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

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

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

Arrays

- An array is a group of continuous or related items that share a common name.
- For instance, we can define an array name salary to represent a set of salaries of a group of employees.
- A particular value is indicated by writing a number called index number or subscript in brackets after the array name.

One –Dimensional Arrays

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one-dimensional array.

Declaration of Array :

Arrays in java may be declared in two forms

Form1

type arrayname[ ];

Form2

type[ ] arrayname;

Creating Arrays :

you can create an array by using the new operator by using syntax

Syntax:

arrayname=new type[array_Size];

It creates an array using new type[array_Size]

It assigns the reference of the newly created array to the variable arrayname.

Declaring , Creating and assigning an array to the variable can be combined in one statement as:

type[]=arrayname=new type[array_Size];

(or)

type[] arrayname={value0, value 1,.....value k};

Array indices are start from 0 to arrayname. length-1

Two Dimensional Array :

It is used to store two dimensional data. It is also used to store data, which contains rows and columns.

If the data is linear we can use one dimensional array to work with multi-level data we have to use Multi-Dimensional Array.

Creating Two Dimensional Array :

Data_Type[][] Array_Name=new int[Row_Size][Column_Size];

Initialization of Two Dimensional Array :

We can initialize the Two Dimensional Array in some ways

Example :

int[][] Student_Marks = new int[2][3];

int[][] Employees = { {10,20,30}, {15,25,35}, {22,44,66}, {33,55,77} };

Accessing Elements of Arrays

Accessing Elements of a One-Dimensional Array

```
Class ArrayExample
{
   public static void main(String[] args)
          {
              // Declare and initialize a one-dimensional array
              Int data [ ] = {5, 10, 15, 20, 25};

      // Access and print individual elements
      System.out.println("First element: " + data[0]);  // Output: 5
      System.out.println("Second element: " + data[1]); // Output: 10
      System.out.println("Third element: " + data[2]);  // Output: 15
      System.out.println("Fourth element: " + data[3]); // Output: 20
      System.out.println("Fifth element: " + data[4]);  // Output: 25
   }
}
```

Accessing Elements of a two-Dimensional Array

```
class Access2DArray
{
  public static void main(String[] args)
 {
    // Declare and initialize a two-dimensional array
     Int matrix[ ] [ ] = {
                                {1, 2, 3},
                                {4, 5, 6},
                                {7, 8, 9}
                            };

    // Access and print individual elements
    System.out.println("Element at row 0, column 0: " + matrix[0][0]); // Output: 1
    System.out.println("Element at row 0, column 1: " + matrix[0][1]); // Output: 2
    System.out.println("Element at row 1, column 2: " + matrix[1][2]); // Output: 6
    System.out.println("Element at row 2, column 1: " + matrix[2][1]); // Output: 8
    System.out.println("Element at row 2, column 2: " + matrix[2][2]); // Output: 9
```

Storage of Array in Computer Memory

In computer memory, arrays are stored in a contiguous block of memory. The array elements are stored sequentially in memory, meaning that each element is placed directly after the previous one. This arrangement allows for efficient access to any element in the array using an index, making arrays a popular data structure in programming languages like Java.

How Arrays Are Stored in Memory:

1. **Contiguous Memory Allocation**:
   - Arrays are stored in a **continuous block of memory**. The size of this memory block is calculated based on the data type of the array and the number of elements.
   - If the array is an integer array, for example, each element will take up 4 bytes of memory (assuming a 32-bit integer), and the total memory size will be 4 * n, where n is the number of elements.
2. **Indexing**:
   - Elements in an array are accessed using an index. The index is used to calculate the memory address of the element.
   - For example, in a one-dimensional array, the memory address of the element at index i is calculated as:

     Address_of_element(i) = Base_address + (i * size_of_element)

where Base_address is the memory address of the first element in the array, i is the index, and size_of_element is the size of each array element in bytes.

3. **Memory Layout Example**:
   - Consider the following integer array:

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

   - If the array is stored starting at memory address 1000, the elements are laid out in memory as:

     Address Value

     1000 10 (arr[0])

     1004 20 (arr[1])

     1008 30 (arr[2])

     1012 40 (arr[3])

     1016 50 (arr[4])

   - Here, each element takes 4 bytes (since it's an int), and the elements are stored consecutively.

Multi-Dimensional Arrays:

- In the case of multi-dimensional arrays (e.g., 2D arrays), the elements are stored in **row-major order** in Java. This means that the elements of each row are stored sequentially in memory.
- Consider a 2D array:

```
int[][] matrix = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9}
};
```

In memory, this array would be laid out as:

Address Value

1000 1 (matrix[0][0])

1004 2 (matrix[0][1])

1008 3 (matrix[0][2])

1012 4 (matrix[1][0])

1016 5 (matrix[1][1])

1020 6 (matrix[1][2])

1024 7 (matrix[2][0])

1028 8 (matrix[2][1])

1032 9 (matrix[2][2])

- row are stored first, followed by the elements of the second row, and so on.

Types of Arrays and Memory Allocation:

1. **Primitive Arrays**:
   - Arrays that store primitive types like int, char, float, etc., store the actual values in contiguous memory.
   - Example:

     ```
     int[] arr = {1, 2, 3};
     ```

     Each int value (4 bytes) is stored contiguously in memory.
2. **Object Arrays**:
   - Arrays that store object references (e.g., arrays of String or user-defined objects) do not store the actual objects in contiguous memory.
   - Instead, the array stores references (memory addresses) to the objects, which may be located anywhere in memory.
   - Example:

     ```
     String[] arr = {"Apple", "Banana", "Cherry"};
     ```

     The array arr contains references to String objects, and those strings are stored at different memory locations.

Advantages of Contiguous Memory Storage:

1. **Efficient Indexing**:
   - Since arrays are stored in contiguous memory, the memory address of any element can be calculated quickly using the index. This makes accessing elements very fast (O(1) time complexity).
2. **Cache-Friendly**:
   - Contiguous memory storage takes advantage of CPU caching. When an element of an array is accessed, nearby elements are likely loaded into the cache, speeding up future access.

Disadvantages:

1. **Fixed Size**:
   - array size is too large or too small.
2. **Inefficient Insertion/Deletion**:
   - Inserting or deleting elements in the middle of an array requires shifting elements, which can be slow (O(n) time complexity).

Operations on Array Elements

In Java, you can perform various operations on array elements, such as arithmetic operations, traversals, modifications, and more. Below are some examples of common operations performed on array elements.

