# UNIT – 2

## Topics to be covered:

Classes and objects, class declaration, creating objects, methods, constructors and constructor overloading, garbage collector, importance of static keyword and examples, this keyword, arrays, command line arguments, nested classes.

## Introduction to classes

### Fundamentals of the class

A class is a group of objects that has common properties. It is a **template** or blueprint from which objects are created. The objects are the **instances** of class. Because an object is an instanceof a class, you will often see the two words *object* and *instance* used interchangeably.

A class is used to define new type of the data. Once defined, this new type can be used to create objects of its type. The class is the logical entity and the object is the logical and physical entity.

The general form of the class

A class is declared by use of the **class** keyword. The classes that have been used up to thispoint are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a **class** definition is shown here:

```
class classname
{
	type instance-variable1;
	type instance-variable2;
	……………………………….
	……………………………..
	type instance-variableN;

	type method1(parameterlist)
	{
		//body of the method1
	}
	type method2(parameterlist)
	{
		//body of the method2
	}
	…………………………………….
	…………………………………….
	type methodN(parameterlist)
	{
		//body of the methodN
```

```
        }
}
```

- The data or variables, defined within the class are called, ***instance variable***.
- The methods also contain the code.
- The methods and instance variable collectively called as ***members***.
- Variable declared within the methods are called ***local variables***.

**A Simple Class**

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods.

```
class Box
{       //instance variables
        double  width;
        double  height;
        double depth;
}
```

As stated, a class defines new data type. The new data type in this example is, Box. This defines the template, but does not actually create object.

> **Note:** By convention the First letter of every word of the class name starts with Capital letter, but not compulsory. For example ―box‖ is written as ―Box‖ The First letter of the word of the method name begins with small and remaining first letter of every word starts with Capital letter, but not compulsory. For example ―boxVolume()‖

**Creating the Object**

There are three steps when creating an object from a class:

- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' key word is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

**Step 1:**

```
Box b;
```

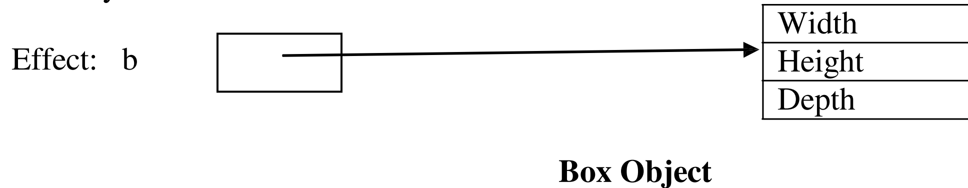
**Effect:** b | null |

Declares the class variable. Here the class variable contains the value **null**. An attempt to access the object at this point will lead to Compile-Time error.

**Step 2:**

```
Box b=new Box();
```

Here new is the keyword used to create the object. The object name is b. The **new** operator allocates the memory for the object, that means for all instance variables inside the object, memory is allocated.




**Box Object**

**Step 3:**

There are many ways to initialize the object. The object contains the instance variable. The variable can be assigned values with reference of the object.

```
b.width=12.34;
b.height=3.4;
b.depth=4.5;
```

Here is a complete program that uses the **Box** class:

**BoxDemo.java**

```
class Box
{
        double  width;
        double  height;
        double depth;
}
// This class declares an object of type Box.


class BoxDemo
{
public static void main(String args[])
{
        //declaring the object (Step 1) and instantiating (Step 2) object
        Box mybox = new Box();
        double vol;
        // assign values to mybox's instance variables  (Step 3)
        mybox.width  =  10;
        mybox.height = 20;
        mybox.depth = 15;
        // compute volume of box
        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Volume is " + vol);
}
}
```

When you compile this program, youwill find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Javacompiler automatically puts each class into its own **.class**

file. It is not necessary for both the**Box** and the **BoxDemo** class to actually be in the same source file.

