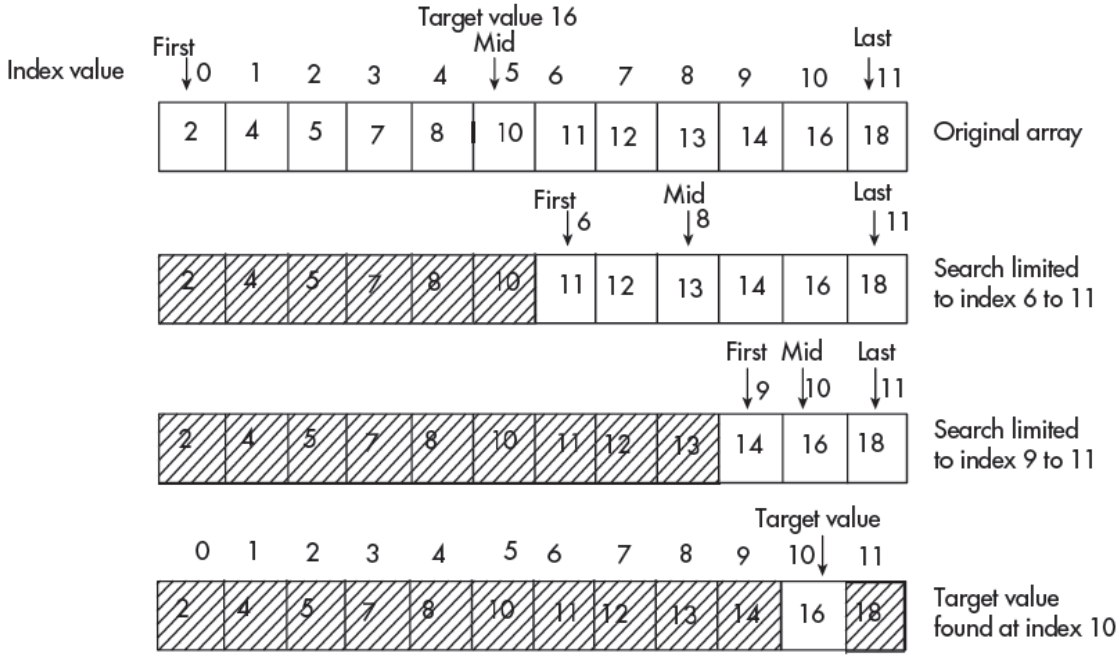


Fig. 7.10 Illustration of binary search

Example:

```
import java.util.Scanner;
class BinarySearch
{
  public static void main(String args[])
  {
      boolean b =false;
      int[] array = {15, 20, 43, 45, 76, 80, 86, 88, 90, 94, 96,
98};
      int length= array.length;
      Scanner in = new Scanner(System.in);
      System.out.println("Enter the number which you want to
search");
      int key = in.nextInt();
```

```
      int beginning,end,mid;
      beginning =0;
      end = length-1;
      while( beginning <=end)
      {
        if(beginning == end && array[end]!=key)
        {
          System.out.println("Your number is not in the array");
          break;
        }
        mid = (beginning +end )/2;
        if(array[mid]==key)
        {
          System.out.println("Your number is in the array at index =
" + mid);
          break;
        }
        else if(array[mid]<key)
          beginning = mid+1;
        else
          end = mid-1;
      }
   }
}
```

Output:

```
C:\>javac BinarySearch.java

C:\>java BinarySearch
Enter the number which you want to search
45
Your number is in the array at index = 3

C:\>java BinarySearch
Enter the number which you want to search
96
Your number is in the array at index = 10

C:\>java BinarySearch
Enter the number which you want to search
783
Your number is not in the array
```

10. Class Arrays

- The package java.util defines the class Arrays with static methods. –
- for general processes that are carried out on arrays such as
  - sorting an array for full length of the array or for part of an array,
  - binary search of an array for the full array or part of an array,
- for comparing two arrays if they are equal or not.
- for filling a part or the full array with elements having a specified value.
- for copying an array to another array.
- The sort method of Arrays class is based on quicksort technique.
- The methods are applicable to all primitive types as well as to class objects.

The methods of class Arrays are as follows:

Sort

- The class defines several overloaded methods for sorting arrays of different types.
- As per Java SE7, the sort method for int array is

1. public static void sort (int[] array)

Example: PredefinedMethodSort.java

```
import java.util.Arrays;
class PredefinedMethodSort
{
     public static void main(String args[])
     {
      int[] array1 = new int []{41,4,31,14,5};

      System.out.println("Array-1 elements are :");
         display(array1);
         System.out.println();

      //Predefined Method Sort of Arrays class
      Arrays.sort(array1);

      System.out.println("After Sort, Array-1 elements are :");
         display(array1);
         System.out.println();
       }

     static void display(int[] array)  //Method definition
     {
       for (int x : array)
           System.out.print(x + "   ");
     }
```

```
}

Output:

C:\>javac PredefinedMethodSort.java

C:\ >java PredefinedMethodSort
Array-1 elements are :
41   4   31   14   5
After Sort, Array-1 elements are :
4   5   14   31   41
```

Searching

There are two versions of overloaded binary Search method that are defined in class Arrays.

1. public static int binarySearch(int [] array, int key)

Equals

public static boolean equals (int [] a, int [] b)

The output is a Boolean value—it returns true, if the elements and their order in the two arrays are same; otherwise, it returns false.

Fill

The two versions of method fill defined in class Arrays are as follows.

1. public static void fill (byte [] array, byte value)
   The method fills the entire array with a specified value.
2. public static void fill(byte [] array, int startIndex, int endIndex, byte value)

The method fills the specified subset of an array with the specified value

CopyOf

This method was added in Java SE 6.

1. public static byte [] copyOf(byte [] original, int length)
   The method copies the array into a new array of specified length
2. copyOfRange
   public static char [] copyOfRange(char [] original, int fromIndex, int toIndex)

asList

public static <T> List<T> asList(T… array)
The method returns a fixed-sized list backed by the array.

toString

The method header is given as public static String toString (int [] array)
The method returns a string representation of the array elements.

