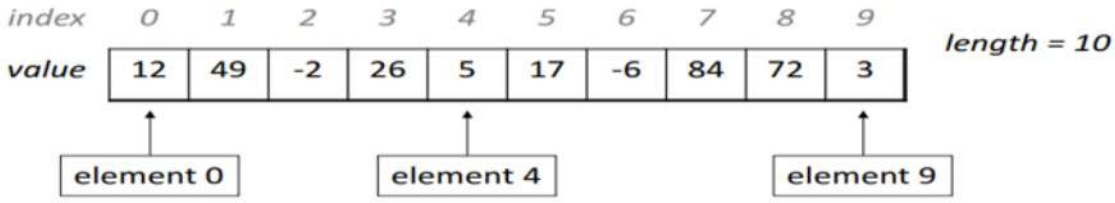
UNIT -III: Arrays, Inheritance and Interfaces

Array:

An *array* is a container object that holds a **fixed number of values of a single type**. So all elements in an array must be of the same data type and are stored in contiguous memory locations.

Each item in an array is called an *element*, and each element is accessed by its *index* and index value begins with 0.



Types of arrays:

Java supports primarily two types of arrays

a) Single dimensional array
b) Multi dimensional array

a) single dimensional array:

A single dimensional array(or) one dimensional array(1d array) is an array where elements are stored in a linear fashion i.e., in a single row

Declaration, creation and Initialization of one dimensional array:

A one dimensional array can be declared as follows

type[] arrayname;

where type is the data type of the contained elements and arrayname is any valid identifier.

After declaring an array we can create an array of the specified type using new operator as follows.

arrayName = new type[length];

This statement allocates an array with enough memory to hold the number of elements specified by length.

Alternatively we can declare and create an array by combining above two statements as follows

type[ ] arrayname = new type[length];

Ex1:

```
int[ ] numbers; // declares an array of integers
numbers=new int[5];  //allocate memory for 5 integer elements

(or)

int[ ] numbers = new int[5];
```

Ex2:

```
double[] salary;    //declares an array of double values
salary= new double[10];

(or)
double[ ] salary = new double[10];
```

Ex3:

```
String[] colors = new String[3];
            //declares an array and allocates memory for 3 string elements
```

Note1:

The declaration does not actually create an array rather it simply tells the compiler that this variable will hold an array of the specified type.

Note2:

We can also declare an array like this: All the three following syntax are valid for array declaration.

int[] myArray;

int []myArray;

int myArray[];

Initialization of arrays:

Array initialization involves creating an array object and assigning values to its elements. There are two ways to assign the values (or) to store the values in to the array

Method1:

We can declare a one dimensional array and directly initialize the elements at the time of declaration as follows.

type[ ] arrayName={comma separated elements};

Ex1:

```
int[] numbers = {10,20,30,40,50};
```

This line declares an integer array named numbers and initializes it with the given values. The length of the array is determined by the number of elements provided.

Ex2:

```
String[] fruits = {"apple", "banana", "orange"};
boolean[] isEven = {true, false, true};
double[] prices = {2.99, 4.50, 1.99};
```

Method2:

After declaring an array and allocating memory, assign the elements one by one using their respective indices.

Ex:

```
int[] numbers = new int[5]; // Declares an integer array of size 5

// Initialize elements
```

```
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;
```

Ex:

```
String[] colors = new String[3];

colors[0] = "red";
colors[1] = "green";
colors[2] = "blue";
```

Accessing array elements:

Array elements are accessed using their **indices**. Indexing starts from **0**, meaning the first element has an index of 0, the second has an index of 1, and so on. The last element's index is always one less than the total number of elements in the array.

To access an element, use the array name followed by the index enclosed in square brackets [].

This is illustrated in the following example

Ex:

```
Consider the following array

int[] numbers = {10, 20, 30, 40, 50};

// Accessing elements
int firstElement = numbers[0]; // Accesses the first element (10)
int thirdElement = numbers[2]; // Accesses the third element (30)
int lastElement = numbers[numbers.length - 1]; // Accesses the last element (50)
```

Note:

Trying to access an element with an index outside the valid range (0 to length - 1) will result in an ArrayIndexOutOfBoundsException.

Length of an Array

For 1D arrays, the length of an array indicates the number of elements it contains and is established when the array is created. After creation, the length is fixed and cannot be changed.

Java provides the built-in property "length" to determine the length of an array. This property is available for all array types and returns the number of elements in the array

For example, consider the array

```
int[] numbers = new int[5];
```

now numbers.length will give number of elements in the array i.e., 5

Reading the data into the array using keyboard:

we can store the values in to the array from the keyboard. This can be done using a loop in which the index can be varied as shown below.

For example, consider the array

```
int[] numbers = new int[5];
```

Now the statement

```
for(int i=0;i<numbers.length;i++)
        numbers[i] = sc.nextInt();     //sc is a Scanner class object
```

will read 5 elements from the keyboard and store it in the array

Printing the array elements:

similarly to print the array elements, we can write the statement

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

Ex1:

```
// Java program to read elements into the array and printing them
import java.util.Scanner;

public class ArrayDemo
{
```

```
    public static void main(String []args)
    {
       Scanner sc=new Scanner(System.in);
       System.out.println("enter the total number of elements:");
       int n=sc.nextInt();
       int[] source = new int[n];

       //read all the elements using for loop
        System.out.println("enter the "+ n +" elements:");
        for(int i=0;i<source.length;i++)
              source[i]=sc.nextInt();

       //printing all the elements using for loop
       System.out.println("the elements you entered are");
       for(int i=0;i<source.length;i++)
             System.out.print(source[i]+"\t");

    }
}
```

```
enter the total number of elements:
5
enter the 5 elements:
34 45 56 67 2
the elements you entered are
34      45      56      67      2
```

Ex2:

```
// Java program to find Maximum and minimum of array elements
import java.util.Scanner;

public class ArrayDemo
{
    public static void main(String []args)
    {
          Scanner sc=new Scanner(System.in);
       System.out.println("enter the total number of elements:");
       int n=sc.nextInt();
       int[] source = new int[n];

       //read all the elements using for loop
        System.out.println("enter the "+ n +" elements:");
        for(int i=0;i<source.length;i++)
```

```
            source[i]=sc.nextInt();

        //finding maximum and minimum elements
        int max=source[0];
        int min=source[0];

        for(int i=0;i<source.length;i++)
        {
           if(max<source[i])
              max=source[i];

           if(min>source[i])
              min=source[i];

        }
        //printing the maximum and minimum element
        System.out.println("The maximum element is "+max);
        System.out.println("The minimum element is "+min);

    }
}
```

Output:

```
enter the total number of elements:
6
enter the 6 elements:
34 45 56 67 20 -2
The maximum element is 67
The minimum element is -2
```