Sum of All Elements in an Array

```
 class ArrayOperations
{
   public static void main(String[] args)
{
      int numbers[ ] = {10, 20, 30, 40, 50};
      int sum = 0;
      // Loop through the array to calculate the sum of elements
      for (int i = 0; i < numbers.length; i++)
      {
            sum = sum+ numbers[i];
      }

      System.out.println("Sum of all elements: " + sum);
   }
}
```

Finding the Maximum Element in an Array

```
class ArrayOperations
{
   public static void main(String[] args)
{
      Int      numbers[ ] = {10, 20, 30, 40, 50};
      int max = a[i];

      // Loop through the array to find the maximum element
      for (int i = 1; i < numbers.length; i++)
{
                        if (numbers[i] > max)
 {
                        max = numbers[i];
                         }
```

```
        System.out.println("Maximum element: " + max);
    }
}
```

Finding the Minimum Element in an Array

```
class ArrayOperations {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};
        int min = numbers[0];

        // Loop through the array to find the minimum element
        for (int i = 1; i < numbers.length; i++) {
            if (numbers[i] < min) {
                min = numbers[i];
            }
        }

        System.out.println("Minimum element: " + min);
    }
}
```

Get the First and Last Element of an Array

To get the first and last elements of an array, you need to access the elements at index 0 (for the first element) and index array.length - 1 (for the last element). Here are examples in different programming languages:

```
public class ArrayFirstLastElement {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};

        // Get the first element
        int firstElement = arr[0];

        // Get the last element
        int lastElement = arr[arr.length - 1];
```

```
        System.out.println("First element: " + firstElement);
        System.out.println("Last element: " + lastElement);
    }
}
```

Output:

First element: 10

Last element: 50

To compare two arrays in Java, you need to determine if they are equal in terms of their content and order. You can use the Arrays class from the java.util package, which provides utility methods for comparing arrays.

Here's how you can compare two arrays:

Using Arrays.equals()

The Arrays.equals() method checks if two arrays are equal by comparing their length and corresponding elements.

Example Code

```
import java.util.Arrays;

public class CompareArrays {
    public static void main(String[] args) {
        int[] array1 = {1, 2, 3, 4, 5};
        int[] array2 = {1, 2, 3, 4, 5};
        int[] array3 = {1, 2, 3, 4, 6};

        // Compare array1 and array2
        boolean areEqual1 = Arrays.equals(array1, array2);
        System.out.println("array1 and array2 are equal: " + areEqual1);

        // Compare array1 and array3
        boolean areEqual2 = Arrays.equals(array1, array3);
        System.out.println("array1 and array3 are equal: " + areEqual2);
    }
}
```

Assigning One Array To Another Array

```
public class CopyArray {
    public static void main(String[] args) {
        // Initialize the original array
        int[] arr1 = new int[] {1, 2, 3, 4, 5};

        // Create another array arr2 with the same size as arr1
        int[] arr2 = new int[arr1.length];

        // Copy all elements from arr1 to arr2
        for (int i = 0; i < arr1.length; i++) {
            arr2[i] = arr1[i];
        }

        // Displaying elements of the original array
        System.out.println("Elements of the original array: ");
        for (int i = 0; i < arr1.length; i++) {
            System.out.print(arr1[i] + " ");
        }

        System.out.println();

        // Displaying elements of the new array
        System.out.println("Elements of the new array: ");
        for (int i = 0; i < arr2.length; i++) {
            System.out.print(arr2[i] + " ");
        }
    }
}
```

Dynamic change of arrays

In Java, arrays have a fixed size once they are created. If you need a dynamically sized collection, you'll want to use ArrayList from the java.util package, which provides dynamic resizing capabilities. Here's how you can use ArrayList:

Using ArrayList in Java

1. **Import ArrayList**: Make sure to import the ArrayList class:

   ```
   import java.util.ArrayList;
   ```

2. **Create an ArrayList**: You can create an ArrayList and use it similarly to an array, but with dynamic resizing:

```
public class Main {
   public static void main(String[] args) {
      // Create an ArrayList of integers
      ArrayList<Integer> myList = new ArrayList<>();

      // Add elements
      myList.add(1);
      myList.add(2);
      myList.add(3);

      // Remove an element
      myList.remove(Integer.valueOf(2)); // Removes the element with value 2

      // Print the elements
      for (int num : myList) {
         System.out.print(num + " ");  // Output: 1 3
      }
   }
}
```

Common Operations with ArrayList:

Adding Elements:

```
myList.add(4);  // Adds 4 to the end of the list
```

```
myList.add(1, 5);  // Adds 5 at index 1
```

Removing Elements:

```
myList.remove(2);  // Removes the element at index 2
myList.remove(Integer.valueOf(3));  // Removes the first occurrence of the value 3
```

Accessing Elements:

```
int element = myList.get(0);  // Gets the element at index 0
```

-

Iterating Over Elements:

```
for (int i = 0; i < myList.size(); i++) {
 System.out.println(myList.get(i));
}
```

Getting Size:

```
int size = myList.size();  // Gets the number of elements in the list
```

Clearing All Elements:

```
myList.clear();  // Removes all elements from the list
```

ArrayList is a versatile and commonly used collection in Java for managing dynamic-sized list.

Arrays Sorting

Array sorting refers to the process of arranging the elements of an array in a specific order, typically in ascending or descending order. In Java, there are several ways to sort arrays, including using built-in methods or implementing custom sorting algorithmS

```
public class SortArrayExample2
{
    public static void main(String[] args)
    {
    //creating an instance of an array
    int[] arr = new int[] {4,2,3,1};
    System.out.println("Array elements after sorting:");
    //sorting logic
    for (int i = 0; i < arr.length; i++)
    {
    for (int j = i + 1; j < arr.length; j++)
    {
```

```
        int tmp = 0;
        if (arr[i] > arr[j])
        {
        tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
        }
        }
        //prints the sorted element of the array
        System.out.println(arr[i]);
        }
        }
}
```

Descending Order

```
public class SortArrayExample2
{
        public static void main(String[] args)
        {
        //creating an instance of an array
        int[] arr = new int[] {78, 34, 1, 3, 90, 34, -1, -4, 6, 55, 20, -65};
        System.out.println("Array elements after sorting:");
        //sorting logic
        for (int i = 0; i < arr.length; i++)
        {
        for (int j = i + 1; j < arr.length; j++)
        {
        int tmp = 0;
        if (arr[i] <  arr[j])
        {
```

```
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            }
            }
            //prints the sorted element of the array
            System.out.println(arr[i]);
            }
            }
}
```

Search for Values in Arrays

To search for values in arrays in Java, you can use various methods depending on the type of search you want to perform. Below are examples of two common types of searches: **linear search** and **binary search**.