11. Two-dimensional Arrays

- An array may hold other arrays as its elements.
- If the elements of an array are one-dimensional arrays, the array becomes a two-dimensional array.
- A two-dimensional array is treated as an array of arrays, and each of these arrays may have a different number of elements.
- E.g. Matrices are two-dimensional arrays.
- List of telephone numbers is another such example.
- A two-dimensional array may be defined as:
- int telNumber [][] = new int [5][10];
- The array will contain 5 numbers each having 10 digits.
- A two-dimensional array may as well be defined and initialized as
- int arrayB [ ][ ] = {{11, 12, 13 }, {7, 6, 4}};
- The two-dimensional array may as well be declared as
- int arrayC [][] = new int[2][3]{{1,2,3}. {4.5.6},{7,8,9}}

Example: TwoDimArray.java

```
class TwoDimArray
{
    public static void main(String[] args)
    {
      int num2D[][]= {{1,2,3},{4,5,6},{7,8,9}};

      for(int[] y : num2D)
      {
        for(int x : y)
           System.out.print( x + " ");
        System.out.println();
      }
    }
}
```

Output:

```
C:\ >javac TwoDimArray.java

C:\ >java TwoDimArray
1 2 3
4 5 6
7 8 9
```

12.Arrays of Varying Lengths

- A two-dimensional array is treated as an array whose elements are one-dimensional arrays, which may have different sizes.
- A two-dimensional array may be declared as
  `int a2D [][] = new int [3 ][];`
- The arrays may as well be declared as
  `int array [][] = {{5, 7, 8 },{10, 11 }, {4, 3, 2, 7,5 }};`

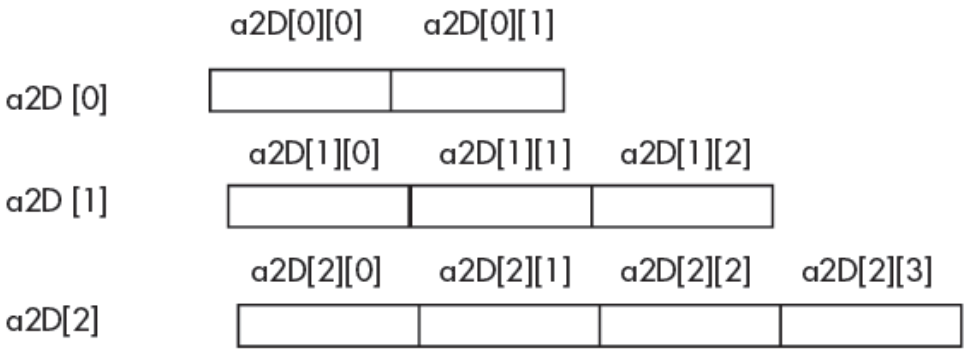


Fig. 7.11 Two-dimensional arrays of varying lengths

Example: TwoDimArray2.java

```
class TwoDimArray2
{
    public static void main(String[] args)
    {
     int num2D[][]= {{5,7,8},{10,11},{4,3,2,7,5}};

     for(int[] y : num2D)
     {
       for(int x : y)
          System.out.print( x + " ");
       System.out.println();
     }
    }
}
C:\>javac TwoDimArray2.java

C:\>java TwoDimArray2
5 7 8
10 11
4 3 2 7 5
```

13.Three-dimensional Arrays

- When an array holds two-dimensional arrays as its elements, the array is a three-dimensional array.
- A practical example includes an array of matrices.
- Each basic element of such an array needs three index values for its reference.
- A three-dimensional array may be declared as

```
int tDArray [][][]; //Declaration
double d3Array [][][]; //Declaration
```

Example: ThreeDimArray.java

```
class ThreeDimArray
{
    public static void main(String[] args)
    {
     int num3D[][][]= {  { {1,2,3},   {4,5,6},   {7,8,9}    },
                 { {11,12,13},{14,15,16},{17,18,19} }
                };

       for(int[][] z: num3D)
       {
      for(int[]y : z)
      {
       for(int x : y)
           System.out.print( x + " ");
       System.out.println();
      }
     System.out.println();
      }
    }
}
```

Output:

```
C:\1. JAVA\UNIT-3.1>javac ThreeDimArray.java
C:\1. JAVA\UNIT-3.1>java ThreeDimArray
1 2 3
4 5 6

7 8 9


11 12 13

14 15 16
```

17 18 19

14. Arrays as Vectors.

- Similar to Arrays, vectors are another kind of data structure that is used for storing information.
- Using vector, we can implement a dynamic array.
- The following are the vector constructors:
- Vector() creates a default vector having an initial size of 10.
- Vector(int size) creates a vector whose initial capacity is specified by size.
- Example
  `Vector vec = new Vector(5);` // declaring with initial size of 5
- Vector(int size, int incr) creates a vector with initial capacity specified by size and increment is specified by incr.
- The increment is the number of elements added in each reallocation cycle.

Advantages of Vectors.

Vectors have a number of advantages over arrays.

i. Vectors are dynamically allocated, and therefore, they provide efficient memory allocation.
ii. Size of the vector can be changed as and when required.
iii. They can store dynamic list of objects.
iv. The objects can be added or deleted from the list as per the requirement.

Table 7.1 Some of the important methods of Vector class

Method	*Description*
void add(int index, Object element)	Inserts the specified element at the specified position in the given vector
void addElement(Object obj)	Adds the specified component to the end of the given vector and increases its size by one
void clear()	Removes all the elements from the given vector
int capacity()	Returns the current capacity of the given vector
void copyInto(Object[] anArray)	Copies the components of the given vector into the specified array
Object firstElement()	Returns the first component (i.e., item at index 0) of the given vector
Object lastElement()	Returns the last component of the given vector
Enumeration elements()	Returns the enumeration of the components of the given vector
Object get(int index)	Returns the element at the specified position in the given vector
Object remove(int index)	Removes the element at the specified position in the given vector
int size()	Returns the number of elements currently in the vector

Example: VetorArray.java

```
import java.util.*;
class VectorArray
{
    public static void main(String[] args)
    {
     Scanner in = new Scanner(System.in);
     System.out.println("Enter the Vector capacity : ");
     int n = in.nextInt();

     //Declaring Vector with capacity n
     Vector<Integer> vec = new Vector<Integer>(n);

System.out.println("Intial size of Vector" + vec.size());
System.out.println("Intial capacity of Vector" + vec.capacity());
     //Adding the elements to the vector
     for(int i=0; i<n;i++)
        vec.add(i+10);

System.out.println("Current size of Vector" + vec.size());
System.out.println("Intial capacity of Vector" + vec.capacity());

     //Printing the vector elements
     System.out.println("Vector Elements are ");
     for (int i = 0; i < vec.size(); i++)
           System.out.print(vec.get(i) + " ");

     //removing element from Vector at index 3
     vec.remove(3);

     System.out.println("\n Vector Elements after removing element
at index 3");
     for (int i = 0; i < vec.size(); i++)
           System.out.print(vec.get(i) + " ");
    }
}
```