To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

Volume is 3000.0

As stated earlier, each object has its own copies of the instance variables. This means thatif you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is importantto understand that changes to the instance variables of one object have no effect on the instancevariables of another. For example, the following program declares two **Box** objects:

```
// This program declares two Box objects.
class Box
{
	double width;
	double height;
	double depth;
}
class BoxDemo2
{
public static void main(String args[])
{
	Box mybox1 = new Box();
	Box mybox2 = new Box();
	double vol;
// assign values to mybox1's instance variables
	mybox1.width = 10;
	mybox1.height = 20;
	mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
	mybox2.width = 3;
	mybox2.height = 6;
	mybox2.depth = 9;
// compute volume of first box
	vol = mybox1.width * mybox1.height * mybox1.depth;
	System.out.println("Volume is " + vol);
// compute volume of second box
	vol = mybox2.width * mybox2.height * mybox2.depth;
	System.out.println("Volume is " + vol);
}
}
```

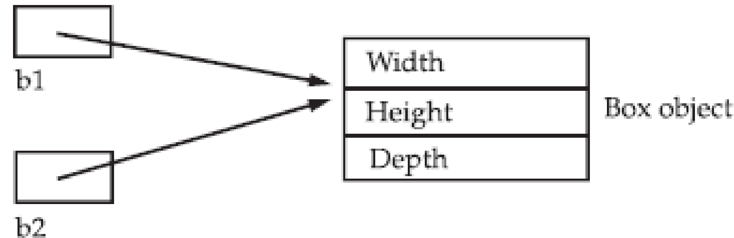
**Assigning Object Reference Variables**

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

Box b1 = new Box();
Box b2 = b1;

You might think that **b2** is being assigned a reference to a copy of the object referred to by**b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However,this would

be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the*same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of theoriginal object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changesmade to the object through **b2** will affect the object to which **b1** is referring, since they are thesame object.
This situation is depicted here:




## Introduction to Methods

Classes usually consist of two things: ***instance variables and methods***. The topic of methods is a large one because Java gives them so much power and flexibility. In fact, much of the next chapter is devoted to methods.

This is the general form of a method:

*type name*(*parameter-list*)
{
    // body of method
}
Here, *type* specifies the type of data returned by the method. This can be any valid type,including class types that you create. If the method does not return a value, its return typemust be **void**. The name of the method is specified by *name.* This can be any legal identifierother than those already used by other items within the current scope. The *parameter-list* is asequence of type and identifier pairs separated by commas. Parameters are essentially variablesthat receive the value of the ***arguments***passed to the method when it is called. If the methodhas no parameters, then the parameter list will be empty.

Methods that have a return type other than **void** return a value to the calling routine usingthe following form of the **return** statement:

**return *value*;**

Here, *value* is the value returned.

**Adding a method to the Box class**

**Box.java**

```
class Box
{
	double width, height, double depth;
// display volume of a box
void volume()
```

```
	{
	System.out.print("Volume is ");
	System.out.println(width * height * depth);
	}
}
```

Here the method name is "volume()". This methods contains some code fragment for computing the volume and displaying. This method cane be accessed using the object as in the following code:

**BoxDemo3.java**

```
class BoxDemo3
{
public static void main(String args[])
{
Box mybox1 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's

// display volume of first box
mybox1.volume();
// display volume of second box
}
}
```

### Returning a Value

A method can also return the value of specified type. In this case the type of the method should be clearly mentioned. The method after computing the task returns the value to the **caller** of the method.

**BoxDemo3.java**

```
Box
{
	double width, height, depth;

double volume()
	{
		return (width*height*depth);
	}
}

class BoxDemo3
{
public static void main(String args[])
{
	Box mybox1 = new Box();
	// assign values to mybox1's instance variables
	mybox1.width = 10;
```

```
        mybox1.height = 20;
        mybox1.depth  =  15;
        double vol;
/* assign different values to mybox2's
//calling the method vol=
mybox1.volume();
System.out.println("the Volume is:"+vol);
}
}
```

**Adding a method that takes the parameters**

We can also pass arguments to the method through the object. The parameters separated with comma operator. The values of the actual parameters are copied to the formal parameters in the method. The computation is carried with formal arguments, the result is returned to the caller of the method, if the type is mentioned.

```
double volume(double w,double h,double d)
        {
                width=w;
                height=h;
                depth=d;
                return (width*height*depth);
        }
```

# Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created. Even if we use some method to initialize the variable, it would be better this initialization is done at the time of the object creation.