Ex3:

```
// Java program to illustrate creating an array of objects
class Student
{
   public int roll_no;
   public String name;
   Student(int roll_no, String name)
   {
      this.roll_no = roll_no;
      this.name = name;
   }
}
```

```
// Elements of the array are objects of a class Student.
public class Demo {
    public static void main(String[] args)
    {
        // declares an Array of Student
        Student[] arr;

        // allocating memory for 5 objects of type Student.
        arr = new Student[5];

        // initialize the first elements of the array
        arr[0] = new Student(1, "pavan");

        // initialize the second elements of the array
        arr[1] = new Student(2, "kumar");


        arr[2] = new Student(3, "raju");
        arr[3] = new Student(4, "krishna");
        arr[4] = new Student(5, "gowtham");

        // accessing the elements of the specified array
        for (int i = 0; i < arr.length; i++)
            System.out.println("Element at " + i + " : "
                               + arr[i].roll_no + " "
                               + arr[i].name);
    }
}
```

Output:

```
Element at 0 : 1 pavan
Element at 1 : 2 kumar
Element at 2 : 3 raju
Element at 3 : 4 krishna
Element at 4 : 5 gowtham
```

Operations on Array Elements:

Here are some common operations performed on arrays.

a.Traversing through the array

b.asigning Array to Another Array

c.Sorting of arrays

d.Search for values in Arrays

a) Traversing through the array:

We can use a loop (for, while, or enhanced for) to process each element in the array.

```
// Java program to read elements into the array and printing them
import java.util.Scanner;

public class ArrayDemo
{
   public static void main(String []args)
   {
      Scanner sc=new Scanner(System.in);
      System.out.println("enter the total number of elements:");
      int n=sc.nextInt();
      int[] source = new int[n];

      //read all the elements using for loop
       System.out.println("enter the "+ n +" elements:");
       for(int i=0;i<source.length;i++)
            source[i]=sc.nextInt();

      //Traversing and printing all the elements using enhanced for loop
      System.out.println("the elements you entered are");
     for(int i: source)
        System.out.println(i);
   }
}
```

Output:

```
enter the total number of elements:
4
```

```
enter the 4 elements:
23 45 67 8
the elements you entered are
23
45
67
8
```

b) Assigning one array to another:

we can assign one array to another array. When you assign one array to another, both variables refer to the same array object. This means any changes made to one array will affect the other.

```
//program to illustrate assigning one array to another array
public class ArrayAssignment {
    public static void main(String[] args) {
        int[] originalArray = {1, 2, 3, 4, 5};

        // Assigning the original array to another array
        int[] anotherArray = originalArray;

        // Modifying the original array
        originalArray[0] = 10;

        // Printing both arrays
        System.out.println("Original Array: ");
        for(int i:originalArray)
            System.out.println(i);

       System.out.println("another Array: ");
        for(int i:anotherArray)
            System.out.println(i);
    }
}
```

Output:

```
Original Array:
10
2
3
4
5
```

```
another Array:
10
2
3
4
5
```

c) Sorting of arrays

Sorting an array in Java can be done using various techniques, (implementing algorithms like Bubble Sort, insertion Sort, etc.)

```
//java program to sort the array elements using Bubble Sort
import java.util.Scanner;

public class BubbleSort {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input the size of the array
        System.out.println("Enter the number of elements: ");
        int n = sc.nextInt();

        // Input the elements of the array
        int[] array = new int[n];
        System.out.println("Enter the elements: ");
        for (int i = 0; i < n; i++) {
            array[i] = sc.nextInt();
        }

        // Perform bubble sort
        bubbleSort(array);

        // Display the sorted array
        System.out.println("Sorted array: ");
        for (int i : array) {
            System.out.print(i + " ");
        }
    }

    // Method to perform bubble sort
    public static void bubbleSort(int[] array) {
        int n = array.length;
```

```
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - 1 - i; j++) {
                if (array[j] > array[j + 1]) {
                    // Swap array[j] and array[j + 1]
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }
}
```

Output:

```
Enter the number of elements:
5
Enter the elements:
1 -5 4 2 89
Sorted array:
-5 1 2 4 89
```

d) Search for values in Arrays

Searching for values in an array in Java can be done using various techniques, depending on the type of search you want to perform. The most common methods are **linear search** and **binary search**

```
// Java Program to perform binary search
import java.util.Scanner;


public class BinarySearch {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);


        // Input the size of the array

        System.out.println("Enter the number of elements: ");

        int n = sc.nextInt();
```

```
        // Input the elements of the array
        int[] array = new int[n];
        System.out.println("Enter the elements (sorted): ");
        for (int i = 0; i < n; i++) {
            array[i] = sc.nextInt();
        }

        // Input the element to be searched
        System.out.println("Enter the element to search: ");
        int key = sc.nextInt();

        // Perform binary search
        int result = binarySearch(array, key);

        // Display the result
        if (result == -1) {
            System.out.println("Element not found in the array.");
        } else {
            System.out.println("Element found at index: " + result);
        }

        sc.close();
    }
```

```
// Method to perform binary search
public static int binarySearch(int[] array, int key) {
    int left = 0;
    int right = array.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        // Check if key is present at mid
        if (array[mid] == key) {
            return mid;
        }

        // If key is greater, ignore the left half
        if (array[mid] < key) {
            left = mid + 1;
        }
        // If key is smaller, ignore the right half
        else {
            right = mid - 1;
        }
    }

    // Key not found
    return -1;
}
```

| }
|
| Output:

Enter the number of elements:

6

8Enter the elements (sorted):

20 30 40 50 60 70

Enter the element to search:

50

Element found at index: 3 |

Note:

Dynamic Change of Array Size

In Java, arrays are of fixed size, meaning you cannot dynamically change the size of an array after it has been created. However, you can simulate dynamic resizing by creating a new array with the desired size and copying the elements from the original array to the new one. Java's ArrayList class is often used to handle dynamic resizing internally.

Class Arrays:

The Arrays class is a part of the java.util package and provides several static methods to perform common array operations easily. It is a utility class that offers functionalities like sorting, searching, filling, comparing, and converting arrays to strings. The Arrays class is essential for working with arrays in Java and simplifying common array-related operations.

Some commonly used methods are

a) Arrays.sort() method:

This method Sorts the elements of an array into ascending order.

Ex:

```
int[] numbers = {3, 5, 1, 2, 4};
Arrays.sort(numbers);
System.out.println(Arrays.toString(numbers)); // Output: [1, 2, 3, 4, 5]
```

b) Arrays.binarySearch() method:

This method Searches for a specific value in a sorted array using the binary search algorithm. Returns the index of the value if found, or a negative value if not found.

```
int[] numbers = {1, 2, 3, 4, 5};
int index = Arrays.binarySearch(numbers, 3);
System.out.println(index); // Output: 2
```

c) Arrays.fill() method:

This method Fills all elements of an array with a specified value

```
int[] numbers = new int[5];
Arrays.fill(numbers, 7);
System.out.println(Arrays.toString(numbers)); // Output: [7, 7, 7, 7, 7]
```

d) Arrays.equals() method:

This method Compares two arrays for equality. Returns true if both arrays are of the same length and contain the same elements in the same order.

```
int[] arr1 = {1, 2, 3};
int[] arr2 = {1, 2, 3};
boolean areEqual = Arrays.equals(arr1, arr2);
System.out.println(areEqual); // Output: true
```

Two dimensional arrays:

A two-dimensional array in Java is a collection of elements arranged in rows and columns. It can be thought of as an array of arrays. Each element in the array is identified by two indices

- **Row index:** Specifies the row number.
- **Column index:** Specifies the column number.