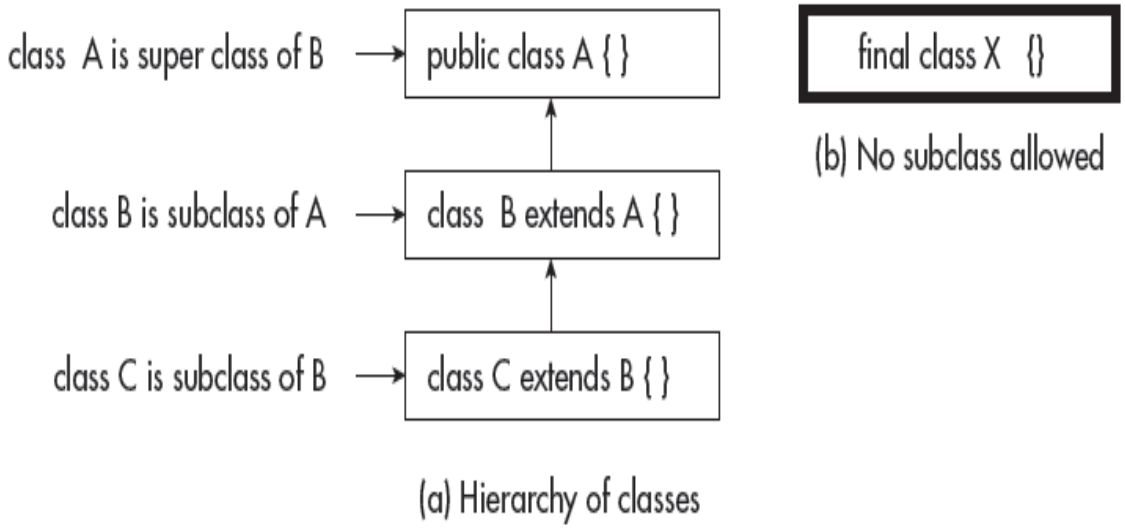
Output:

```
C:\>javac VectorArray.java

C:\>java VectorArray
Enter the Vector capacity :
7
Intial size of Vector0
Intial capacity of Vector7
Current size of Vector7
Intial capacity of Vector7
Vector Elements are
10 11 12 13 14 15 16
 Vector Elements after removing element at index 3
   10 1 12 14 15 16
```

II. Inheritance:

1. Introduction

- Inheritance is the backbone of object-oriented programming (OOP).
- It is the mechanism by which a class can acquire properties and methods of another class.
- Using inheritance, an already tested and debugged class program can be reused for some other application.
- **Super class** This is the existing class from which another class, that is, the subclass is generally derived.
- In Java, several derived classes can have the same super class.
- **Subclass** A class that is derived from another class is called subclass.
- In Java, a subclass can have only one super class.



Benefits of Inheritance

- It allows the reuse of already developed and debugged class program without any modification.
- It allows a number of subclasses to fulfil the needs of several subgroups.
- A large program may be divided into suitable classes and subclasses that may be developed by separate teams of programmers.

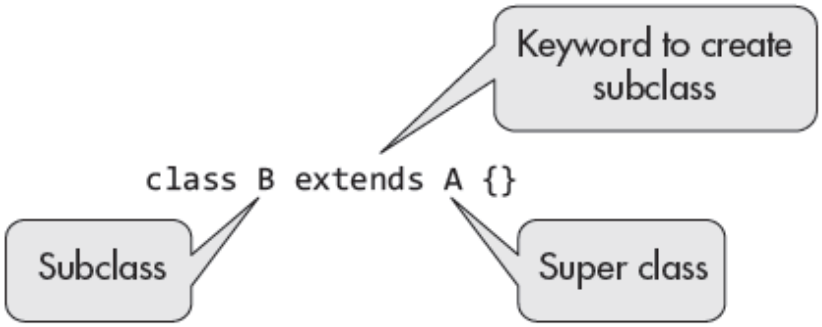
Disadvantages of Inheritance

1. The tight coupling between super and subclasses increases and it becomes **very difficult to use them independently**.
2. **Program processing time increases** as it takes more time for the control to jump through various levels of overloaded classes.
3. **When some new features are added** to super and derived classes as a part of maintenance, the changes affect both the classes.
4. **When some methods are deleted** in super class that is inherited by a subclass, the

methods of subclass will no longer override the super class method.

2. Process of Inheritance

- Inheritance means deriving some characteristics from something that is generic.
- In the context of Java, it implies deriving a new class from an existing old class, that is, the super class.
- A super class describes general characteristics of a class of objects.
- A subset of these objects may have characteristics different from others.
- There are two ways of dealing with this problem.
- Either, make a separate class for the subset to include all the characteristics or, to have another class that inherits the existing class, extend this class to include the special characteristics.



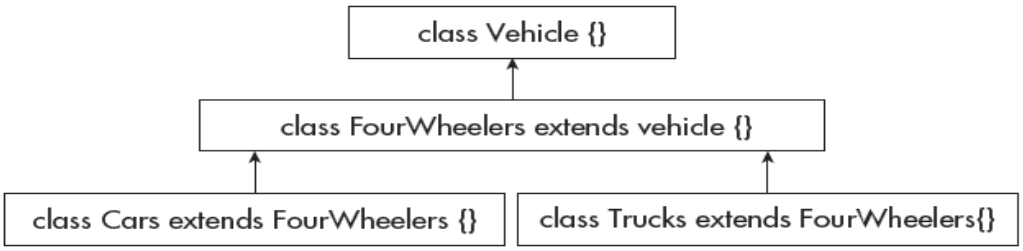


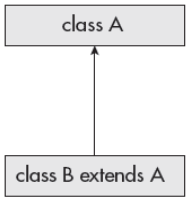
Fig. 8.4 Example for process of inheritance

3. Types of Inheritances

The following types of inheritances are supported by Java.

1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4. Multiple inheritance using interfaces

i. Single inheritance: It is the simple type of inheritance. In this, a class extends another one class only.



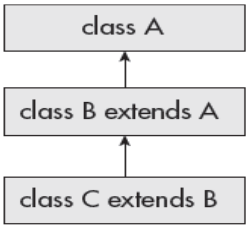
Example: SingleInheritance.java

```
class DemoA
{
      void displayA()
      {
            System.out.println("Super Class Method");
      }
}
class DemoB extends DemoA
{
      void displayB()
      {
            System.out.println("Sub Class Method");
      }
}
class SingleInheritance
{
     public static void main(String args[])
     {
      DemoA objA = new DemoA();
      objA.displayA();

      DemoB objB = new DemoB();
      objB.displayB();
       }
}
```

Output:

```
C:\>javac SingleInheritance.java
C:\>java SingleInheritance
Super Class Method
Sub Class Method
```

ii. Multilevel inheritance: In this type, a derived class inherits a parent or super class; The derived class also acts as the parent class to other class.

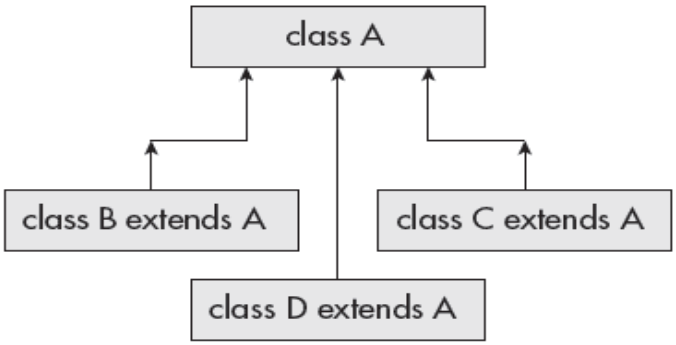


Example: Multilevel.java

```
class DemoA
{
      void displayA()
      {
            System.out.println("Class-A Method");
      }
}

class DemoB extends DemoA
{
      void displayB()
      {
            System.out.println("Class-B Method");
      }
}

class DemoC extends DemoB
{
      void displayC()
      {
            System.out.println("Class-C Method");
      }
}

class Multilevel
{
     public static void main(String args[])
     {
      //calling class-A method
      DemoA objA = new DemoA();
      objA.displayA();
```

```
      //calling class-B method
      DemoB objB = new DemoB();
      objB.displayB();

      //calling class-C method
      DemoC objC = new DemoC();
      objC.displayC();
       }
}
Output:
C:\>javac Multilevel.java

C:\>java Multilevel
Class-A Method
Class-B Method
Class-C Method
```

iii. **Hierarchical inheritance:** In this type, one class is inherited by many sub classes.