A***constructor***initializes an object immediately upon creation. It has the same name as theclass in which it resides and is syntactically similar to a method. Once defined, the constructoris automatically called immediately after the object is created, before the **new** operator completes.Constructors look a little strange because they have no return type, not even **void**. This isbecause the implicit return type of a class' constructor is the class type itself. It is the constructor'sjob to initialize the internal state of an object so that the code creating an instance will havea fully initialized, usable object immediately.

**Example Program:**

```
class Box
{
        double width;
        double height;
        double depth;
// This is the constructor for Box.
        Box()
```

```
	{
		System.out.println("Constructing Box");
		width = 10;
		height = 10;
		depth = 10;
	}
// compute and return volume
	double volume()
	{
		return width * height * depth;
	}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
double vol;
// get volume of first box vol =
mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
}}
```

## Parameterized Constructors

While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is notvery useful—all boxes have the same dimensions. What is needed is a way to construct **Box**objects of various dimensions. The easy solution is to add parameters to the constructor.

```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box
{
	double  width;
	double height;
	double depth;
// This is the constructor for Box.
Box(double w, double h, double d)
	{
	width = w;
	height = h;
	depth = d;
	}
// compute and return volume
double volume() {
return width * height * depth;
}
}

class BoxDemo7 {
```

```
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box vol =
mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box vol =
mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

## Constructor-overloading

In Java it is possible to define two or more class constructors that share the same name, as long as their parameter declarations are different. This is called constructor overloading.

When an overloaded constructor is invoked, Java uses the type and/or number ofarguments as its guide to determine which version of the overloaded constructor to actuallycall. Thus, overloaded constructors must differ in the type and/or number of their parameters.

**Example: All the constructors names will be same, but their parameter list is different.**

**OverloadCons.java**

```
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

```
class OverloadCons
{
public static void main(String args[])
{
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

## Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wonderinghow such objects are destroyed and their memory released for later reallocation. In somelanguages, such as C++, dynamically allocated objects must be manually released by use ofa **delete** operator. Java takes a different approach; it handles deallocation for you automatically.The technique that accomplishes this is called ***garbage collection***.Java has its own set of algorithms to do this as follow.

There are Two Techniques: **1) Reference Counter 2) Mark and Sweep**. In the Reference Counter technique, when an object is created along with it a reference counter is maintained in the memory. When the object is referenced, the reference counter is incremented by one. If the control flow is moved from that object to some other object, then the counter value is decremented by one. When the counter reaches to zero (0), then it's memory is reclaimed.

In the Mark and Sweep technique, all the objects that are in use are ***marked*** and are called live objects and are moved to one end of the memory. This process we call it as compaction. The memory occupied by remaining objects is reclaimed. After these objects are deleted from the memory, the live objects are placed in side by side location in the memory. This is called copying.

It works like this: when noreferences to an object exist, that object is assumed to be no longer needed, and the memoryoccupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.Garbage collection only occurs during the execution of your

program. The main job of this is to release memory for the purpose of reallocation. Furthermore,different Java run-time implementations will take varying approaches to garbage collection

**The finalize( ) Method**

Sometimes an object will need to perform some action when it is ***destroyed***. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called ***finalization***. By using finalization, you candefine specific actions that will occur when an object is just about to be reclaimed by thegarbage collector.

To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )**method, you will specify those actions that must be performed before an object is destroyed.The garbage collector runs periodically, checking for objects that are no longer referenced byany running state or indirectly through other referenced objects

The **finalize**( ) method has this general form:

```
protected void finalize( )
{

// finalization code here

}
```

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class.

**Overloading methods**

In Java it is possible to define **two or more methods** within the same class that share thesame name, as long as their parameter declarations are different. When this is the case, themethods are said to be *overloaded,* and the process is referred to as ***method overloading***. Methodoverloading is one of the ways that Java supports **polymorphism**.

When an overloaded method is invoked, Java uses the type and/or number ofarguments as its guide to determine which **version** of the overloaded method to actuallycall. Thus, overloaded methods must differ in the type and/or number of their parameters.While overloaded methods may have different return types, the return type alone isinsufficient to distinguish two versions of a method. When Java encounters a call to anoverloaded method, it simply executes the version of the method whose parameters matchthe arguments used in the call.

**Here is a simple example that illustrates method overloading:**

// Demonstrate method overloading.