Declaration, creation and Initialization of two dimensional array:

A two dimensional array can be declared as follows

type[][] arrayname;

where type is the data type of the contained elements and arrayname is any valid identifier.

After declaring an array,two dimensional array with the same number of columns in every row can be created as follows.

arrayName = new type[numrows][numcols];

This creates a 2D array with numrows number of rows and numcols number of columns.

Alternatively we can declare and create an array by combining above two statements as follows

type[ ][ ] arrayname = new type[numrows][numcols];

Ex:

int[ ][ ] marks = new int[3]][3];

this will create a two-dimensional array named marks containing 3 rows and each row contains maximum of 3 elements

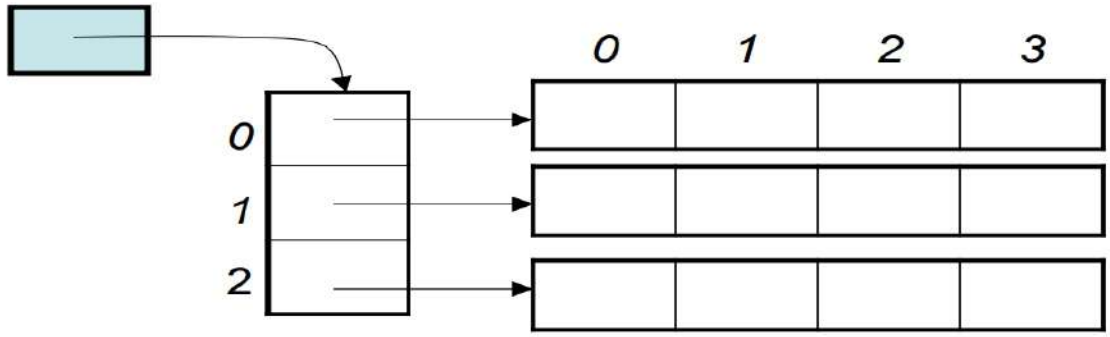
		Column numbers		
		0	1	2
Row numbers	0	Marks[0][0]	Marks[0][1]	Marks[0][2]
	1	Marks[1][0]	Marks[1][1]	Marks[1][2]
	2	Marks[2][0]	Marks[2][1]	Marks[2][2]

2D Array Implementation:

A 2D array is a 1D array of (references to) 1D arrays.

Consider the following 2D array

```
int[][] rating = new int[3][4];
```



Initialization of arrays:

Array initialization involves creating an array object and assigning values to its elements. There are two ways to assign the values (or) to store the values in to the array

Method1:

We can declare a two dimensional array and directly initialize the elements at the time of declaration as follows.

type[ ][ ] arrayName= {{row1 initializer}, {row2 initializer}, …};

Ex1:

int[] [ ] numbers = {{10,20,30},{40,50,60}};

Method2:

After declaring an array and allocating memory, assign the elements one by one using their respective indices.

Ex:

```
int[][] source = new int[2][2];

source[0][0] =10;
source[0][1] =20;
source[1][0] =30;
source[1][1] =40;
```

Length of 2D array:

Java provides the built-in property length, which can be used to determine the number of rows in a 2D array. However, to determine the number of columns, you need to access the length property of a specific row.

For example, Consider the following 2D array.

```
int[][] numbers= new int[3][4];
```

now

numbers.length will give number of rows i.e., 3

and numbers[0].length will give number of columns i.e., 4

Reading the data into the 2D array using keyboard:

We can store values into a 2D array from the keyboard by using nested loops, where the row and column indices are varied as shown below.

For example, consider the array

```
int[][] numbers = new int[3][4];
```

To read 12 elements from the keyboard and store them in the array, you can use the following code

```
for(int i=0;i<numbers.length;i++)
        for(int j=0;j<numbers[i].length;j++)
            numbers[i][j] = sc.nextInt();     //sc is a Scanner class object
```

This code uses two loops. The outer loop iterates over the rows, and the inner loop iterates over the columns, allowing you to fill each element of the 2D array.

Printing the array elements:

Similarly, to print the array elements, you can use the following code:

```
for (int i = 0; i < numbers.length; i++) {
    for (int j = 0; j < numbers[i].length; j++) {
        System.out.print(numbers[i][j] + " ");
    }
    System.out.println();  // Moves to the next line after each row
}
```

This code will print the elements of the 2D array in a matrix format, with each row's elements on a new line.

```
// Java program to read elements into the 2Darray and printing them
import java.util.Scanner;

public class ArrayDemo
{
    public static void main(String []args)
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("enter the total number of rows:");
        int m=sc.nextInt();
        System.out.println("enter the total number of columns:");
        int n=sc.nextInt();
        int[][] source = new int[m][n];

        //read all the elements using for loop
         System.out.println("enter the array elements:");
         for(int i=0;i<source.length;i++)
            for(int j=0;j<source[i].length;j++)
                source[i][j]=sc.nextInt();

        //printing all the elements using for loop
        System.out.println("the elements you entered are");
        for (int i = 0; i < source.length; i++)
        {
            for (int j = 0; j < source[i].length; j++)
            {
                    System.out.print(source[i][j] + " ");
            }
```

```
            System.out.println();  // Moves to the next line after each row
        }


    }
}
```

Output:

```
enter the total number of rows:
3
enter the total number of columns:
4
enter the array elements:
1 2 3 4 10 12 14 16 20 22 24 26
the elements you entered are
1 2 3 4
10 12 14 16
20 22 24 26
```

