MODULE II

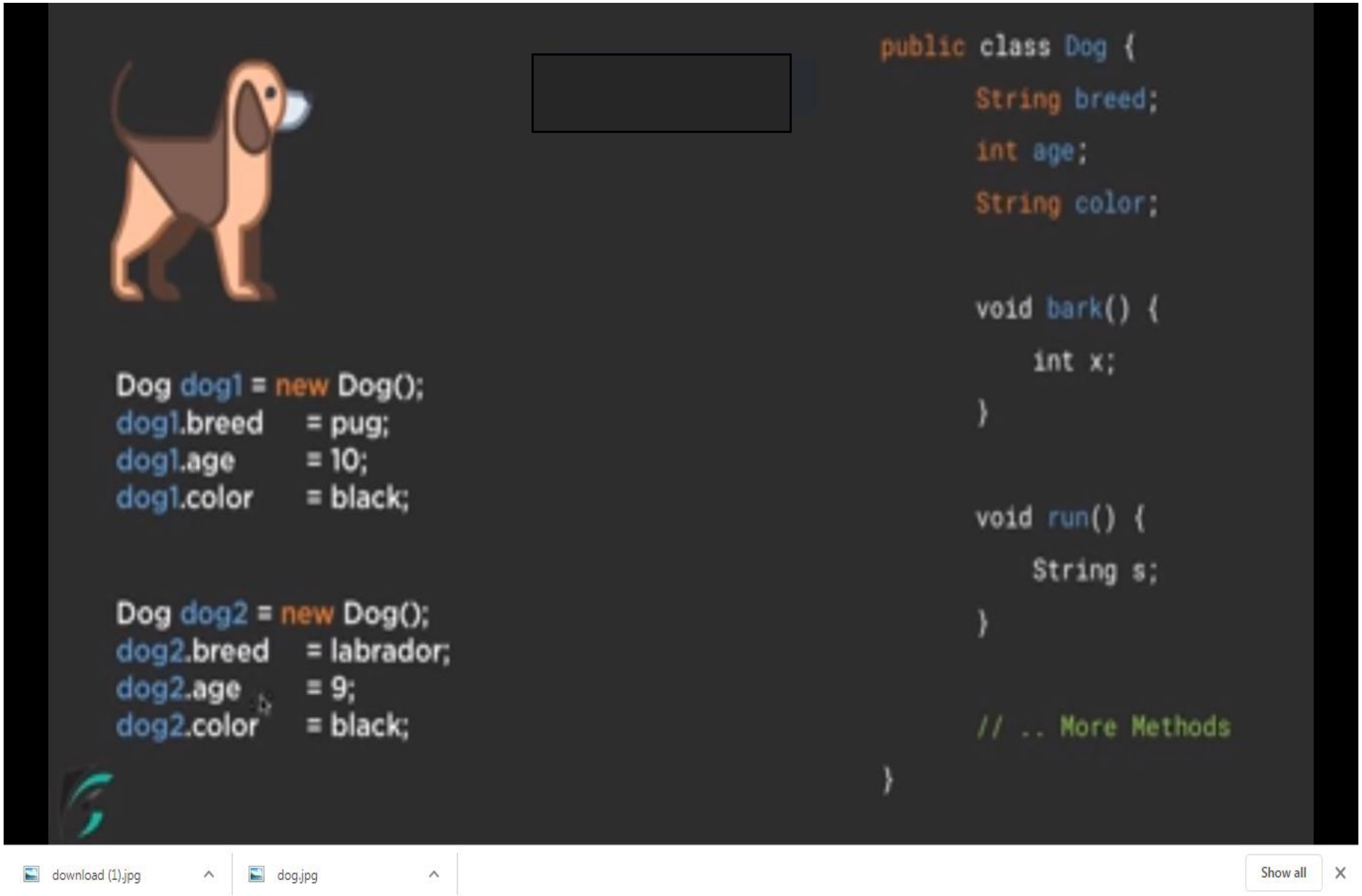
Syllabus of module 2

- Classes fundamentals
- Objects
- Methods
- Constructors
- Parameter passing
- Overloading
- Access control keywords.