```
class OverloadDemo
{
	void test()
	{
		System.out.println("No parameters");
	}
// Overload test for one integer parameter.
	void test(int a)
	{
		System.out.println("a: " + a);
	}
}

class Overload
{
public static void main(String args[])
{
	OverloadDemo ob = new OverloadDemo();
	double result;
// call all versions of test()
	ob.test();
	ob.test(10);
}
}
```

The test() method is overloaded two times, first version takes no arguments, second version takes one argument. When an overloaded method is invoked, Java looks for a match between arguments of the methods.Method overloading supports **polymorphism** because it is one way that Java implementsthe –one interface, multiple methods‖ paradigm.

## static keyword

There will be times when you will want to define a class member that will be usedindependently of any object of that class. Normally, a class member must be accessed onlyin conjunction with an object of its class. However, it is possible to create a member that canbe used by itself, without reference to a specific instance.

To create such a member, precedeits declaration with the keyword **static**. When a member is declared **static**, it can be accessedbefore any objects of its class are created, and without reference to any object. You can declareboth methods and variables to be **static**. The most common example of a **static** member is**main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the classshare the same **static** variable.These variables are also called as "***Class Variable***".

Methods declared as **static** have several restrictions:

- They can only call other **static** methods.
- They must only access **static** data.

• They cannot refer to **this** or **super** in any way.

If you need to do computation in order to initialize your **static** variables, you can declare a**static** block that gets executed exactly once, <u>when the class is first loaded.</u> The followingexample shows a class that has a **static** method, some **static** variables, and a **static** initializationblock:

```
class UseStatic
	{
		//static variable
		static int a = 3;
		static int b;
		//static method
		static void meth(int x)
		 {
		System.out.println("x = " + x);
		System.out.println("a = " + a);
		System.out.println("b = " + b);
		}
		//static block
		static
		{
		System.out.println("Static block initialized.");
		b = a * 4;
		}
public static void main(String args[])
{
	//static methods are called without object
	meth(42);
}
}
```

**Note: Static blocks are executed first, even than the static methods.**

**static** variables **a** and **b**, as well as to the local variable **x**.Here is the output of the program:
Static block initialized.

x = 42
a = 3
b = 12

## <u>The this Keyword</u>

Sometimes a method will need to refer to the object that invoked it. To allow this, Java definesthe **this** keyword. **this** can be used inside any method to refer to the ***current object***. That is,**this** is always a reference to the object on which the method was invoked. You can use **this**anywhere a reference to an object of the current class' type is permitted.To better understand what **this** refers to, consider the following version of **Box( )**:

```
// A redundant use of this.
```

```
Box(double w, double h, double d)
        {
                this.width = w;
                this.height = h;
                this.depth = d;
        }
```

**Note: This is mainly used to hide the local variables from the instance variable**.

**Example:**

```
class  Box
{
        //instance variable
        double width, height, depth;
        Box(double width, double height, double depth)
        {
                //local variables are assigned, but not the instance variable
                width=width;
                height=height;
                depth=depth;
        }
}
```

To avoid the confusion, this keyword is used to refer to the instance variables, as follows:

```
class  Box
{
        //instance variable
        double width, height, depth;
        Box(double width, double height, double depth)
        {
                //the instance variable are assigned through the this keyword.
                this.width=width;
                this.height=height;
                this.depth=depth;
        }
}
```

## Introduction to Arrays

An Array is a collection of elements that share the same type and name. The elements from the array can be accessed by the index. To create an array, we must first create the array variable of the desired type. The general form of the One Dimensional array is as follows:

**type var_name[];**

Here type declares the base type of the array. This base type determine what type of elements that the array will hold.

Example:

**int month_days[];**

Here type is int, the variable name is month_days. All the elements in the month are integers. Since, the base type is int.

In fact, the value of **month_days** is set to **null**, which represents an arraywith no value. To link **month_days** with an actual, physical array of integers, you must allocateone using **new** and assign it to **month_days**. **new** is a special operator that allocates memory.

The general form of **new** as it applies to one-dimensional arrays appearsas follows:

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

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements inthe array, and *array-var* is the array variable that is linked to the array. That is, to use **new** toallocate an array, you must specify the type and number of elements to allocate. The elementsin the array allocated by **new willautomatically be initialized to zero**. This example allocatesa 12-element array of integers and links them to **month_days**.

**month_days = new int[10];**

month_days

| Element | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Once you have allocated an array, you can access a specific element in the array byspecifying its index within square brackets. All array indexes start at zero. For example,this statement assigns the value 28 to the second element of **month_days**.

**month_days[1] = 28;**

| Element | 0 | **28** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| Index | 0 | **1** | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

The next line displays the value stored at index 3.
System.out.println(month_days[3]);

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

**ArrayTest.java**

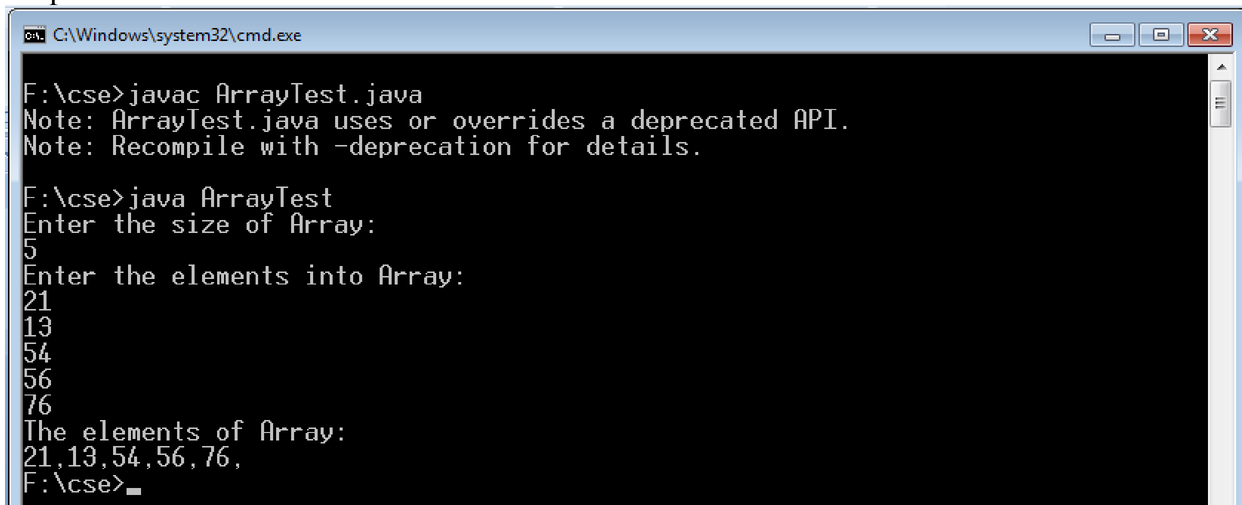
import java.io.*;
class ArrayTest

```
{
        public static void main(String args[]) throws IOException
        {

                DataInputStream dis=new DataInputStream(System.in);
                int a[]; //declaring array variable int n, i; //size
                of the array System.out.println("Enter the size
                of Array:");
                n=Integer.parseInt(dis.readLine());
                a=new int[n]; //allocating memory to array a and all the elements are set zero

                //read the elements into array
                System.out.println("Enter the elements into Array:");
                for(i=0;i<n;i++)
                {
                        a[i]=Integer.parseInt(dis.readLine());
                }
                //displaying the elements
                System.out.println("The elements of Array:");
                for(i=0;i<n;i++)
                {
                        System.out.print(a[i]+",");
                }
        }
}
```

Output




**Multidimensional Arrays**

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, lookand act like regular multidimensional arrays. However, as you will see, there are a coupleof

subtle differences. To declare a multidimensional array variable, specify each additionalindex using another set of square brackets. For example, the following declares a twodimensional

array variable called **twoD**.
int twoD[][] = new int[4][4];

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

| | Right Index Determines the Columns | | | |
|---|---|---|---|---|
| **Left index determines the Rows** | [0,0] | [0,1] | [0,2] | [0,3] |
| | [1,0] | [1,1] | [1,2] | [1,3] |
| | [2,0] | [2,1] | [2,2] | [2,3] |
| | [3,0] | [3,1] | [3,2] | [3,3] |

## Example Program for Matrix Addition