Ex: Write a java program to perform matrix addition

```
import java.util.Scanner;

public class MatrixAddition {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input the size of the matrices
        System.out.print("Enter the number of rows of first matrix: ");
        int r1 = sc.nextInt();
        System.out.print("Enter the number of columns of first matrix: ");
        int c1 = sc.nextInt();

         System.out.print("Enter the number of rows of second matrix: ");
        int r2 = sc.nextInt();
        System.out.print("Enter the number of columns of second matrix: ");
        int c2 = sc.nextInt();


        // Initialize two matrices and the result matrix
        int[][] matrix1 = new int[r1][c1];
        int[][] matrix2 = new int[r2][c2];
```

```
        if((r1==r2) && (c1==c2))
        {
            System.out.println("Matrix addition is possible");
            int[][] result = new int[r1][c1];

             // Input elements for the first matrix
            System.out.println("Enter the elements of the first matrix:");
            for (int i = 0; i < r1; i++) {
                for (int j = 0; j < c1; j++) {
                    matrix1[i][j] = sc.nextInt();
                }
            }

            // Input elements for the second matrix
            System.out.println("Enter the elements of the second matrix:");
            for (int i = 0; i < r2; i++) {
                for (int j = 0; j < c2; j++) {
                    matrix2[i][j] = sc.nextInt();
                }
            }

             // Perform matrix addition
             for (int i = 0; i < r1; i++) {
                for (int j = 0; j < c1; j++) {
                    result[i][j] = matrix1[i][j] + matrix2[i][j];
                }
            }
            // Output the result
            System.out.println("Resultant Matrix after Addition:");
            for (int i = 0; i < r1; i++) {
                for (int j = 0; j < c1; j++) {
                    System.out.print(result[i][j] + " ");
                }
                System.out.println(); // Move to the next line after each row
            }

        }
        else
                System.out.println("matrix addition is not possible");

        sc.close();
    }
}
```

```
Output:

Enter the number of rows of first matrix: 2
Enter the number of columns of first matrix: 2
Enter the number of rows of second matrix: 2
Enter the number of columns of second matrix: 2
Matrix addition is possible
Enter the elements of the first matrix:
1 2 3 4
Enter the elements of the second matrix:
2 3 4 5
Resultant Matrix after Addition:
3 5
7 9
```

Arrays of varying lengths or jagged arrays:

In Java, arrays of varying lengths are often referred to as **jagged arrays** or **ragged arrays**. These are arrays of arrays where each sub-array can have a different length.

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

        // Print the jagged array
        for (int i = 0; i < jaggedArray.length; i++) {
            for (int j = 0; j < jaggedArray[i].length; j++) {
                System.out.print(jaggedArray[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

```
Output:

1 2
3 4 5
6 7 8 9
```

Vectors in Java

The Vector class in Java is part of the Java Collections Framework and provides a dynamic array that can grow or shrink as needed. It is found in the java.util package and implements the List interface, making it similar to an ArrayList but with some key differences.

Key Characteristics

1. **Dynamic Size**: Unlike arrays, vectors can grow or shrink in size dynamically to accommodate the addition and removal of elements.
2. **Synchronized**: Vectors are synchronized, meaning they are thread-safe. This makes them suitable for use in multi-threaded environments, but it also means they have a performance overhead compared to non-synchronized collections like ArrayList.

Here are some commonly used methods of the Vector class:

1. **void addElement(Object element):** It inserts the element at the end of the Vector.
2. **int capacity():** This method returns the current capacity of the vector.
3. **int size():** It returns the current size of the vector.
4. **void setSize(int size):** It changes the existing size with the specified size.
5. **boolean contains(Object element):** This method checks whether the specified element is present in the Vector. If the element is been found it returns true else false.
6. **boolean containsAll(Collection c):** It returns true if all the elements of collection c are present in the Vector.
7. **Object elementAt(int index):** It returns the element present at the specified location in Vector.
8. **Object firstElement():** It is used for getting the first element of the vector.
9. **Object lastElement():** Returns the last element of the array.
10. **Object get(int index):** Returns the element at the specified index.
11. **boolean isEmpty():** This method returns true if Vector doesn't have any element.
12. **boolean removeElement(Object element):** Removes the specifed element from vector.
13. **boolean removeAll(Collection c):** It Removes all those elements from vector which are present in the Collection c.
14. **void setElementAt(Object element, int index):** It updates the element of specifed index with the given element.

```
import java.util.*;
public class VectorDemo
{
  public static void main(String[] args)
  {
      Vector<String> v=new Vector<String>();
      v.addElement("A");
      v.addElement("B");
      v.addElement("C");
      v.addElement("D");
      v.addElement("E");
      System.out.println(v);
      System.out.println(v.firstElement());
      System.out.println(v.lastElement());
      System.out.println(v.elementAt(3));
      v.removeElement("D");
      System.out.println(v);
      v.removeElementAt(2);
      System.out.println(v);
      v.removeAllElements();
      System.out.println(v);
  }
}
```

Output:

```
[A, B, C, D, E]
A
E
D
[A, B, C, E]
[A, B, E]
[]
```

arrays vs vectors:

Feature	arrays	Vectors
Size	Fixed size	Dynamic size
Synchronization	Not synchronized	Synchronized

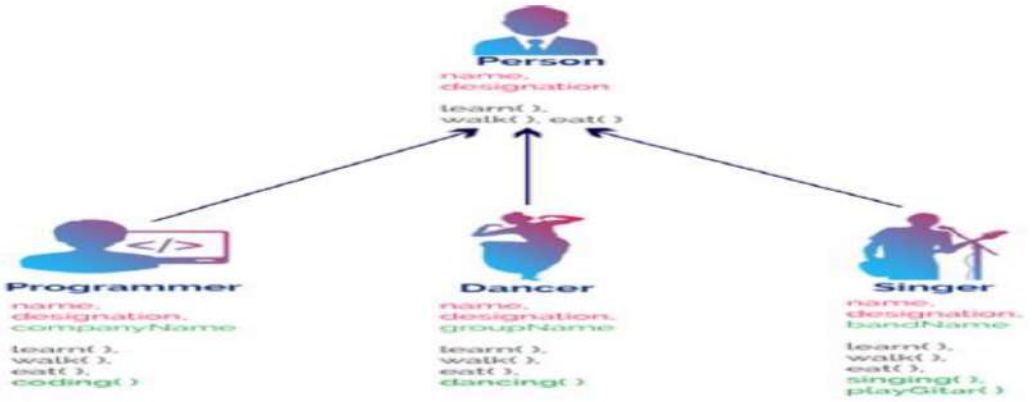
Performance	Faster due to no synchronization overhead	Slower due to synchronization overhead
Thread Safety	Not thread-safe	Thread-safe
Growth Policy	Fixed size, cannot grow	Doubles its size when needed
Legacy	Part of core Java since JDK 1.0	Legacy class, part of core Java since JDK 1.0
Usage	Suitable for single-threaded environments	Suitable for multi-threaded environments
Traversal	Uses simple loops or enhanced for-loop	Uses both `Iterator` and `Enumeration`
Memory Management	Static memory allocation	Dynamic memory allocation
Methods	Basic operations, manual handling required	Rich set of methods for manipulation

Inheritance:

Deriving new classes from existing classes such that the new classes acquire all the features(fields and methods) of existing classes is called inheritance.

The class whose features are inherited is known as a superclass(or a base class or a parent class).

The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). A subclass can reuse the methods and fields of the super class. The subclass can add its own fields and methods in addition to the superclass fields and methods.