(c)

Example: Hierarchical.java

```
class DemoA
{
      void displayA()
      {
            System.out.println("Class-A Method");
      }
}

class DemoB extends DemoA
{
      void displayB()
      {
            System.out.println("Class-B Method");
```

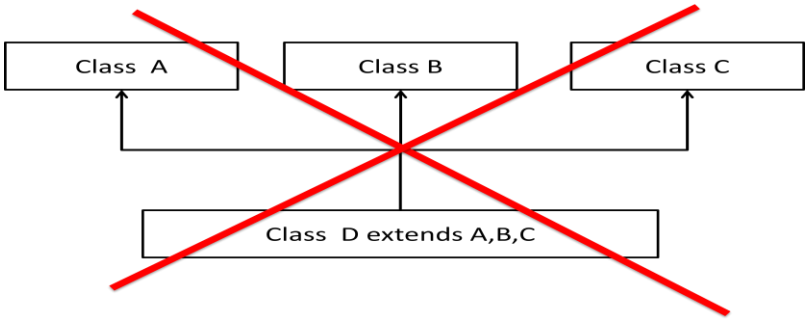
```
      }
}

class DemoC extends DemoA
{
      void displayC()
      {
            System.out.println("Class-C Method");
      }
}

class Heirarchical
{
     public static void main(String args[])
     {
      //calling class-A method
      DemoA objA = new DemoA();
      objA.displayA();

      //calling class-B method
      DemoB objB = new DemoB();
      objB.displayB();

      //calling class-C method
      DemoC objC = new DemoC();
      objC.displayC();
       }
}
```

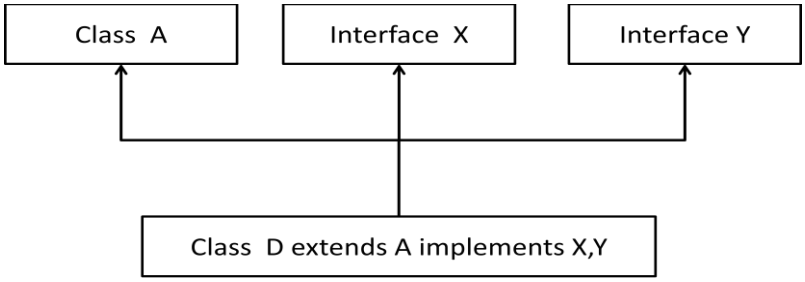
Output:

```
C:\ >javac Hierarchical.java

C:\ >java Hierarchical
Class-A Method
Class-B Method
Class-C Method
```

iv. **Multiple inheritance:** In this, a class is extending more than one class.

- Java does not support multiple inheritance.
- This implies that a class cannot extend more than one class.
- Suppose there is a method in class A. This method is overridden in class B and class C in their own way.
- Since class C extends both the classes A and B.
- So, if class C uses the same method, then there will be ambiguity as which method is called.





Example: Multiple.java

```
interface X
{
      int x=10;
}

interface Y
{
      int y=20;
}


class DemoA
{
      void displayA()
      {
            System.out.println("Class-A Method");
      }
}
```

```
class DemoB extends DemoA implements X,Y
{
      void displayB()
      {
            System.out.println("Class-B Method : x+y = " +
(x+y));
      }
}


class Multiple
{
     public static void main(String args[])
     {
      //calling class-A method
      DemoA objA = new DemoA();
      objA.displayA();

      //calling class-B method
      DemoB objB = new DemoB();
      objB.displayB();

       }
}
```

Output:

```
C:\ >javac Multilevel.java

C:\ >java Multilevel
Class-A Method
Class-B Method
Class-C Method
```

4. Universal Super Class : " Object " Class

- **Object** class is a special class and it is at the top of the class hierarchy tree.
- It is the parent class or super class of all in Java.
- Hence, it is called Universal super class.
- Object is at the root of the tree and every other class can be directly or indirectly derived from the Object class.

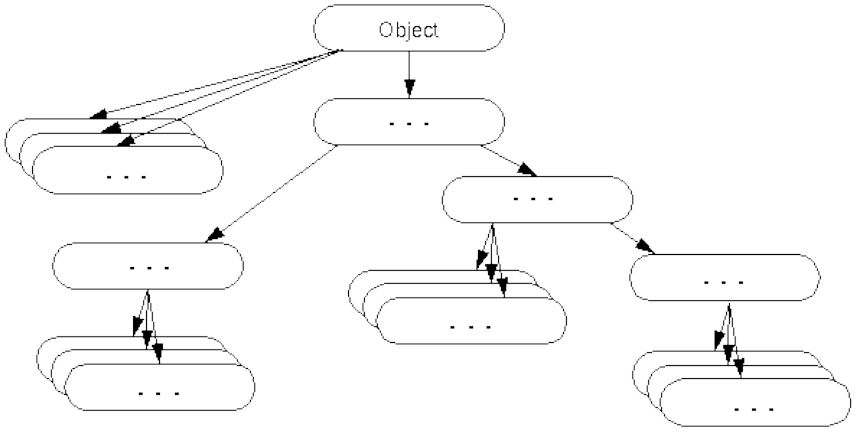


Table 8.2 Methods of object class

Method	*Description*
public String toString()	Returns the string representation of this object.
public int hashCode()	Returns the hashcode number for the given object.
public boolean equals(Object obj)	Compares the given object to this object.
Public final Class getClass()	Returns the Class class object of this object. The Class can be used to get the metadata of this class.

Example:ObjectEquals.java

```
class DemoA
{
      void displayA()
      {
            System.out.println("Class-A Method");
      }
}

class ObjectEquals
{
     public static void main(String args[])
     {
      DemoA obj1 = new DemoA();
      DemoA obj2 = new DemoA();
      boolean test;
```

```
      //Checking - if both object are equal
      test = obj1.equals(obj2);
      display(test);

      // Object assignment
      obj1=obj2;

      //Checking - if both object are equal after assigning the
objects
      test = obj1.equals(obj2);
      display(test);
      }

     public static void display(boolean test)
      {
      if (test)
        System.out.println("Both objects are same");
      else
        System.out.println("Both objects are different");
      }
}
```

Output:

```
C:\>javac ObjectEquals.java

C:\>java ObjectEquals
Both objects are different
Both objects are same
```

5. Inhibiting Inheritance of Class Using Final

- **A class declared as final cannot be inherited further**.
- Class variables or instance variables are declared as constant to make local variables.
- When a class is inherited by other classes, its methods can be overridden.
- In order to prevent the methods from being overridden, that method can be declared as final.