Linear Search

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the first element in the list.
- **Step 3 -** If both are matched, then display "Given element is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then compare search element with the next element in the list.
- **Step 5 -** Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6 -** If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

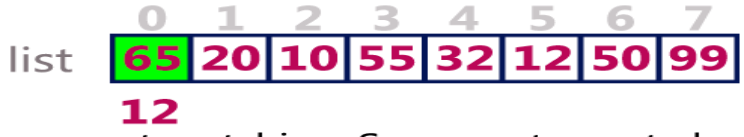
```
Class LinearSearch
{
 Public static void main (String args[])
{
```

```
    Int a[]={10,20,40,50,30};
    Int search_ele=50;
    Boolean flag=false;
    For(int i=0;i<a.length;i++)
    {
        If(search_ele==a[i])
        {
            System.out.println("the element is found at :+i);
            flag=true;
            break;
        }
    }
    If(flag==false)
    {
        System.out.println("element is not found");
        }
}
```

search element 12

Step 1:

search element (12) is compared with first element (65)



Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

Both are not matching. So move to next element

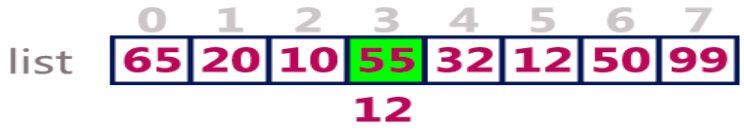
Step 3:

search element (12) is compared with next element (10)

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)



Both are not matching. So move to next element

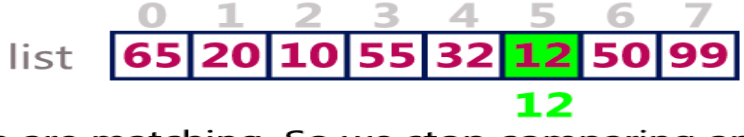
Step 5:

search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)



Both are matching. So we stop comparing and display element found at index 5.

Binary search

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Find the middle element in the sorted list.
- **Step 3 -** Compare the search element with the middle element in the sorted list.
- **Step 4 -** If both are matched, then display "Given element is found!!!" and terminate the function.
- **Step 5 -** If both are not matched, then check whether the search element is smaller or larger than the middle element.
- **Step 6 -** If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7 -** If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8 -** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9 -** If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

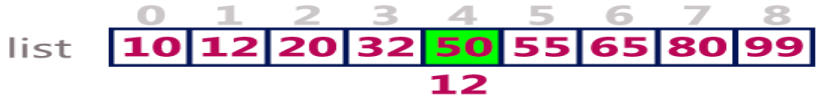
```
public class BinarySearch {

   public static void main(String[] args) {

      int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  // Should be in sorted order

      boolean flag = false;

      int key = 5;

      int l = 0;
      int h = a.length - 1;

      while (l <= h)
      {
         int m = (l + h) / 2;

         if (a[m] == key) {
            System.out.println("Element Found..");
            flag = true;
            break;
         }
```

```
            if (a[m] < key) {
                l = m + 1;
            }

            if (a[m] > key) {
                h = m - 1;
            }
        }

        if (flag == false) {
            System.out.println("Element NOT found..");
        }
    }
}
```

search element **12**

Step 1:

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

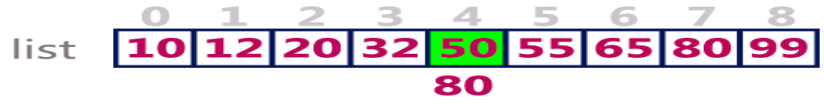
search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

search element **80**

Step 1:

search element (80) is compared with middle element (50)



Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

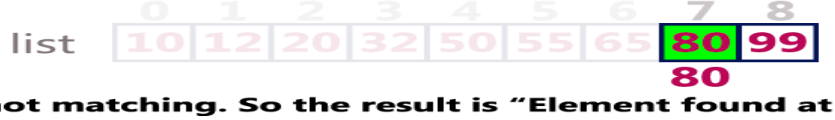
Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

search element (80) is compared with middle element (80)



Both are not matching. So the result is "Element found at index 7"

Arrays as Vectors (Vector Class in Java)

The Vector class in Java implements a dynamic array where elements can be added or removed. It is synchronized, which means it's thread-safe for use in multi-threaded applications. However, because of the synchronization overhead, it's generally slower than ArrayList.

Key Features of Vector:

- Dynamic resizing
- Can hold any type of data
- Supports operations like insertion, deletion, and searching
- Synchronization makes it thread-safe

Declaring and Using a Vector in Java:

```
import java.util.Vector;


public class VectorExample {
   public static void main(String[] args) {
      // Create a Vector to hold integer values
      Vector<Integer> vector = new Vector<>();

      // Adding elements to the Vector
      vector.add(10);
      vector.add(20);
      vector.add(30);
      vector.add(40);
      vector.add(50);

      // Accessing elements using an index
      System.out.println("Element at index 2: " + vector.get(2));  // Output: 30

      // Removing an element at a specific index
      vector.remove(3);  // Removes the element at index 3 (40)

      // Iterating over the elements
      System.out.println("Vector elements after removal:");
      for (int i = 0; i < vector.size(); i++) {
```

```
            System.out.println("Element at index " + i + ": " + vector.get(i));
        }

        // Size of the vector
        System.out.println("Size of the vector: " + vector.size());

        // Checking if the vector contains a specific element
        if (vector.contains(30)) {
            System.out.println("Vector contains 30");
        } else {
            System.out.println("Vector does not contain 30");
        }
    }
}
```

Output:

```
Element at index 2: 30
Vector elements after removal:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 50
Size of the vector: 4
Vector contains 30
```

Arrays Of Varying Lengths

In Java, you can create arrays of varying lengths, also known as **jagged arrays** or **ragged arrays**. A jagged array is an array whose elements are arrays of different lengths, unlike a regular multidimensional array where all rows have the same number of elements.

Declaring and Using Jagged Arrays

When you declare a 2D array, you don't have to specify the size of each row. Instead, you can assign arrays of varying lengths to each row.