In order to inherit a class we make use of the keyword extends.

syntax:

```
class subclass-name extends superclass-name
{
    body of the class
}
```

Advantages of Inheritance:

1. Inheritance helps in code reuse. The child class may use the code defined in the parent class without re-writing it.

2. Application development time is very less as the code size is reduced.

3. Redundancy of the code is reduced. Hence we can get constant results and less storage space.

Ex1:

```
class Lecturer
{
   //fields of parent class
   String designation;
   String collegeName;

   //method of parent class
   void getDetails()
   {
      System.out.println("designation:"+designation);
      System.out.println("collegeName:"+collegeName);
   }
}

class SoftSkillLecturer extends Lecturer
{
   //field of child class
   String mainSubject = "SoftSkill";

   SoftSkillLecturer(String designation, String collegeName)
   {
    this.designation=designation;
    this.collegeName=collegeName;
   }
}

class ComputerLecturer extends Lecturer
{
   //field of child class
```

```
    String mainSubject = "computer";

    ComputerLecturer(String designation,String collegeName)
    {
     this.designation=designation;
     this.collegeName=collegeName;
    }
}

public class InheritanceEx1
{
    public static void main(String args[])
    {
        ComputerLecturer ct = new ComputerLecturer("HoD","Malineni");

        System.out.println("computer teacher details");

        System.out.println("Main subject:"+ct.mainSubject);

        //accessing the method of parent class
        ct.getDetails();

        System.out.println("softskills teacher details");
        SoftSkillLecturer st = new SoftSkillLecturer("Mentor","Malineni2");
        System.out.println("Main subject:"+st.mainSubject);

        //accessing the method of parent class
        st.getDetails();

    }
}
```

Output:

```
computer teacher details
Main subject:computer
designation:HoD
collegeName:Malineni
softskills teacher details
Main subject:SoftSkill
designation:Mentor
collegeName:Malineni2
```

Ex2:

```
class Employee
```

```
{
   String eid;
   String ename;
   String eaddr;
   public void getEmpDetails()
   {
      System.out.println("Employee Id :"+eid);
      System.out.println("Employee name :"+ename);
      System.out.println("Employee Address :"+eaddr);
   }
}

class Manager extends Employee
{
    Manager(String eid1, String ename1, String eaddr1)
    {
       eid = eid1;
       ename = ename1;
       eaddr = eaddr1;
    }

    public void getManagerDetails()
    {
       System.out.println("manager Details");
       System.out.println("-----------------");
       getEmpDetails();
    }
}
class Accountant extends Employee
{
    Accountant (String eid1, String ename1, String eaddr1)
    {
       eid = eid1;
       ename = ename1;
       eaddr = eaddr1;
    }
    public void getAccountantDetails()
    {
       System.out.println("Accountant Details");
       System.out.println("--------------------");
       getEmpDetails();
    }
}
public class InheritanceEx
{
   public static void main(String[] args)
   {
      Manager m = new Manager("E-111","AAA","Hyd");
      m.getManagerDetails();
      System.out.println();
```

```
        Accountant acc = new Accountant ("E-222","BBB","Hyd");
        acc.getAccountantDetails();
    }
}
```

Output:

```
manager Details
----------------
Employee Id :E-111
Employee name :AAA
Employee Address :Hyd

Accountant Details
-------------------
Employee Id :E-222
Employee name :BBB
Employee Address :Hyd
```

Note:

A subclass inherits all the members (fields, methods, and nested classes) from its superclass. **Constructors are not members, so they are not inherited by subclasses**, but the constructor of the superclass can be invoked from the subclass

Note:

In the case of inheritance, when a subclass constructor is invoked, the JVM first executes the no-argument (0-arg) constructor of the superclass, and then it proceeds to execute the subclass constructor.

```
// Superclass
class SuperClass {
    // No-argument constructor of SuperClass
    SuperClass() {
        System.out.println("Superclass constructor executed.");
    }
}

// Subclass
class SubClass extends SuperClass {
    // Constructor of SubClass
    SubClass() {
        // Implicit call to super() happens here
        System.out.println("Subclass constructor executed.");
    }
}

// Main class to run the program
public class ConstructorDemo {
```

```
    public static void main(String[] args) {
        // Creating an instance of SubClass
        SubClass obj = new SubClass();
    }
}
```

Output:

```
Superclass constructor executed.
Subclass constructor executed.
```

super keyword:

The super keyword in Java is used to access the members of the **immediate super class** from the subclass.

It is mainly used

1) To refer super class variables

2) To refer super class methods.

3) To refer super class constructors

super keyword at variable level

When you have a variable in child class which is already present in the parent class then in order to access the variable of parent class, you need to use the super keyword.

To differentiate the super class variables with the sub class variables, in the subclass, the super class variables must be preceded by super keyword.

syntax:

```
super.variablename;
```

```
class Animal
{
    String color="white";
}
class Dog extends Animal
{
    String color="black";
    void printColor()
    {
        System.out.println(color);//prints color of Dog class
        System.out.println(super.color);//prints color of Animal class
    }
}
class TestSuper1
{
    public static void main(String args[])
```

```
    {
        Dog d=new Dog();
        d.printColor();
    }

}
```

Output:

```
black
white
```

In the example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

super keyword at method level

When inheriting methods from a superclass (base class) into a subclass (derived class), there may be instances where the methods in the superclass have the same name as those in the subclass. To distinguish between them, the superclass methods can be invoked in the subclass by using the super keyword.

Syntax:

```
super . method_Name([Param_List]);
```

```
class ParentClass
{


    public void display()
    {
        System.out.println("parent class method");
    }
}

class SubClass extends ParentClass
{
    public void display()
    {
        System.out.println("child class method");
    }

    public void printMsg()
    {
        display();
        super.display();//calling Parent class display method
    }
}
```

```
// Main class to run the program
public class Demo {
    public static void main(String[] args) {
        // Creating an instance of SubClass
        SubClass obj = new SubClass();
        obj.printMsg();
    }
}
```

Output:

```
child class method
nt class method
```

super keyword at constructor level

The super keyword can also be used to invoke the parent class constructor.

In the case of inheritance ,When a subclass constructor is invoked, the JVM first executes the no-argument (0-arg) constructor of the superclass, followed by the execution of the subclass constructor.

Whenever we want to call the no-arg constructor of super class from the sub class constructor, using super() is optional.

But, **Whenever we want to call the parameterized constructor of super class from the sub class constructor , using super(…) is mandatory.**

If we want to access super class constructor from subclass by using "super" keyword then the respective "super" statement must be provided as first statement and must be provided in the subclass constructors only, not in subclass normal Java methods

Ex1:

```
class BC
{
   BC()
   {
      System.out.println("Base Constructor");
   }
}

class DC extends BC
{
        int a,b;
        DC(int x,int y)
        {
                //super(); //optional
                a=x;
                b=y;
           System.out.println("Derived Constructor");
```

```
        }
}

public class SuperConDemo
{
   public static void main(String[] args)
   {
      DC obj = new DC(10,20);
   }
}
```

Output:

Base Constructor
Derived Constructor

Ex2:

```
class BC
{
        int m,n;
        BC(int x, int y)
        {
           m=x;
           n=y;
           System.out.println("Base Constructor");
        }
}
class DC extends BC
{
        int a,b;
        DC(int x,int y)
        {
           super(100,200); //mandatory
           a=x;
           b=y;
           System.out.println("Derived Constructor");
        }
}

public class SuperConDemo
{
   public static void main(String[] args)
   {
      DC obj = new DC(10,20);
   }
}
```