Example:FinalClass.java

```
final class A
{
      int a;
        A(int x) {a=x;}
      void display()
      {
        System.out.println("a = "+ a);
      }
}

class B extends A
{
      int b;
        B(int x,int y)
      {
          super(x);
          this.b=y;
      }
      void display()
      {
        System.out.println("b = "+ b);
      }
}

class FinalClass
{
      public static void main (String args[])
      {
            A objA= new A(10);
            B objB= new B(100,200);

            objA.display();
            objB.display();
      }
}
```

Output:

```
C:\>javac FinalClass.java
FinalClass.java:11: error: cannot inherit from final A
class B extends A
                ^
1 error
```

6. Access Control and Inheritance

- A derived class access to the members of a super class may be modified by access specifiers.
- There are three access specifiers, that is, public, protected, and private.
- The code for specifying access is Access-specifier type member_identifier;

Table 8.3 Access specifiers

Access specifiers	*Access*
No access specifier	Access permitted to any other class belonging to the same package
public	Access permitted to any class in any package
protected	Access permitted to any subclass in any package, also to any class in the same package
private	Access permitted only to members of the same class. No outside code has access to a private member of a class

Table 8.4 Access control

Class	*Access permitted (Yes/No)*			
	No specifier	*Public*	*Protected*	*Private*
Same class members	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	No
Any other class in same package	Yes	Yes	Yes	No
Subclass in different package	No	Yes	Yes	No
Any other class in different package	No	Yes	No	No

Example-1: DefaultAccess.java

```
class DemoA
{
   int a;
   void displayA()
   {
         System.out.println("Class-A Method : a = " + a);
   }
}

class DemoB extends DemoA
{
   int b;
   void displayB()
   {
System.out.println("Class-B Method : a = " + a + "  b = " + b );
   }
}


class DefaultAccess
{
     public static void main(String args[])
     {
   //calling class-A method
   DemoA objA = new DemoA();
   objA.a=100;  // accessing all classes in the same package
   objA.displayA();

   //calling class-B method
   DemoB objB = new DemoB();
   objB.a=200; // objA and objB are different objects. a is
assigned with 200
   objB.b=300; // accessing all classes in the same package
   objB.displayB();

      }
}
```

Output:

```
C:\>javac DefaultAccess.java

C:\>java DefaultAccess
Class-A Method : a = 100
Class-B Method : a = 200  b = 300
```

Example -2: PrivateAccess.java

```
class DemoA
{
      private int a;
      DemoA(int x)
      {
        a = x;
      }
      void displayA()
      {
            System.out.println("Class-A Method : a = " + a);
      }
}

class DemoB extends DemoA
{
      int b;
      DemoB(int p, int q)
      {
        super(p);
        b=q;
      }
      void displayB()
      {
            displayA();
            System.out.println("Class-B Method : b = " + b );
      }
}

class PrivateAccess
{
     public static void main(String args[])
     {
      //calling class-A method
      DemoA objA = new DemoA(150);
      //objA.a=100;  // Error, Can't access private variable
      objA.displayA();

      //calling class-B method
      DemoB objB = new DemoB(500,1000);
      // objB.a=200; // objA and objB are different objects. Error,
Can't access private variable
      //objB.b=300; // accessing all classes in the same package
      objB.displayB();

       }
}
```

Output:

```
C:\>javac PrivateAccess.java

C:\>java PrivateAccess
Class-A Method : a = 150
Class-A Method : a = 500
Class-B Method : b = 1000
```

Example: ProtectedAccess.java

```
class DemoA
{
      protected int a;
      DemoA(int t)
      {
        a = t;
      }
      void displayA()
      {
            System.out.println("Class-A Method : a = " + a);
      }
}

class DemoB extends DemoA
{
      int b;
      DemoB(int p, int q)
      {
        super(p);
        b=q;
      }
      void displayB()
      {
      System.out.println("Class-B Method : a= " + a + " b = " + b );
      }
}

class DemoC extends DemoB
{
      int c;
      DemoC(int x, int y, int z)
      {
        super(x,y);
        c=z;
      }
      void displayC()
      {
      System.out.println("Class-B Method : a = " + a + " b = " + b +
" c = " + c);
      }
```

```
}

class ProtectedAccess
{
     public static void main(String args[])
     {
      //calling class-A method
      DemoA objA = new DemoA(100);
      //objA.a=100;  // Error, Can't access protected variable
      objA.displayA();

      //calling class-B method
      DemoB objB = new DemoB(200,300);
      objB.displayB();

      //calling class-C method
      DemoC objC = new DemoC(250,500,750);
      objC.displayC();

       }
}
```

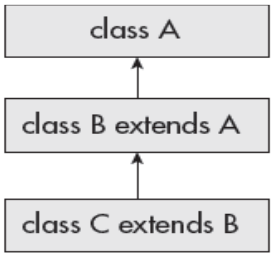
Output:

```
C:\>javac ProtectedAccess.java

C:\>java ProtectedAccess
Class-A Method : a = 100
Class-B Method : a= 200 b = 300
Class-B Method : a = 250 b = 500 c = 750
```

7. Multilevel Inheritance

In this type, a derived class inherits a parent or super class;

- The derived class also acts as the parent class to other class.



See Example : Multilevel.java

8. Application of Keyword Super

The keyword super is used for two purposes:

First, to distinguish between the variables having the same name in super class and subclass.

- When the member is called with an object of subclass, the subclass value will be presented and super class value will get hidden.
- For getting super class value, the keyword super is used.

Second, it is used in defining the constructor of subclass.

- Instead of repeating the assignment of variables of super class, we simply qualify the variable with super.

Example: SuperDemo.java

```
class A
{
      int a;
      A(int x)
      {
        a = x;
      }
      void displayA()
      {
            System.out.println("Class-A Method : a = " + a);
      }
}
class B extends A
{
      int b;
      B(int p, int q)
      {
        super(p); // Calling Super class Construtor
        b=q;
      }
      void displayB()
      {
  // Refering Super class with super keyword
  System.out.println("Class-B Method : a = " + super.a + " b = " + b );
      }
}
class SuperDemo
{
     public static void main(String args[])
     {
      //Creating class B object by calling sub class constructor
      B objB = new B(500,1000);

      //calling class-B method
      objB.displayB();

      }
}
```

Output:

```
C:\>javac SuperDemo.java
C:\>java SuperDemo
```

```
Class-B Method : a = 500 b = 1000
```

9 Constructor Method and Inheritance

- For getting super class value, the keyword super is used.
- Second, it is used in defining the constructor of subclass.
- Instead of repeating the assignment of variables of super class, we simply qualify the variable with super.
  - Example: SuperDemo.java

10.Method Overriding

- It is one of the ways in which polymorphism can be implemented.
- When both super class and its subclass contain a method that has the same name and type signature, the super class definition of the method is overridden by definitions in subclass.
- It is different from the overloaded method in which only the name is same but parameter list has to be different either in type or in number of parameters or order of parameters.
- In the case of overloaded methods, the parameter lists are matched to choose the appropriate method that may be in super class or subclass.
- When two methods with the same name and type signature are defined in super (base) class as well as in subclass (derived class), the subclass definition overrides the super class definition when the method is called by object of subclass;
- It will execute the method defined in subclass and hide the definition of super class.