Example Program: Arrays of Varying Lengths (Jagged Arrays)

java

Copy code

```
public class JaggedArrayExample {
   public static void main(String[] args) {
      // Declaring a 2D array with 3 rows
      int[][] jaggedArray = new int[3][];

      // Initializing each row with a different number of columns
      jaggedArray[0] = new int[3];  // First row has 3 elements
      jaggedArray[1] = new int[2];  // Second row has 2 elements
      jaggedArray[2] = new int[4];  // Third row has 4 elements

      // Populating the jagged array with values
      int value = 1;
      for (int i = 0; i < jaggedArray.length; i++)
         {
                  for (int j = 0; j < jaggedArray[i].length; j++)
         {
                   jaggedArray[i][j] = value++;
                   }
      }

      // Printing the elements of the jagged array
      System.out.println("Jagged Array Elements:");
      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();  // Move to the next line after each row
    }
  }
}
```

Output:

Jagged Array Elements:

1 2 3

4 5

6 7 8 9

INHERITANCE

- **The mechanism of deriving a new class from an old class such that the new class acquires all the properties of the old class is called Inheritance.**
- The old class is known as Parent, base or Super class and the new class that is derived is known as child, derived or subclass.
- The Inheritance allows subclasses to inherit all the variables and methods of their parent classes.

Defining a Subclass

- A Subclass is defined as follows

Class subclassname extends superclassname

{

Variables declaration

Methods declaration

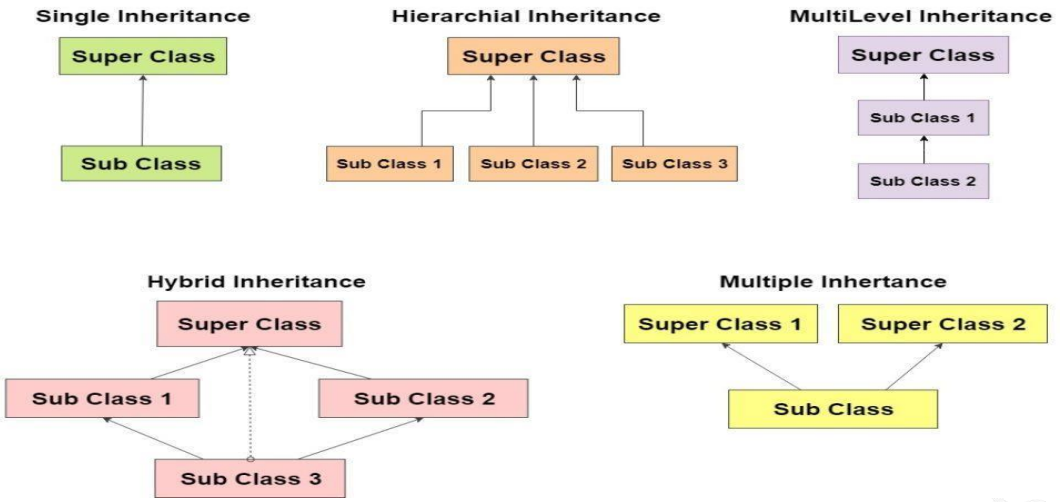
}

- The keyword extends signifies that the properties of the **superclassname** are extended **subclassname.**
- The subclass will now contain its own variables and methods as well those superclass.
- This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it.

Inheritance may take different types

1. Single inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Hybrid Inheritance
5. Multiple Inheritance (**Does not supports in java**)

These forms of inheritance are shown as



1. Single inheritance

The process of Creating one Child class from one Parent class is called single inheritance.

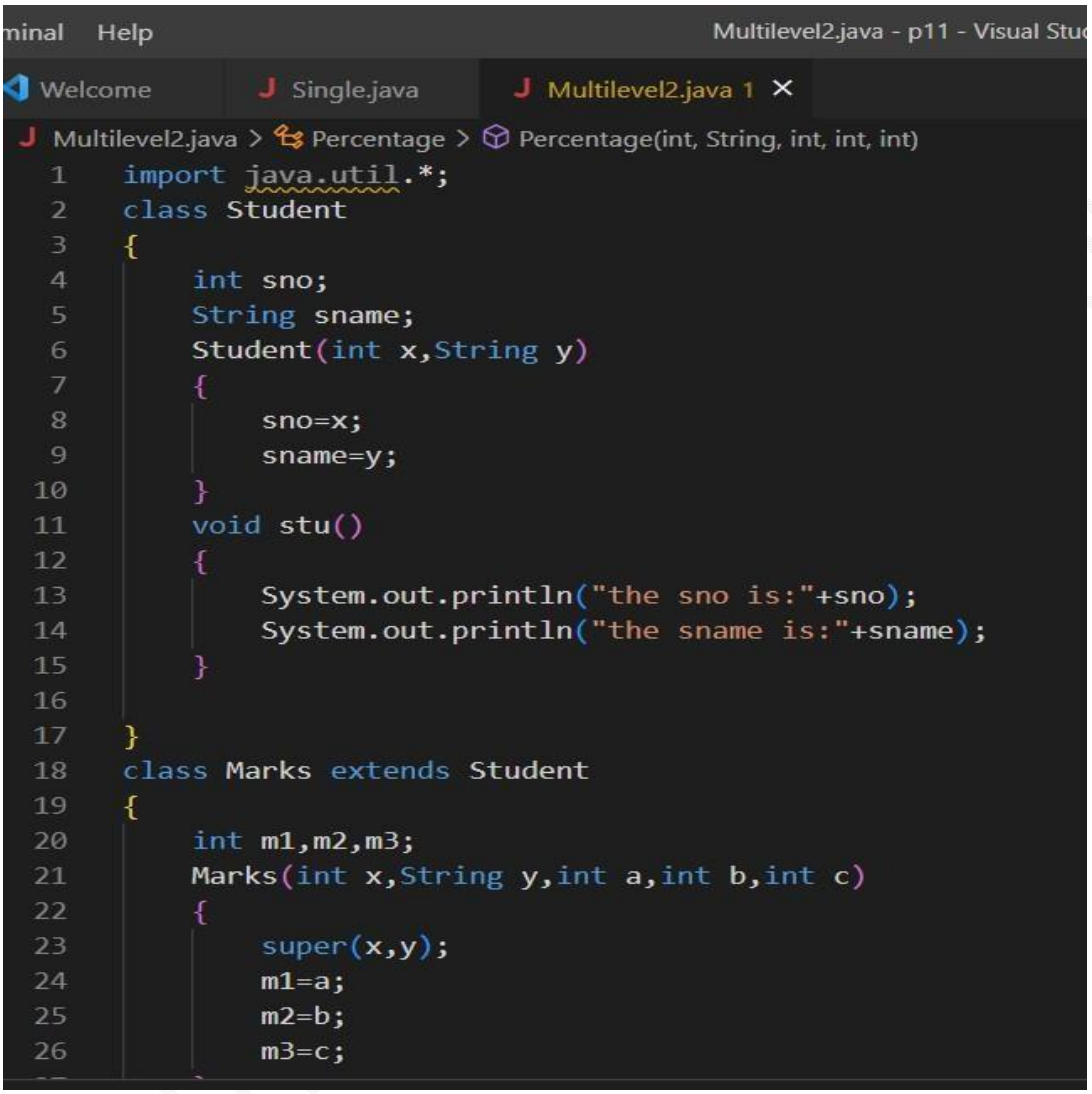
Example:

Single.java > Triangle > tarea()

```
1   class Rectangle
2   {
3       int length,width;
4       Rectangle(int x,int y)
5       {
6           length=x;
7           width=y;
8       }
9       int area()
10      {
11          int res=length*width;
12          return res;
13      }
14  }
15  class Triangle extends Rectangle
16  {
17      int height;
18      Triangle(int x,int y,int z)
19      {
20          super(x,y);
21          height=z;
22      }
23      int tarea()
24      {
25          int res1=area()*height;
26          return res1;
27      }
28  }
29  class Single
30  {
        Run | Debug
31      public static void main(String args[])
32      {
33      Triangle obj=new Triangle(x: 10,y: 5,z: 20);
34      int ra=obj.area();
35      int ta=obj.tarea();
36      System.out.println("the area of rectangle is:"+ra);
37      System.out.println("the area of triangle is:"+ta);
38      }
39  }
```

2. Multilevel Inheritance

Process of deriving a class from another derived class is called multilevel inheritance



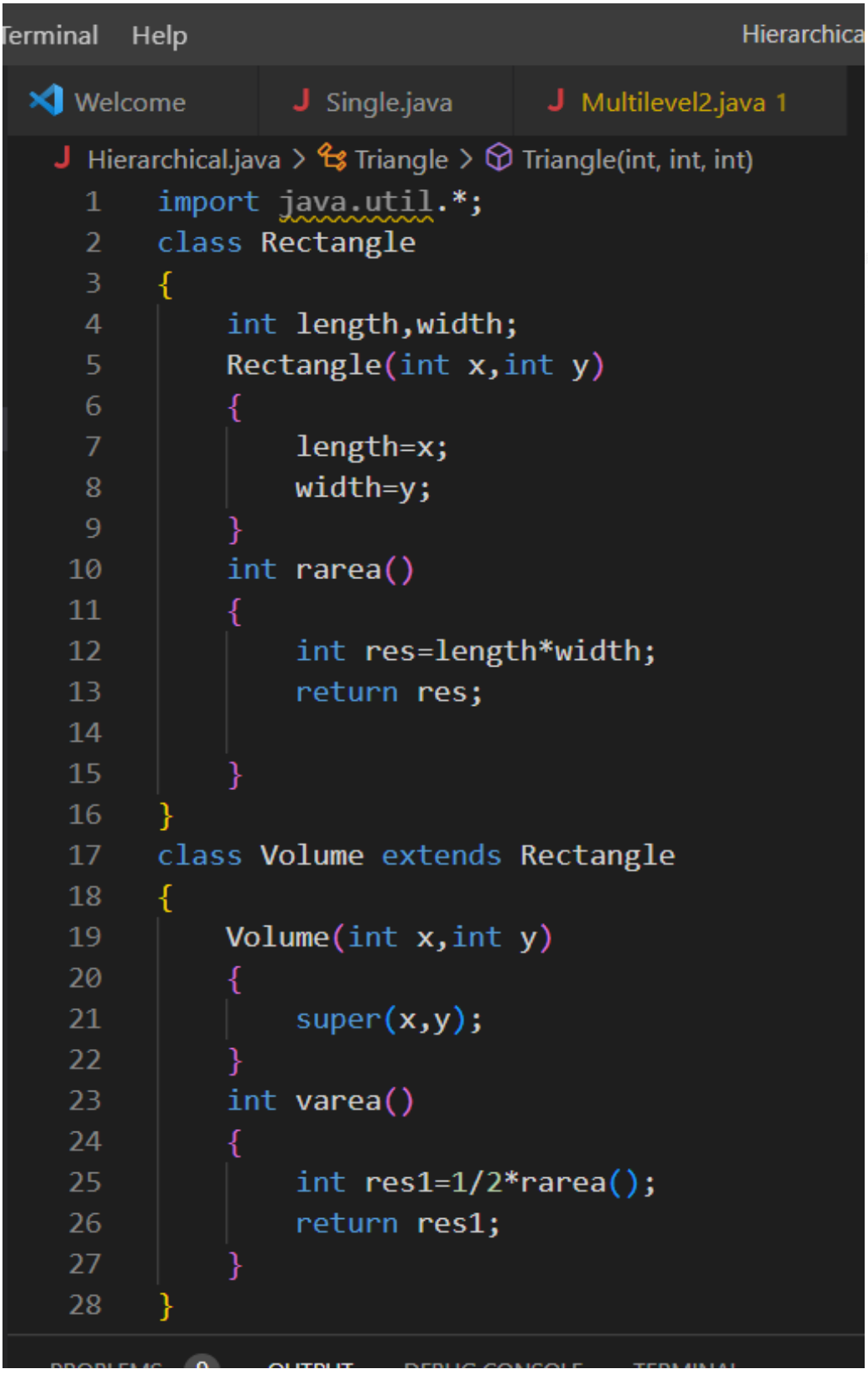
```
26          m3=c;
27      }
28      void stu_marks()
29      {
30          System.out.println("the sub1 marks is:"+m1);
31          System.out.println("the sub2 marks is:"+m2);
32          System.out.println("the sub3 marks is:"+m3);
33      }
34  }
35  class Total extends Marks
36  {
37      Total(int x,String y,int a,int b,int c)
38      {
39          super(x,y,a,b,c);
40
41      }
42      int total()
43      {
44          int tot=m1+m2+m3;
45          return tot;
46      }
47  }
48  class Percentage extends Total
49  {
50      Percentage(int x,String y,int a,int b,int c)
51      {
```

Multilevel2.java > Percentage > Percentage(int, String, int, int, int)

```
51      {
52          super(x,y,a,b,c);
53      }
54      double per()
55      {
56          double avg=total()/3;
57          return avg;
58      }
59  }
60  class Multilevel2
61  {
        Run | Debug
62      public static void main(String args[])
63      {
64          Percentage obj=new Percentage(x: 18,y: "yuvaraju",a: 90,b: 92,c: 93);
65          obj.stu();
66          obj.stu_marks();
67          int tm=obj.total();
68          double pa=obj.per();
69          System.out.println("the student total marks is"+tm);
70          System.out.println("the student total marks is"+pa);
71      }
72  }
```

3. Hierarchical Inheritance

Process of deriving one or more subclasses from one super class is called hierarchical inheritance



```java
}
class Triangle extends Rectangle
{
    int height;
    Triangle(int x,int y,int z)
    {
        super(x,y);
        height=z;
    }
    int tarea()
    {
        int res3=rarea()*height;
        return res3;
    }
}
class Hierarchical
{
    Run | Debug
    public static void main(String args[])
    {
        Triangle obj1=new Triangle(x: 10,y: 20,z: 30);
        int ta=obj1.tarea();
        int ra=obj1.rarea();
        System.out.println("the area of rectangle is:"+ra);
        System.out.println("the area of triangle is:"+ta);
        Volume obj2=new Volume(x: 10,y: 20);
        int va=obj2.varea();
        System.out.println("the area of volume is:"+va);
```

4. Hybrid Inheritance

Combination of above any inheritance is called hybrid inheritance

5. Multiple inheritance

Process of deriving a subclass from one or more superclasses is called multiple inheritance.