Output:

Base Constructor
Derived Constructor

Method Overriding:

Method overriding in Java occurs when a subclass provides a specific implementation of a method that already exists in its superclass. For overriding to happen, the method in the subclass must have the same name, the same parameter list (i.e., the same number, order, and types of parameters), and the same return type (or a covariant return type). This allows the subclass to customize or modify the behavior of the inherited method.

```
// Superclass
class Bank {
    int getRateOfInterest() {
        return 0;
    }
}

// Subclass
class SBI extends Bank {
    int getRateOfInterest() {
        return 8;
    }
}

class ICICI extends Bank {
    int getRateOfInterest() {
        return 7;
    }
}

class AXIS extends Bank {
    int getRateOfInterest() {
        return 9;
    }
}

public class Main {
    public static void main(String[] args) {
        Bank sbi = new SBI();
        Bank icici = new ICICI();
        Bank axis = new AXIS();

        System.out.println("SBI Rate of Interest: " + sbi.getRateOfInterest());
        System.out.println("ICICI Rate of Interest: " + icici.getRateOfInterest());
        System.out.println("AXIS Rate of Interest: " + axis.getRateOfInterest());
    }
}
```

Output:

```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Polymorphism:

Polymorphism is a key concept in object-oriented programming (OOP) that allows objects of different types to be treated as objects of a common super type. In Java, it allows one entity (like a method, variable, or object) to take multiple forms. Polymorphism simplifies code and enhances flexibility and maintainability.

Java primarily supports two types of polymorphism:

1. **Compile-time Polymorphism (Method Overloading)**
2. **Runtime Polymorphism (Method Overriding)**

---

1. Compile-time Polymorphism (Method Overloading)

Compile-time polymorphism is achieved through **method overloading**, which means having multiple methods with the same name but different parameter lists (type, number, or both) within the same class.

Method Overloading

- Method overloading occurs when two or more methods in the same class share the same name but differ in the method signature (number, type, or order of parameters).
- The return type may or may not be different.
- The compiler determines which method to invoke based on the method signature during compile time.

Example of Method Overloading:

```
class Calculator {
    // Overloaded method for adding two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method for adding three integers
    public int add(int a, int b, int c) {
```

```
        return a + b + c;
    }

    // Overloaded method for adding two double values
    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(10, 20));          // Calls the first method
        System.out.println(calc.add(10, 20, 30));      // Calls the second method
        System.out.println(calc.add(10.5, 20.5));      // Calls the third method
    }
}
```

```
30
60
31.0
```

2. Runtime Polymorphism (Method Overriding)

Runtime polymorphism is achieved through **method overriding**, where a subclass provides a specific implementation for a method that is already defined in its parent class. The method is overridden to perform a different task specific to the subclass, and the decision about which method to invoke is made at runtime.

Method Overriding

- Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.
- Both the name, return type, and parameter list of the overridden method must be the same.

- The method that gets invoked is determined at runtime based on the object that is being referred to.

Example of Method Overriding:

```
class Animal {
   // Parent class method
   public void sound() {
      System.out.println("Animal makes a sound");
   }
}

class Dog extends Animal {
   // Subclass overrides the sound() method
   public void sound() {
      System.out.println("Dog barks");
   }
}

class Cat extends Animal {
   // Subclass overrides the sound() method
      public void sound() {
      System.out.println("Cat meows");
   }
}

public class Main {
   public static void main(String[] args) {
      Animal a1 = new Animal();
      Dog d1 = new Dog();
      Cat c1 = new Cat();

      a1.sound();
      d1.sound();
     c1.sound();
   }
}
```

Output:

```
Animal makes a sound
Dog barks
Cat meows
```

Dynamic Method Dispatch:

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime rather than compile-time. It is a core feature of runtime polymorphism in Java and is used primarily in conjunction with method overriding.

Dynamic method dispatch requires a class hierarchy where a subclass overrides a method defined in a superclass.

A superclass reference variable can refer to a subclass object. At runtime, the Java Virtual Machine (JVM) determines which method to invoke based on the actual object type, not the reference type.

The JVM uses the actual object type to call the appropriate overridden method when a method is invoked.

```
class Animal {
    // Parent class method
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    // Subclass overrides the sound() method
    public void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    // Subclass overrides the sound() method
        public void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a1 = new Animal();   // Animal reference
        Animal a2 = new Dog();       // Animal reference, but Dog object
        Animal a3 = new Cat();       // Animal reference, but Cat object

        a1.sound();              // Calls Animal's sound method
        a2.sound();               // Calls Dog's sound method (overridden)
       a3.sound();               // Calls Cat's sound method (overridden)
    }
}
```

Output:

```
Animal makes a sound
```

```
Dog barks
Cat meows
```

Encapsulation:

Encapsulation is a fundamental concept in Java's Object-Oriented Programming (OOP) paradigm. It involves bundling the data (variables) and the methods (functions) that operate on the data into a single unit, typically a class. This concept helps in protecting the data from outside interference and misuse.

In Java, encapsulation is achieved by:

1. Declaring the variables of a class as private.
2. Providing public setter and getter methods to modify and view the variables' values.

```
// A Java class which is a fully encapsulated class.
class Student {
    // private data member
    private String name;

    // getter method for name
    public String getName() {
        return name;
    }

    // setter method for name
    public void setName(String name) {
        this.name = name;
    }
}

// A Java class to test the encapsulated class.
public class Test {
    public static void main(String[] args) {

        Student s = new Student();

       // s.name="pavan"; // we will get error as we cannot access private member  directly

        s.setName("Pavan");