```
import java.io.*;
class AddMatrix
{
  public static void main(String args[]) throws IOException
  {
    int m, n, c, d;
    DataInputStream dis=new DataInputStream(System.in);

    System.out.println("Enter the number of rows and columns of matrix");
    m = Integer.parseInt(dis.readLine());
    n  = Integer.parseInt(dis.readLine());

    int first[][] = new int[m][n];
    int second[][] = new int[m][n];
    int sum[][] = new int[m][n];

    System.out.println("Enter the elements of first matrix");

    for (  c = 0 ; c < m ; c++ )
      for ( d = 0 ; d < n ; d++ )
        first[c][d] = Integer.parseInt(dis.readLine());

    System.out.println("Enter the elements of second matrix");

    for ( c = 0 ; c < m ; c++ )
      for ( d = 0 ; d < n ; d++ )
        second[c][d] = Integer.parseInt(dis.readLine());
```

```
    for ( c = 0 ; c < m ; c++ )
      for ( d = 0 ; d < n ; d++ )
         sum[c][d] = first[c][d] + second[c][d];  //replace '+' with '-' to subtract matrices

    System.out.println("Sum of entered matrices:-");

    for ( c = 0 ; c < m ; c++ )
    {
      for ( d = 0 ; d < n ; d++ )
        System.out.print(sum[c][d]+"\t");

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

**Example program for Matrix Multiplication**

```
import java.io.*;

class MulMatrix
{
  public static void main(String args[]) throws IOException
  {
    int m, n, p, q, sum = 0, c, d, k;

DataInputStream dis = new DataInputStream(System.in);
    System.out.println("Enter the number of rows and columns of first matrix");
    m = Integer.parseInt(dis.readLine());
    n = Integer.parseInt(dis.readLine());

    int first[][] = new int[m][n];

    System.out.println("Enter the elements of first matrix");

    for ( c = 0 ; c < m ; c++ )
      for ( d = 0 ; d < n ; d++ )
        first[c][d] = Integer.parseInt(dis.readLine());

    System.out.println("Enter the number of rows and columns of second matrix");
    p = Integer.parseInt(dis.readLine());
    q = Integer.parseInt(dis.readLine());

    if ( n != p )
      System.out.println("Matrices with entered orders can't be multiplied with each other.");
    else
```

```
    {
      int second[][] = new int[p][q];
      int multiply[][] = new int[m][q];

      System.out.println("Enter the elements of second matrix");

      for ( c = 0 ; c < p ; c++ )
        for ( d = 0 ; d < q ; d++ )
          second[c][d] = Integer.parseInt(dis.readLine());

      for ( c = 0 ; c < m ; c++ )
      {
        for ( d = 0 ; d < q ; d++ )
        {
          for ( k = 0 ; k < p ; k++ )
          {
            sum = sum + first[c][k]*second[k][d];
          }

          multiply[c][d] = sum;
          sum = 0;
        }
      }

      System.out.println("Product of entered matrices:-");

      for ( c = 0 ; c < m ; c++ )
      {
        for ( d = 0 ; d < q ; d++ )
          System.out.print(multiply[c][d]+"\t");

        System.out.print("\n");
      }
    }
  }
}
```

**Alternative Array Declaration Syntax**

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

*type*[ ] *var-name;*

Here, the square brackets follow the type specifier, and not the name of the array variable.
For example, the following two declarations are equivalent:

```
int al[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

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

creates three array variables of type **int**. It is the same as writing

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

The alternative declaration form is also useful when specifying an array as a return type for a method. Both forms are used.

## Command Line arguments

The java command-line argument is an argument i.e. passed at the time of running the java program.The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt. The command line arguments can be accessed easily, because they are stored as Strings in String array passed to the args in the main method. The first command line argument is stored in args[0], the second argument is stored in args[1], the third argument is stored in args[2], and so on.

**Example program:**

**Write a Java program to read all the command line arguments?**