Java does not directly implement multiple inheritance.

however, this concept is implemented using a secondary inheritance path in the form of interfaces. Class A

{

}

Class B

{

}

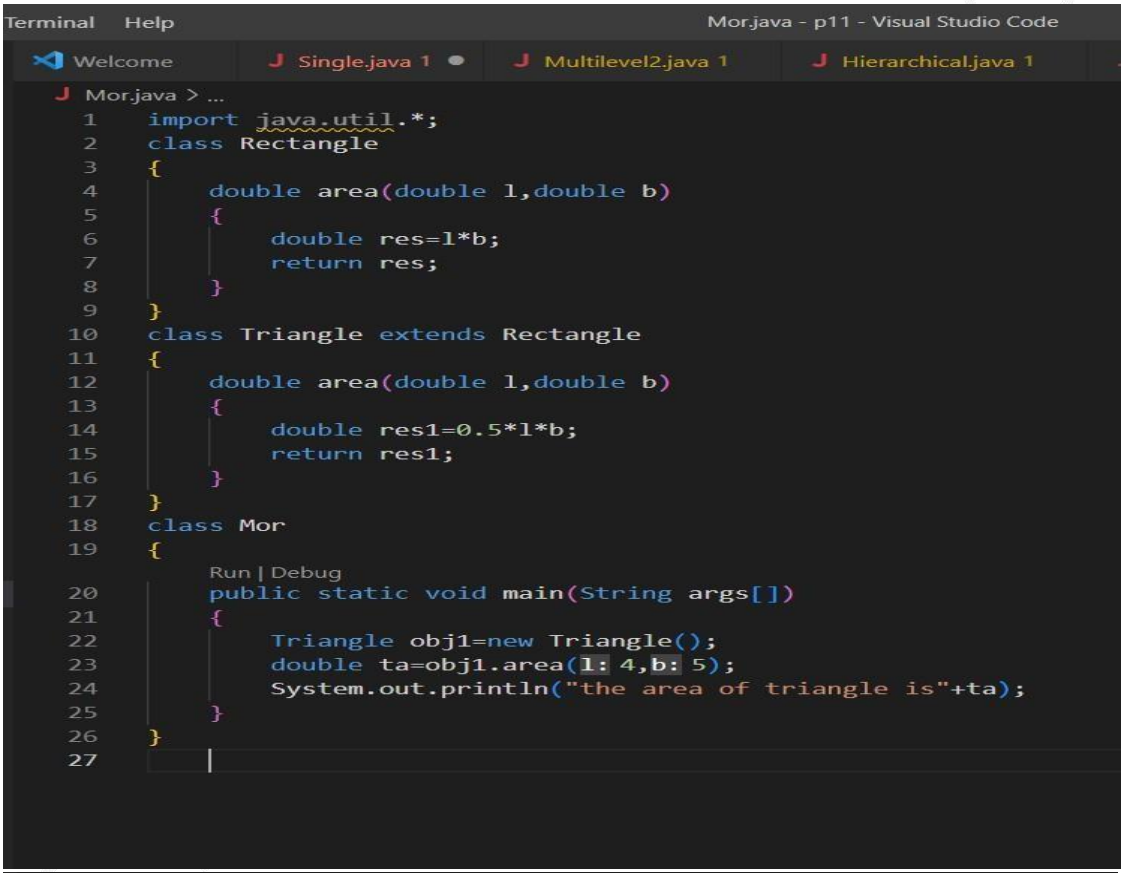
Class C extends A,B // java does not allow this { }

{

}

Method Overrididng

A method in subclass, whose name, parameter list and return type are same as that of the method in superclass is called overrided methods.




```
[Running] cd "f:\p11\" && javac Mor.java && java Mor
the area of triangle is10.0

[Done] exited with code=0 in 3.822 seconds
```

Abstract Methods and Classes

- An Abstract method is a method without method body or a method without implementation.
- An Abstract method is written when the same method has to perform different tasks depending on the object calling it.

Example:

```
class A                          // Automatically Becomes Abstract Class
{
    void m1();                   // Abstract Method
    void m2()                    // Concrete Method
{
    System.out.println("method 2");
}
}
```