Binding

- It involves associating the method call to method body. There are two types of binding as follows:

Static binding : When the binding is performed at compile time by the compiler, it is known as static or early binding.

- For instance, binding for all static, private, and final methods is done at the compile time.

Dynamic binding : It is also called late binding. Here, the compiler is not able to resolve the call (or binding) at compile time.

- Method overriding is one such example where dynamic binding is involved.
- The basic difference between static and dynamic binding is that static binding occurs at compile time, whereas dynamic binding happens at run time.

Example: MethodOverriding.java

```
class A
{
  void display()
  {
       System.out.println("Super Class Method");
  }
}

class B extends A
```

```
{
  void display()
  {
        System.out.println("Sub Class Method");
  }
}

class MethodOverriding
{
     public static void main(String args[])
     {
  //calling the class A method
  A objA = new A();
  objA.display();

  //calling the class B method
  objA = new B();
  objA.display();
      }
}
```

Output:

```
C:\>javac MethodOverriding.java

C:\>java MethodOverriding
Super Class Method
Sub Class Method
```

11.Dynamic Method Dispatch

- It is a mechanism by which runtime polymorphism is achieved for overridden method in Java.
- It is implemented through super class reference. A super class reference can refer to an object of its subclass.
- A base class pointer can refer to derived class object. There may be many subclasses inherited from a super class.
- Each subclass has its own version or definition of the overridden method.
- The dynamic method dispatch chooses the right version of the method corresponding to the object reference.
- In the method, first a reference variable of super class is created.
- The value of subclass object is assigned to the variable and the overridden method is called by the super class reference.

Example: MethodOverriding.java

12.Abstract Classes

The **abstract** keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Example: Abstract.java

```
abstract class A
{
      int a;
      void setValue(int x)
      {
         a=x;
      }
      abstract void display();
}

class B extends A
{
      int b;
      void setValues(int x, int y)
      {
         a=x;
         b=y;
      }
      void display()
      {
      System.out.println("Class-B Method : a = " + a + " b = " + b);
      }
}

class C extends A
{
      int c;
      void setValues(int x, int y)
      {
         a=x;
         c=y;
      }
      void display()
      {
      System.out.println("Class-B Method : a = " + a + " c = " + c);
      }
}
```

```
class Abstract
{
     public static void main(String args[])
     {
      //calling class-B method
      System.out.println("Through objB");
      B objB = new B();
      objB.setValues(10,20);
      objB.display();

      //calling class-C method
      System.out.println("Through objC");
      C objC = new C();
      objC.setValues(150,250);
      objC.display();
       }
}
```

13.Interfaces and Inheritance.

- Multiple inheritance of classes is not permitted in Java.
- To some extent, this restriction can be overcome through interfaces.
- A class may implement more than one interface besides having one super class.
- An interface can extend one or more interfaces, and a class can also implement more than one interface.
- An interface is a collection of constants and abstract methods that are implemented by a class.
- An interface cannot implement itself like a class;
- An interface just contains the method head, and there is no method body. The class that implements the interface contains the full definition of the method.

Example: Multiple.java

III. Interfaces:

1. Introduction

- https://www.w3schools.com/java/java_interface.asp
  An interface is a completely "abstract class" that is used to group related methods with empty bodies
- https://www.javatpoint.com/interface-in-java
  An interface in Java is a blueprint of a class. It has static constants and abstract methods.
- https://www.geeksforgeeks.org/interfaces-in-java/
  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).
- An interface also introduces a **new reference type**.
- An interface represents an **encapsulation of constants, classes, interfaces,** and one or more abstract methods that are implemented by a class.
- An interface **does not contain instance variables**.
- **An interface cannot implement itself**; it has to be implemented by a class.
- **The methods in an interface have no body**.
- Only headers are declared with the parameter list that is followed by a semicolon.
- The **class that implements the interface has to have full definitions of all the abstract methods** in the interface.
- **An interface can be implemented by any number of classes** with their own definitions of the methods of the interface.
- **Different classes can have different definitions of the same methods but the parameter list must be identical to that in the interface**.
- Thus, **interfaces provide another way of dynamic polymorphic implementation of methods**.
- **Any number of interfaces can be implemented by a class.**
- This fulfils the need for multiple inheritance.
- **The multiple inheritances of classes are not allowed in Java**, and therefore, interfaces provide a stopgap arrangement.

Similarities between Interface and Class

- Declaring an interface is similar to that of class; the keyword *class* is replaced by keyword *interface*.
- Its accessibility can be controlled just like a class.
- An interface declared public is accessible to any class in any package, whereas the ones without an access specifier is accessible to classes in the same package only.
- One can create variables as object references of interface that can use the interface.
- It can contain inner classes (nested classes) and inner interfaces.
- Since Java 8, an interface can have full definitions of methods with default or static modifiers.

Types of Interfaces

Top level interfaces

- It is an interface that is not nested in any class or interface.
- It comprises a collection of abstract methods.
- It can contain any number of methods that are needed to be defined in the class.

Nested interface

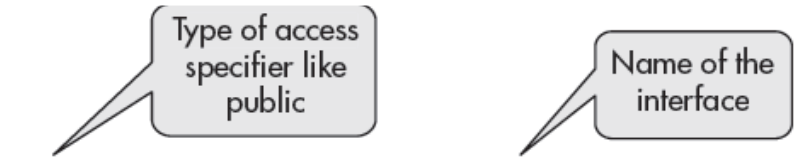
- It is an interface that is defi ned in the body of a class or interface.
- In nested interfaces, one or more interfaces are grouped, so that it becomes easy to maintain.
- It is referred to by the outer interface or class and cannot be accessed directly.

Generic interface

- Like a class, an interface is generic if it declares one or more types of variables.
- It comprises methods that accept or return an object.
- Thus, we can pass any parameter to the method that is not of the primitive type.

2. Declaration of Interface

- Declaration of an interface starts with the access modifier followed by keyword interface.
- It is in then followed by its name or identifier that is followed by a block of statements;
- These statements contain declarations of variables and abstract methods.
- The variables defined in interfaces are implicitly public, static, and final.
- They are initialized at the time of declaration. The methods declared in an interface are public by default.