        System.out.println(s.getName());
    }
}
```

Output:

Pavan

Advantages:

1. The Data is not accessible to the outside world and only the methods(which are wrapped) can access it and that **keeps data both safe from outside interference and misuse.**

2. Data encapsulation led to the important OOP concept of data hiding

Abstraction:

Abstraction is a key concept in Java's Object-Oriented Programming (OOP) paradigm. It involves hiding the complex implementation details of a system and exposing only the essential features to the user. This helps in reducing complexity and allows the programmer to focus on what an object does rather than how it does it.

In Java, abstraction can be achieved using:

1. **Abstract Classes**
2. **Interfaces**

Differences between Encapsulation and abstraction:

Aspect	Encapsulation	Abstraction
Definition	Bundling data and methods that operate on the data into a single unit.	Hiding the complex implementation details and showing only the essential features.
Purpose	To hide and protect the data from outside interference and misuse.	To reduce complexity by hiding unnecessary details from the user.
Implementation	Achieved using access modifiers (private, protected, public) and getter/setter methods.	Achieved using abstract classes and interfaces.
Focus	Focuses on how the object behaves.	Focuses on what the object does.
Security	Enhances security by restricting access to data.	Enhances security by exposing only necessary details.

final keyword:

The final keyword can be applied to classes, methods, and variables. It prevents modification in various contexts:

- **final Variables**: A variable marked as final cannot be modified once it has been initialized.
- **final Methods**: A method marked as final cannot be overridden by subclasses.
- **final Classes**: A class marked as final cannot be subclassed.

final methods:

- If you make any method as final, you cannot override it.
- When we don't want to redefine the definition of method into various derived classes then that method must be made it as final.
- Syntax:

  <final> <returntype> <method_name> ([formal paramters list])

  {

  }

final classes:

- The final class is a class that is declared with the final keyword.
- final class means that the class cannot be extended or inherited.
- If we try to inherit a final class, then the compiler throws an error during compilation.
- All wrapper classes in Java are final classes, such as String, Integer, etc.

final variables:

- A variable declared with the final keyword, whose value cannot be changed after initialization is called a final variable.

Access Control and Inheritance:

When dealing with inheritance, access control plays a crucial role in determining which members of the superclass are accessible to the subclass.

Public members of the superclass are fully accessible in the subclass, regardless of package boundaries. They can be inherited, accessed, and overridden freely by the subclass.

protected members can be accessed by the subclass even if it is in a different package. This provides a controlled level of access where the superclass allows subclasses to use or modify certain members without exposing them to the entire application.

Default members are accessible only if the subclass is in the same package. This means that a subclass in a different package will not be able to access or override them.

Private members are not inherited by the subclass and cannot be accessed directly. However, they can still influence subclass behavior through the use of public or protected methods in the superclass.

Universal Super Class – Object class:

In Java, the Object class is considered the **universal superclass** of all classes. Every class in Java, either directly or indirectly, inherits from the Object class. If a class doesn't explicitly extend any other class, it implicitly extends Object.

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

Here, A is super class to B and Object class is super class to A

Object <--- A <--- B

Methods of Object class:

The Object class defines several fundamental methods that every Java object inherits. Some of the most important ones include:

- public String toString():

  Returns a string representation of the object. By default, it returns the class name followed by the hash code, but it can be overridden to return a more meaningful string.

- protected void finalize():

  This method is called by the garbage collector before the object is destroyed. It allows an object to clean up resources before being collected.

- public boolean equals(Object obj):

  Compares this object to the specified object. By default, it compares object references (using ==), but it can be overridden to compare the content of objects.

- public int hashCode():

  Returns a hash code value for the object. It is typically used in conjunction with the equals method when working with data structures like HashMap or HashSet.

```
class Car {
    String model;

    Car(String model) {
        this.model = model;
    }

    public String toString() {
        return "Car model: " + model;
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Car car = (Car) obj;
        return model.equals(car.model);
    }

    public int hashCode() {
        return model.hashCode();
    }
}

public class Demo {
    public static void main(String[] args) {
        Car car1 = new Car("Tesla");
        Car car2 = new Car("Tesla");

        System.out.println(car1.toString());        // Output: Car model: Tesla
        System.out.println(car1.equals(car2));      // Output: true (based on content)
        System.out.println(car1.hashCode() == car2.hashCode());  // Output: true
        System.out.println(car1.hashCode());
    }
}
```

Output:

```
Car model: Tesla
true
true
80698615
```

concrete methods vs abstract methods:

Concrete method is a method which will have both method declaration and method implementation.

To declare concrete methods, no need to use any special keyword

```
void add(int i, int j)
 {
     int k=i+j;
     System.out.println(k);
 }
```

An **abstract method** in Java is a method that is declared without an implementation. It is declared using the abstract keyword.

abstract methods are allowed in abstract classes and interfaces.

Subclasses of the abstract class must provide the implementation of the abstract method, or they must also be declared abstract.

Ex:

abstract void add(int i, int j);

abstract class:

An abstract class in Java is a class that is declared with the abstract keyword. abstract classes allow zero or more number of concrete methods and zero or more number of abstract methods.

For abstract classes, we are able to create only reference variables;
we are unable to create objects.

```
abstract class A { ---- }
A a = new A();   --> Error
A a = null; --> No Error
```

Note:

If a class contains at least one abstract method, the class must be declared as abstract

```
// Abstract class with an abstract method calculate
abstract class Calculation {
   abstract void calculate(double x);
}

// Subclass 1: Prints square of x
class Square extends Calculation {
```

```
    void calculate(double x) {
        System.out.println("Square of " + x + " is: " + (x * x));
    }
}

// Subclass 2: Prints square root of x
class SquareRoot extends Calculation {

    void calculate(double x) {
        System.out.println("Square root of " + x + " is: " + Math.sqrt(x));
    }
}

// Subclass 3: Prints cube of x
class Cube extends Calculation {

    void calculate(double x) {
        System.out.println("Cube of " + x + " is: " + (x * x * x));
    }
}

// Main class to test the program
public class Main {
    public static void main(String[] args) {
        double value = 8.0;

        // Creating objects for each subclass
        Calculation square = new Square();
        Calculation squareRoot = new SquareRoot();
        Calculation cube = new Cube();

        // Calling calculate method for each subclass
        square.calculate(value);       // Prints square of value
        squareRoot.calculate(value);   // Prints square root of value
        cube.calculate(value);         // Prints cube of value
    }
}
```

Output:

```
Square of 8.0 is: 64.0
Square root of 8.0 is: 2.8284271247461903
Cube of 8.0 is: 512.0
```

Interface:

The interface in Java is *a* mechanism to achieve abstraction. In Java, an interface is used to specify a set of abstract methods that a class must implement. Interfaces are declared using the interface keyword and can contain abstract methods, default methods, static methods, and constants.

In Java applications, for interfaces, we are able to create only reference variables, we are unable to create objects.

Declaring an Interface

An interface in Java is declared using the interface keyword. It can contain abstract methods (methods without a body), default methods (methods with a default implementation), and static methods

Ex:

```
// Define the Payment interface
interface Payment
{
   void makePayment(double amount);
}
```

Note1:

In the case of interfaces, by default, all the variables are "public static final" and must be initialized at the time of declaration otherwise compiler will throw an error.

```
//program to illustrate use of variables in interfaces
interface SampleInterface
{
        int UPPER_LIMIT = 100;
        //int LOWER_LIMIT; // Error - must be initialized
}
public class Test3 implements SampleInterface{
      public static void main(String[] args) {
                System.out.println("UPPER LIMIT = " + UPPER_LIMIT);
                // UPPER_LIMIT = 150; // Can not be modified
```

```
    }
}
```

Note2:

In Java applications, constructors are possible in classes and abstract classes but constructors are not possible in interfaces.

Implementing an Interface:

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. **When a class implements an interface, it must provide concrete implementations for all the abstract methods declared in the interface.** For this, We use the implements keyword to implement an interface.

Ex:

```
// Define the Payment interface
interface Payment {
    void makePayment(double amount);
}

// Implement the Payment interface for CreditCard
class CreditCard implements Payment {
    private String cardNumber;

    public CreditCard(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    public void makePayment(double amount) {
        System.out.println("Paid " + amount + " using Credit Card: " +
cardNumber);
    }
}

// Implement the Payment interface for PhonePe
class PhonePe implements Payment {
```