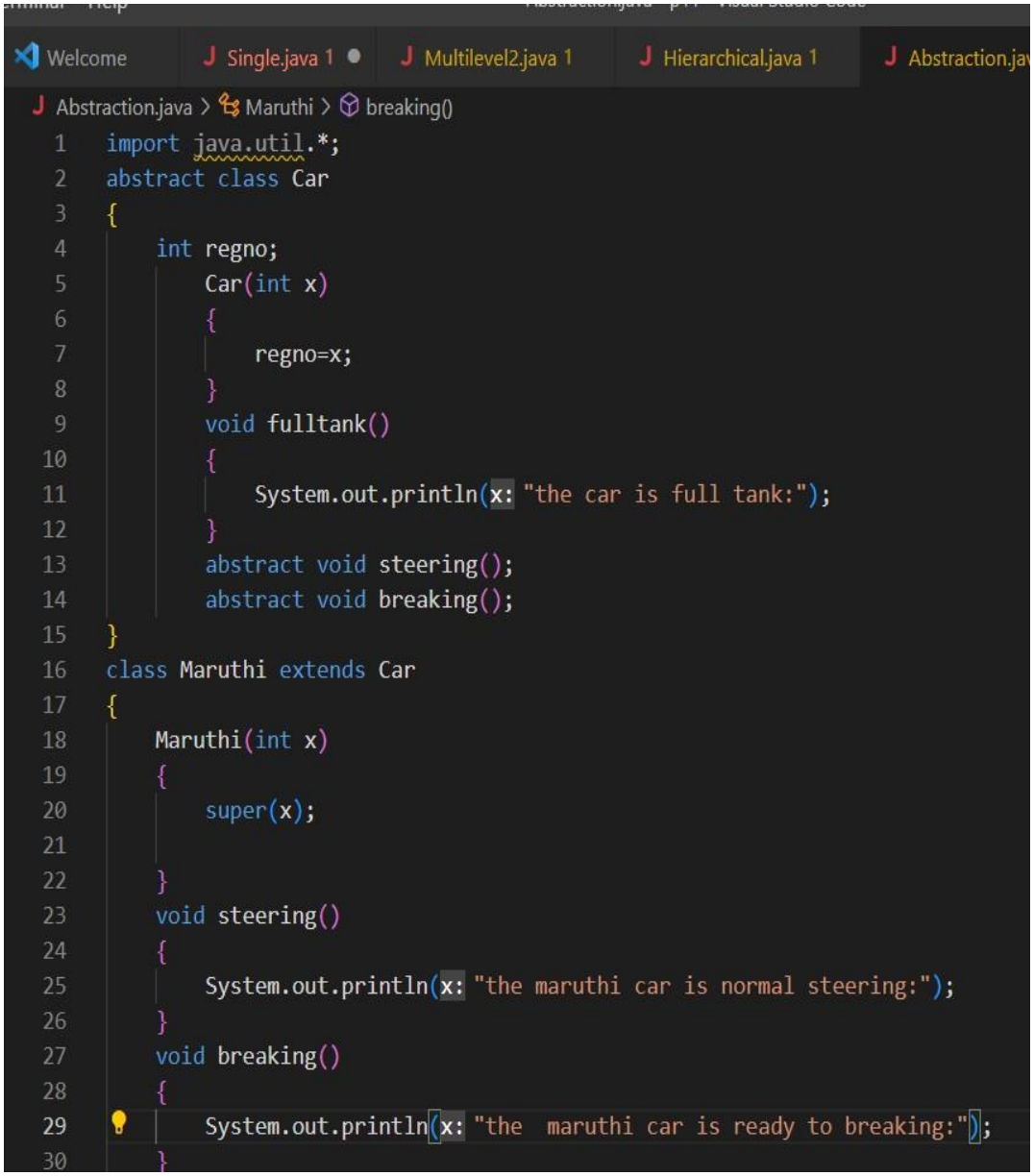
- A Class that contains one or more Abstract Methods is called Abstract Class.
- An Abstract class is a class that contains 0 or more Abstract Methods.
- Abstract class can contain instance variables and concrete methods in addition to abstract methods. Since, abstract class contains incomplete methods, it is not possible to estimate the total memory required to create the object.

Example:

```
Abstract class MyClass
{
abstract void calculate(double x);
}
Class Sub1 extends MyClass
{
void calculate(double x)
{
System.out.println("Square ="+(x*x));
```

```
}
}
Class Sub3 extends MyClass
{
void calculate(double x)
{
System.out.println("Square Root ="+Math.sqrt(x));
}
}
Class Different
{
public static void main(String args[])
{
Sub1 obj1 = new Sub1();

Sub2 obj2 = new Sub2();

Sub3 obj3 = new Sub3();

obj1.calculate(3);

obj2.calculate(4);

obj3.calculate(5);
}
}
```

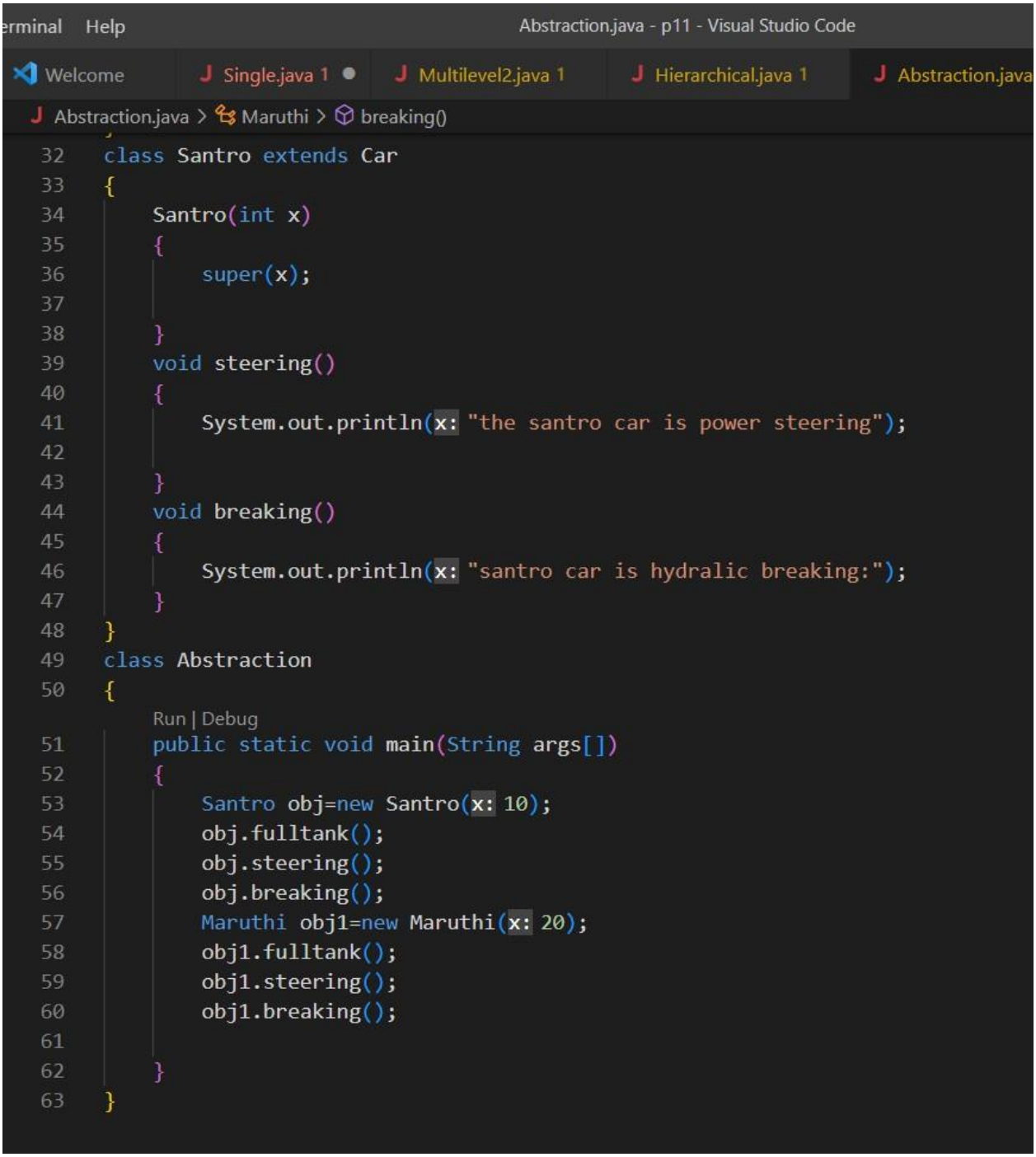
Example:2



Welcome | Single.java 1 | Multilevel2.java 1 | Hierarchical.java 1 | Abstraction.ja