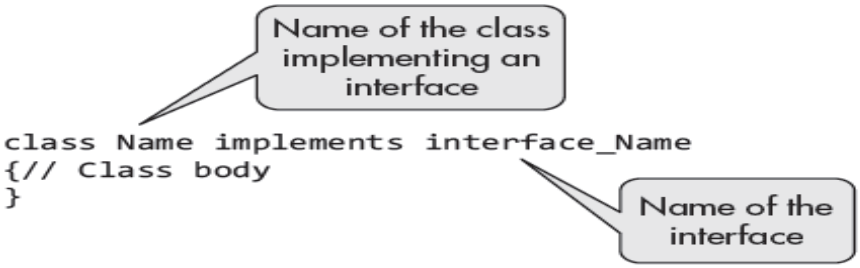


Members of Interface

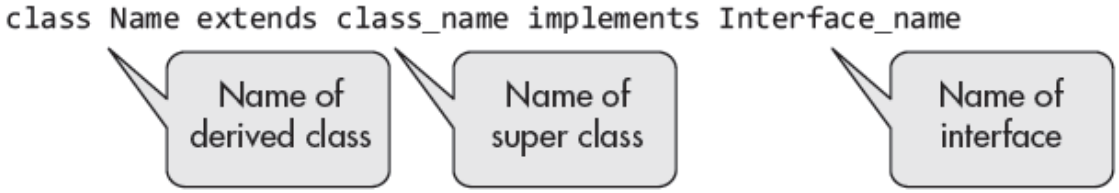
- The members declared in the body of the interface.
- The members inherited from any super interface that it extends.
- The methods declared in the interface are implicitly public abstract member methods.
- The field variables defined in interfaces are implicitly public, static, and final.
- However, the specification of these modifiers does not create a compile-type error.
- The field variables declared in an interface must be initialized; otherwise, compile-type error occurs.
- Since Java SE8, static and default methods with full definition can also be members of interface.

3. Implementation of Interface

Declaration of class that implements an interface

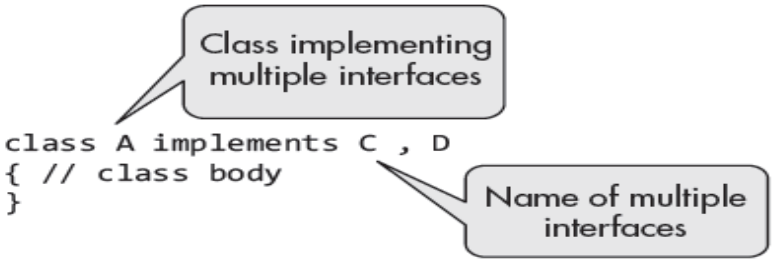


If a class extends another class as well as implements interfaces, it is declared as



4. Multiple Interfaces

- Multiple interfaces can also be implemented in Java.
- For this, the class implements all the methods declared in all the interfaces.
- When the class is declared, names of all interfaces are listed after the keyword *implements* and separated by comma.
- As for example, if class A implements interfaces C and D, it is defined as



Example: Multiple.java

Interface References:

- For interface references, variables can be declared as object references.
- In this case, the object reference would use interface as the type instead of class.
- The appropriate method is called on the basis of actual instance of the interface that is being referred to.

Example: InterfaceRef.java

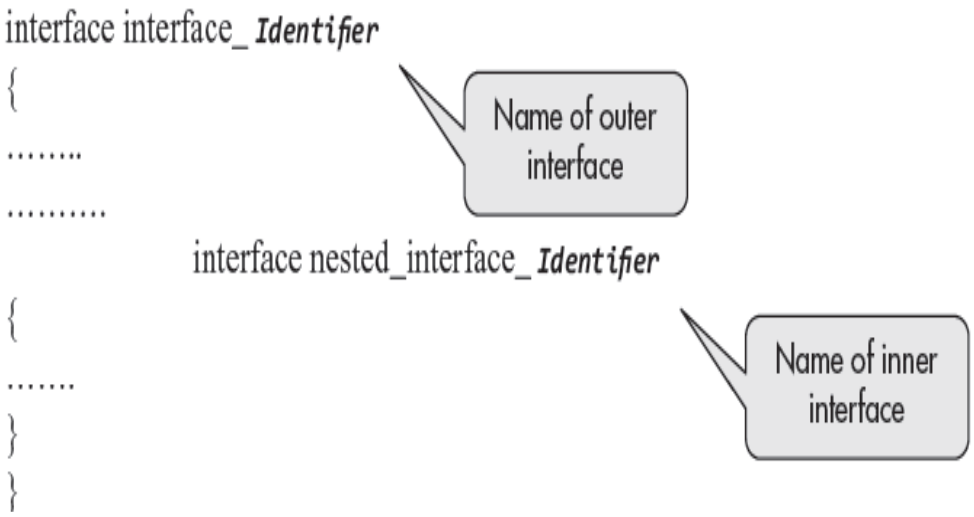
```
interface X
{
      int x = 10;
      public void display();
}

interface Y
{
      int y = 20;
      public void add();
}
class A implements X,Y
{
      public void display()
      {
      System.out.println("Class-B Method : x = "+ x + " y = " + y);
      }
      public void add()
      {
      System.out.println("Class-B Method : x+y = "+ (x+y));
      }
}

class InterfaceRef
{
     public static void main(String args[])
     {
      //Reference of X
      X objX = new A();
      objX.display();

      //Reference of Y
      Y objY = new A();
      objY.add();
       }
}
```

Output:

```
C:\ >javac InterfaceRef.java

C:\ >java InterfaceRef
Class-B Method : x = 10 y = 20
Class-B Method : x+y = 30
```

5. Nested Interfaces

- An interface may be declared as a member of a class or in another
- interface.
- In the capacity of a class member, it can have the attributes that are applicable to other class members.
- In other cases, an interface can only be declared as public or with default (no-access modifier) access.
- Syntax of nested interface in another interface is given as



Example: NestedInterface.java

```
interface OuterX
{
   int x = 10;
   public interface InnerY
   {
         int y = 20;
   }
}

class A implements OuterX, OuterX.InnerY
{
   void display()
   {
         System.out.println("Class-A Method : x = "+ x + " y
= " + y);
   }
}

class NestedInterface
{
     public static void main(String args[])
     {
   //calling class-A method
   A objA = new A();
   objA.display();
      }
}
```

Output:

```
C:\ >javac NestedInterface.java

C:\ >java NestedInterface
Class-A Method : x = 10 y = 20
```

6. Inheritance of Interfaces

Inheritance of Interfaces is similar to the Inheritance of classes. Interface can be derived from another interface.

```
Syntax:

access_specifier interface  NewInterface  extends OldInterface
{
  //Body of the interface
}

Example:

interface A{}
interface B{}
interface C extends A,B
{
//Body of the interface
}
```

Example Program: InterfaceInheritance.java

```
interface X
{
   int x = 10;
}

interface Y extends X
{
   int y = 20;
}

class A implements Y
{
   void display()
   {
System.out.println("Class-A Method : x = "+ x + " y = " + y);
   }
}
class InterfaceInheritance
{
     public static void main(String args[])
     {
       //calling class-A method
       A objA = new A();
       objA.display();
     }
```

```
}
Output:
C:\1. JAVA\UNIT-3.3>javac InterfaceInheritance.java

C:\1. JAVA\UNIT-3.3>java InterfaceInheritance
Class-A Method : x = 10 y = 20
```