```
    private long phoneNumber;

    public PhonePe(long phoneNumber) {
        this.phoneNumber = phoneNumber;
    }


    public void makePayment(double amount) {
        System.out.println("Paid " + amount + " using PhonePe phone number: "
+ phoneNumber);
    }
}

// Main class to test the payment methods
public class PaymentSystem {
    public static void main(String[] args) {
        Payment payment1 = new CreditCard("1234-5678-9876-5432");
        payment1.makePayment(250.75);

        Payment payment2 = new PhonePe(9999999999L);
        payment2.makePayment(150.50);
    }
}
```

Output:

```
Paid 250.75 using Credit Card: 1234-5678-9876-5432
Paid 150.5 using PhonePe phone number: 9999999999
```

Multiple Interfaces:

In Java, a class can implement multiple interfaces. This means a class can provide implementations for abstract methods of multiple interfaces them. Here's how you can do it:

```
interface A {
  // members of A
}


interface B {
  // members of B
}
```

```
class C implements A, B {
  // abstract members of A
  // abstract members of B
}
```

Note:

multiple inheritance in java can be achieved through interfaces

Ex:

```
// Flyable interface
interface Flyable {
 void fly();
}

// Swimmable interface
interface Swimmable {
 void swim();
}

// Walkable interface
interface Walkable {
 void walk();
}

// Class Duck implementing all three interfaces
class Duck implements Flyable, Swimmable, Walkable {

 public void fly() {
 System.out.println("Duck is flying...");
 }

 public void swim() {
 System.out.println("Duck is swimming...");
 }

 public void walk() {
 System.out.println("Duck is walking...");
 }
}
// Main class to test the implementation
```

```
public class Main {
 public static void main(String[] args) {
 Duck duck = new Duck();
 duck.fly(); // Calls fly method from Flyable interface
 duck.swim(); // Calls swim method from Swimmable interface
 duck.walk(); // Calls walk method from Walkable interface
 }
}
```

```
Duck is flying...
Duck is swimming...
Duck is walking...
```

Ex:

```
interface CreditCardPayment {
    void processCreditCardPayment(double amount);
}

interface DigitalWalletPayment {
    void processDigitalWalletPayment(double amount);
}


public class PaymentProcessor implements CreditCardPayment,
DigitalWalletPayment {
    @Override
    public void processCreditCardPayment(double amount) {
        System.out.println("Processing credit card payment of $" + amount);
        // Add logic to process credit card payment
    }

    @Override
    public void processDigitalWalletPayment(double amount) {
        System.out.println("Processing digital wallet payment of $" + amount);
        // Add logic to process digital wallet payment
    }

    public static void main(String[] args) {
        PaymentProcessor processor = new PaymentProcessor();
        processor.processCreditCardPayment(100.0);
        processor.processDigitalWalletPayment(50.0);
    }
}
```

```
Output:

Processing credit card payment of $100.0
Processing digital wallet payment of $50.0
```

Inheritance of Interfaces:

Inheritance of interfaces in Java allows one interface to inherit or extend another interface.The extends keyword is used for extending interfaces.

```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Here, the *Polygon* interface extends the *Line* interface. Now, if any class implements *Polygon*, it should provide implementations for all the abstract methods of both *Line* and *Polygon*.

Default methods:

A default method in Java is a method defined within an interface using the default keyword, complete with its own implementation.(includes a method body)

Default methods in interfaces allow you to add new methods to interfaces with a default implementation without affecting the classes that implement the interface.

This means that implementing classes are not required to override these methods unless they need a specific behavior. This ensures backward compatibility, as existing classes that implement the interface will continue to function correctly even after new methods are added.

Ex:

```
interface MyInterface {
    default void display() {
        System.out.println("Default implementation in the interface.");
    }
}
```

```
interface Vehicle {
    void startEngine();
    void stopEngine();

    // Default method
    default void turnAlarmOn() {
        System.out.println("Turning the vehicle alarm on.");
    }

    // Another default method
    default void turnAlarmOff() {
        System.out.println("Turning the vehicle alarm off.");
    }
}

class Car implements Vehicle {
    @Override
    public void startEngine() {
        System.out.println("Car engine started.");
    }

    @Override
    public void stopEngine() {
        System.out.println("Car engine stopped.");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.startEngine();
        myCar.turnAlarmOn();  // Using the default method
        myCar.stopEngine();
        myCar.turnAlarmOff(); // Using the default method
    }
}
```

Output:

```
Car engine started.
Turning the vehicle alarm on.
Car engine stopped.
Turning the vehicle alarm off.
```

static methods:

static methods in interfaces were introduced in Java 8, providing a way to include utility methods (or) helper methods directly within the interface.

static methods in interfaces belong to the interface itself rather than to instances of the classes that implement the interface. This means they are called using the interface name.

static methods in interfaces cannot be overridden by implementing classes. This ensures that the utility methods remain consistent and cannot be altered by subclasses.

```
interface MathOperations {
    // Abstract method
    int add(int a, int b);

    // Static method
    static int multiply(int a, int b) {
        return a * b;
    }

    // Another static method
    static int subtract(int a, int b) {
        return a - b;
    }
}

class Calculator implements MathOperations {
    @Override
    public int add(int a, int b) {
        return a + b;
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Addition: " + calc.add(5, 3));

        // Using static methods from the interface
        System.out.println("Multiplication: " + MathOperations.multiply(5, 3));
        System.out.println("Subtraction: " + MathOperations.subtract(5, 3));
    }
}
```

Output:

Addition: 8
Multiplication: 15
Subtraction: 2

Functional interface:

An interface having a single abstract method is called a Functional Interface or Single Abstract Method Interface. Lambda expression is used to provide the implementation of such a functional interface. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code which saves a lot of code.

Functional interfaces can also contain any number of default and static methods.

```
@FunctionalInterface
interface MyFunctionalInterface {

    public int addMethod(int a, int b);
}
public class BeginnersBookClass {

   public static void main(String args[]) {
        // lambda expression
          MyFunctionalInterface sum = (a, b) -> a + b;
        System.out.println("Result: "+sum.addMethod(12, 100));
    }
}
```

Output:

```
Result: 112
```

Annotations:

Annotations are a powerful tool in Java for providing metadata about your code, and they play a significant role in frameworks like Spring, Hibernate, and Junit

They do not change the action of the compiled program but can be used to give instructions to the compiler, provide runtime instructions, and be used by frameworks or libraries to process code in a specific way.

Basic Syntax

Annotations are specified using the @ symbol followed by the annotation name. They can be applied to various elements like classes, methods, fields, parameters, etc.

```
@AnnotationName
public class MyClass {
    // Class content
}
```

Commonly used Annotations:

@Override

Indicates that a method is intended to override a method in a superclass. This helps prevent errors, such as mismatches in method signatures.

@SuppressWarnings

Instructs the compiler to suppress specific warnings, like unchecked or deprecation warnings.

@FunctionalInterface

Indicates that an interface is intended to be a functional interface, which means it has exactly one abstract method.