```
import java.io.*;
class CommnadLine
{
        public static void main(String args[])
        {
```

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

Here the String class has a method length(), which is used to find the length of the string. This length can be used to read all the arguments from the command line.

## Nested classes

It is possible to define a class within another class; such classes are known as *nested classes.* The scope of the nested class (***inner class***) is bounded by the scope of it's enclosing class (***outer class***). Some important points of nested class are as follow:

- The nested class has right to access members of the enclosing class, including the private membersdirectly.
- However, the enclosing class has no right to access the members of nested class directly, but with the help of the inner class object it can access.

**Syntax**

Following is the syntax to write a nested class. Here, the class **Outer_Demo** is the outer class and the class **Inner_Demo** is the nested class.

```
class Outer{
  class Nested
          {
                    // body of the Inner class
           }
//body of the outer class
}
```

There are two types of nested class: static and non-static.

**Static inner class**

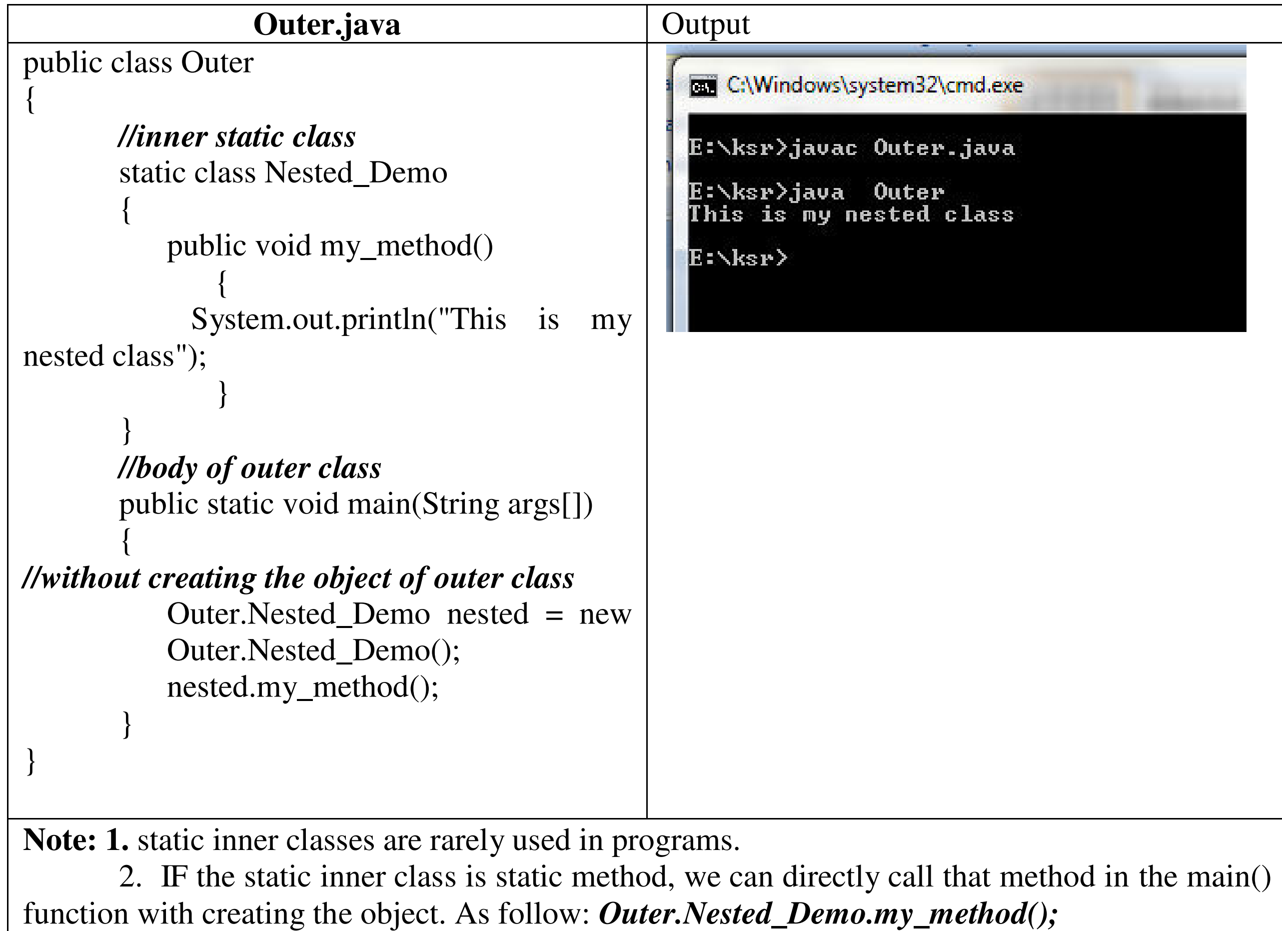
A static inner class is a nested class which is a static member of the outer class. It can be accessed ***without instantiating the outer class***, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows −

**Syntax**