7. Default Methods in Interfaces

- The enhancement in Java 8 allowing the interface to have full definition of default methods and static methods that are implicitly inherited by the class implementing the interface.
- Java allows only one super class, by using default methods in interface java allows that interface behaves just like an abstract super-class.
- New functionality can be added to the interface, which is inherited by classes implementing the interface.
- The inherited methods are also members of the class, and therefore, these maybe called other methods of class.
- A default method cannot be declared final.
- A default method cannot be synchronized; however, blocks of statements in the default method may be synchronized.
- The object class is inherited by all classes. Therefore, a default method should not override any non- final method of object class.
-

A default method is declared with keyword *default* as

```
public interface A {
default void display () {System.out.println("It is
interface A.");}
```

Keyword default for specifying default method

Example : DefaultMethods.java

```
interface X
{
   int x = 10;
   default void display()
   {
      System.out.println("Method in Interface X the value x
= " + x);
   }
}

class A implements X
{

}


class DefaultMethods
{
     public static void main(String args[])
     {
   //calling class-A method
   A objA = new A();
   objA.display();
      }
}
```

Output:

```
C:\>javac DefaultMethods.java

C:\>java DefaultMethods
Method in Interface X the value x = 10
```

8. Static Methods in Interface

- The Java version 8 allows full definition of static methods in interfaces.
- A static method is a class method.
- For calling a static method, one does not need an object of class.
- It can simply be called with class name as

```
class_name.method_name()
```

Example: StaticMethods.java

```
interface X
{
   int x = 10;
   static void display()
   {
      System.out.println("Method in Interface X the value x
= " + x);
   }
}

class StaticMethods
{
     public static void main(String args[])
     {
   //calling static method by specifying interface X
   X.display();
      }
}
Output:
C:\>javac StaticMethods.java

C:\>java StaticMethods
Method in Interface X the value x = 10
```

Additional Information

Java Lambda Expressions

- Lambda Expressions were added in Java 8.
- A lambda expression is a **short block of code** which takes in parameters and returns a value.
- Lambda expressions are **similar to methods**, but they **do not need a name** and **they can be implemented right in the body of a method**.

Syntax
The simplest lambda expression contains a single parameter and an expression:

```
Parameter -> expression
```

To use more than one parameter, wrap them in parentheses:

```
(Parameter1, Parameter2) -> { Code of Block }
```

```
Example1:
import java.util.ArrayList;

public class LambdaExpressions
{
      public static void main(String[] args)
      {
      ArrayList<Integer> numbers = new ArrayList<Integer>();
      numbers.add(5);
      numbers.add(9);
      numbers.add(8);
      numbers.add(1);
      numbers.forEach((n) -> { System.out.println(n); });
      }
}

C:\>javac LambdaExpressions.java

C:\>java LambdaExpressions
5
9
8
1

C:\1. JAVA\UNIT-3.3>
```

9. Functional Interfaces

- In Java SE8, a new package java.util.function on functional interfaces has been introduced for writing Lambda functions.
- Functional interfaces are interfaces with one abstract method.
- They are also called SAM or single abstract method type.
- However, a functional interface can have more than one static and default methods.
- The programmer may include an annotation, to lessen the work of complier.
  @FunctionalInterface
- By adding the above annotation, it can be helpful in detecting compile time errors.
- If the functional interface contains more than one abstract method, the compiler will throw an error.

Example:Funcanal.java

```
import java.util.function.Function;
import java.util.function.BinaryOperator;

class Functional
{
     public static void main(String args[])
     {
   //declaring logrithm and minimum as functional interfaces
   Function <Double, Double> logrithm = Math::log;
   BinaryOperator<Integer> minimum = Math::min;

   //calling logrithm.apply() which is method reference to
the Function
   System.out.println("Log of 10 to the base e = "+
logrithm.apply(10.0));

   //calling minimum.apply() which is method reference to
the BinaryOperator
   System.out.println("Minimum of 20 and 46 is    "+
minimum.apply(20,46));
     }
}
```

Output:

```
C:\ >javac Functional.java

C:\ >java Functional
Log of 10 to the base e = 2.302585092994046
Minimum of 20 and 46 is  20
```

Functional Consumer<T>

- The interface declaration is

```
@FunctionalInterface
public interface Consumer {void accept(T t);}
```

- It declares one abstract method void accept(T t).
- The method only consumes its argument. It does not give any return value.

Example: ConsumerDemo.java

```
import java.util.function.Consumer;

class ConsumerDemo
{
      public static void main(String args[])
      {
      //declaring logrithm and minimum as functional interfaces
      System.out.print("double value =  ") ;
      Consumer<Double> FunInt1 = (Double d) -> { display(d); };
      FunInt1.accept(3.14);

      System.out.print("\nString Array =  ") ;
      Consumer<String> FunInt2 = (String s) -> { display(s); };
      String[] sray = {"CSE","ECE","CIVIL"};
      for(String str: sray)
        FunInt2.accept(str);

      System.out.print("\nInteger Array =  ") ;
      Consumer<Integer> FunInt3 = (Integer n) -> { display(n); };
      Integer[] iray = {1,2,3,4,5};
      for(Integer num: iray)
        FunInt3.accept(num);
      }
      public static<T> void display(T t)
      {
            System.out.print(t +" ");
      }
}
```

Output:

```
C:\ >javac ConsumerDemo.java

C:\ >java ConsumerDemo
double value =  3.14
String Array =  CSE ECE CIVIL
Integer Array =  1 2 3 4 5
```

10.Annotations.

- Annotation framework in Java language was first introduced in Java 5 through a provisional interface Apt; It is **a type of metadata** that can be **integrated with the source code without affecting the running of the program**.
- Annotations may be retained up to runtime and may be used to instruct the compiler and runtime system to do or not to do certain things.
- Since Java SE 8, the annotations may be applied to classes, fields, interfaces, methods, and type declarations like throw clauses.
- The annotations are no longer simply for metadata inclusion in the program but have become a method for user's communication with compiler or runtime system.
- An annotation like @Override consists of two distinct words @ and Override.
- It may as well be written as @ Override;
- The name Override is the name of the interface that defines the annotation.
- There are a number of annotations that are predefined and are part of the package java.lang.annotation.
- However, a programmer may also defi ne an annotation.

Example: Annotational.java

```
class A
{
 public void display()
 {
      System.out.println("In class A");
 }
}

class B extends A
{
 @Override public void display()
 {
      System.out.println("In class B");
 }
}

class C extends A
{
 @Override public void display()
 {
      System.out.println("In class C");
 }
}
class Annotational
{
      public static void main(String args[])
      {
      A objA = new A();
      B objB = new B();
      C objC = new C();
```

```
      objA.display();
      objB.display();
      objC.display();
 }
}
```

Output:

```
C:\1. JAVA\UNIT-3.3>javac Annotational.java

C:\1. JAVA\UNIT-3.3>java Annotational
In class A
In class B
In class C
```

If we change the method name in class C as displayC( ), Error will generate as

```
C:\1. JAVA\UNIT-3.3>javac Annotational.java
Annotational.java:19: error: method does not override or implement a
method from a supertype
        @Override public void displayC()
        ^
1 error
```