Abstraction.java > Maruthi > breaking()

```
1   import java.util.*;
2   abstract class Car
3   {
4       int regno;
5           Car(int x)
6           {
7               regno=x;
8           }
9           void fulltank()
10          {
11              System.out.println(x: "the car is full tank:");
12          }
13          abstract void steering();
14          abstract void breaking();
15  }
16  class Maruthi extends Car
17  {
18      Maruthi(int x)
19      {
20          super(x);
21
22      }
23      void steering()
24      {
25          System.out.println(x: "the maruthi car is normal steering:");
26      }
27      void breaking()
28      {
29          System.out.println(x: "the  maruthi car is ready to breaking:");
30      }
```




```
[Running] cd "f:\p11\" && javac Abstraction.java && java Abstraction
the car is full tank:
the santro car is power steering
santro car is hydralic breaking:
the car is full tank:
the maruthi car is normal steering:
the  maruthi car is ready to breaking:

[Done] exited with code=0 in 2.407 seconds
```

final class : prevents inheritance

sometimes we may like to prevent a class being further subclasses for security reasons. A class that cannot be subclasses is called a final class. Any attempt to inherit final classes will cause an error and the compiler will not allow it.

```
final class A
{
}
class B extend A                    //error, cannot inherit a because it is a final class
{
}
```

Interfaces:

Defining an Interface

- An Interface is basically a kind of class
- Like classes, interface contain methods and variables but with a major difference.
- ***The difference is that interfaces define only***
  - ***Abstract Method &***
  - ***Final and Static Variables***
- i.***e methods*** are declared without any body ***and variables*** are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized.
- All ***Methods*** and ***Variables*** in the interface are implicitly ***public.***

The syntax for defining an interface is very similar to that of defining a class

```
Interface InterfaceName
{
static and final Variables

Abstract Methods
}
```

Where ***Interface*** is the keyword and ***InterfaceName*** is any valid java variable

Example:

```
Interface Item
{
static final int code = 1001;
static final String name = "Fan";

void display();

}
```

Implementing Interface

- ✓ An Interface will have 0 or more abstract methods which are all **public and abstract by default.**
- ✓ An Interface can have variables **which are public, static and final by default, means all the variables of the interface are constants**.
- ✓ Objects cannot be created to an interface whereas reference can be created.
- ✓ ***Once interface is defined, any number of classes can implement an interface.***
- ✓ ***Also one class can implement any number of interfaces.***
- ✓ To Implement an interface, a class must create the complete set of methods defined by the interface.
- ✓ To implement an interface, include the ***implements clause*** in a class definition, and then create the methods defined by the interface.
- ✓ General form of a class that includes the implements clause looks like

Class ClassName [extends SuperClass] [implements Interface1[,... Interface N]]

{

// class body

}

Example

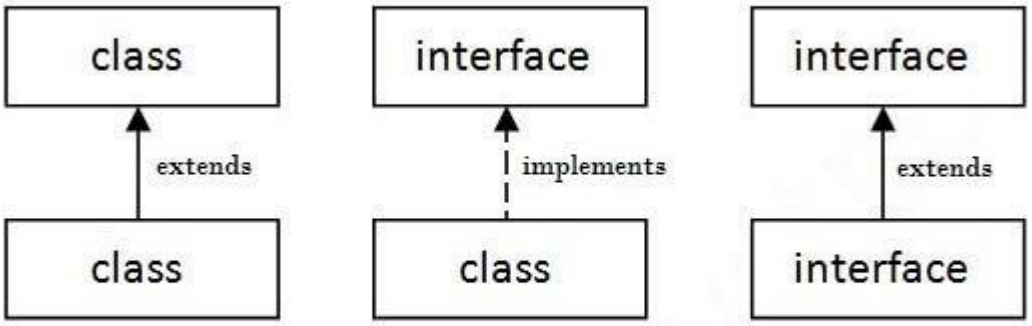
Class A Extends B Implements I1,I2

{

}

- ✓ i.e if a class implements more than one interface, the interfaces are separated with a comma.

The Relationship between classes and Interfaces are



Example:

```
Interface Bank
{
float rateOfInterest();
}
Class SBI implements Bank
{
public float reateOfInterest()
{
    return (7.8f);
}
}
class ICICI implements Bank
{
public float reateOfInterest()
{
return (9.8f);
```

```
}
}
class IB implements Bank
{
public float reateOfInterest()
{
return (8.8f);
}
}
class InterfaceDemo
{
public staticvoid main(String args[])
{
SBI obj1 = new SBI();
float sbi_roi = obj1.rateOfInterest();

ICICI obj1 = new ICICI();
float icici_roi = obj1.rateOfInterest();
IB obj1 = new IB();
float ib_roi = obj1.rateOfInterest();
System.out.println("SBI rate of Interest is "+ sbi_roi);
System.out.println("ICICI rate of Interest is "+ sbi_icici);
System.out.println("IB rate of Interest is "+ sbi_ib);
}
}
```

Interfaces can be Extended

- ✓ Like classes, interface can also be extended.
- ✓ i.e an interface can be sub interfaced from other interfaces.
- ✓ The new sub interface will inherit all the members of the super interface in the manner similar to subclasses.
- ✓ This is achieved using the keyword "extends".
- ✓ General form of extending interfaces is

Syntax:

Interface NameNew extends name1[,...nameN]

{

Body of Interface

}

Example:

```
interface A
{
void meth1();
void meth2();
}
interface B extends A
{
void meth3();
}
Class  MyClass implements B
{
public void meth1()
{
System.out.println("implementing  meth1().....");
}
public void meth2()
{
System.out.println("Implementing  meth2()....");
}
public void meth3()
{
System.out.println("Implementing  meth3()....");
}
```

```
}
Class InterfaceDemo
{
Public static void main(string args[])
{
MyClass obj = new MyClass();
obj.meth1();
obj.meth2();
obj.meth3();
}
}
```

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

  Note : if a class that implements an interface and the class does not give implementations to all the methods of the interface, then the class becomes an abstract class and cannot be instantiated