```
class MyOuter
{
```

```
    static class Nested_Demo
              {
               }
}
```

**Example program:**

| **Outer.java** | Output |
| --- | --- |
| public class Outer<br>{<br>    ***//inner static class***<br>    static class Nested_Demo<br>    {<br>        public void my_method()<br>            {<br>          System.out.println("This is my nested class");<br>            }<br>    }<br>    ***//body of outer class***<br>    public static void main(String args[])<br>    {<br>***//without creating the object of outer class***<br>        Outer.Nested_Demo nested = new Outer.Nested_Demo();<br>        nested.my_method();<br>    }<br>} |  |

**Note: 1.** static inner classes are rarely used in programs.
2. IF the static inner class is static method, we can directly call that method in the main() function with creating the object. As follow: ***Outer.Nested_Demo.my_method();***

**Non-static inner class:**

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

**Example program:**

```
public class Outer2
```

```
{       int outer_x=100;
        void test()
        {  //creating the object of Inner class
                Inner inn=new Inner();
                inn.disp();
        }
        //inner non-static class
        public class Inner
        {   public void disp()
                {
        System.out.println("The value of x is:"+outer_x);
                }
        }
        public static void main(String args[])
        { //creating the out class object
                Outer2 out=new Outer2();
                out.test();
        }
}
```

```
C:\Windows\system32\cmd.exe

E:\ksr>javac Outer2.java

E:\ksr>java  Outer2
The value of x is:100

E:\ksr>_
```

## Introduction to Strings ( Topic beyond Syllabus)

**String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.

- The **first thing** to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects.

For example, in thestatement

**System.out.println("This is a String, too");**

the string –This is a String, too‖ is a **String** constant.

- The **second thing** to understand about strings is that objects of type **String** are immutable; once a **String** object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:
    - • If you need to change a string, you can always create a new one that containsthe modifications.

• Java defines a peer class of **String**, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java

Strings can be created in a many ways. The easiest is to use a statement like this:

1. String initialization

   **String myString = "this is a test";**

2. Reading from input device

   **DataInputStream dis=new DataInputStream(System.in);**
   **String st=dis.readLine();**

Once you have created a **String** object, you can use it anywhere that a string is allowed. For example, this statement displays **myString**:

System.out.println(myString);

Java defines one operator for **String** objects: **+**. It is used to concatenate two strings.
For example, this statement:

**String myString = "I" + " like " + "Java.";**

results in **myString** containing –I like Java.‖

The **String** class contains several methods that you can use.
Here are a few.

1. **equals( ) -** used to test whether two strings are equal or not
2. **length()** –used to find the length of the string
3. **charAt(i) -** used to retrieve the character from the string at the index i.
4. **compareTo(String)** –returns 0, if the string lexicographically equals to the argument, returns greater than 0 if the argument is lexicographically greater than this string, returns less than 0 otherwise.
5. **indexOf(char)** –returns the index of first occurrence of the character.
6. **lastIndesOf(char)**- returns the last index of the character passed to it.
7. **concat(String)** -Concatenates the string with the specified argument

Example :**int n=myString.length(); //gives the length of the string**