Creating and Using Classes in Java

- Syntax of Class :

```
class classname
{
        type instance-variable1;
        type instance-variable2;
        // ...
        type instance-variableN;
        type methodname1(parameter-list)
        {       // body of method       }
        type methodname2(parameter-list)
        {       // body of method       }
        // ...
        type methodnameN(parameter-list)
        {       // body of method       }
}
```



[redacted]

```
Dog dog1 = new Dog();
dog1.breed    = pug;
dog1.age      = 10;
dog1.color    = black;


Dog dog2 = new Dog();
dog2.breed    = labrador;
dog2.age      = 9;
dog2.color    = black;
```

```
public class Dog {
        String breed;
        int age;
        String color;

        void bark() {
            int x;
        }

        void run() {
            String s;
        }

        // .. More Methods
}
```

- Class defines a new data type.
- Once defined, it can be used to create objects of that type.
- A class is a template for an object, and an object is an instance of a class.
- The methods and variables defined within a class are called members of the class.
- The variables, defined within a class are called instance variables. The data for one object is separate and unique from the data for another.
- Methods determine how class data can be used.

- Example:

  Define a class **Box** which defines three instance variables: **width**, **height**, and **depth**.

```
class Box
{
        double width;
        double height;
        double depth;
}
```

- This class defines a new data type called **Box**

Declaring a class Object

- An object is an instance of a class.
- Declaration of an object is a two-step process :
  - Declare a variable of the class type.
  - Acquire an actual, physical copy of the object and assign it to that variable. This can be done using the **new** operator.

- The new operator dynamically allocates memory for an object and returns a reference to it. This reference is the address in memory of the object allocated by new. This reference is then stored in the variable.
- Example :

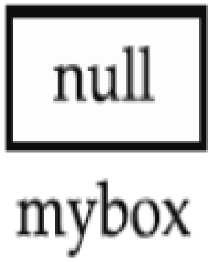
```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The above statements can also be written as follows

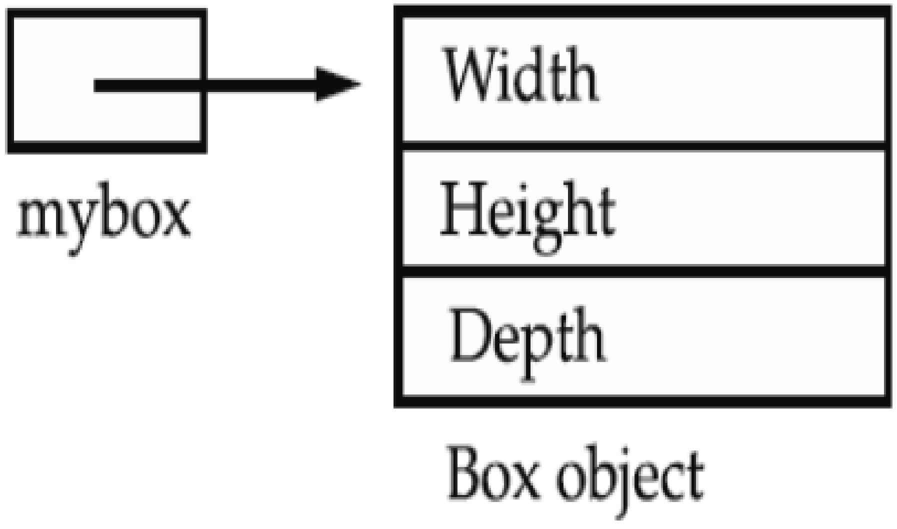
```
Box mybox = new Box();
```

Statement | Effect

```
Box mybox;
```



```
mybox = new Box();
```



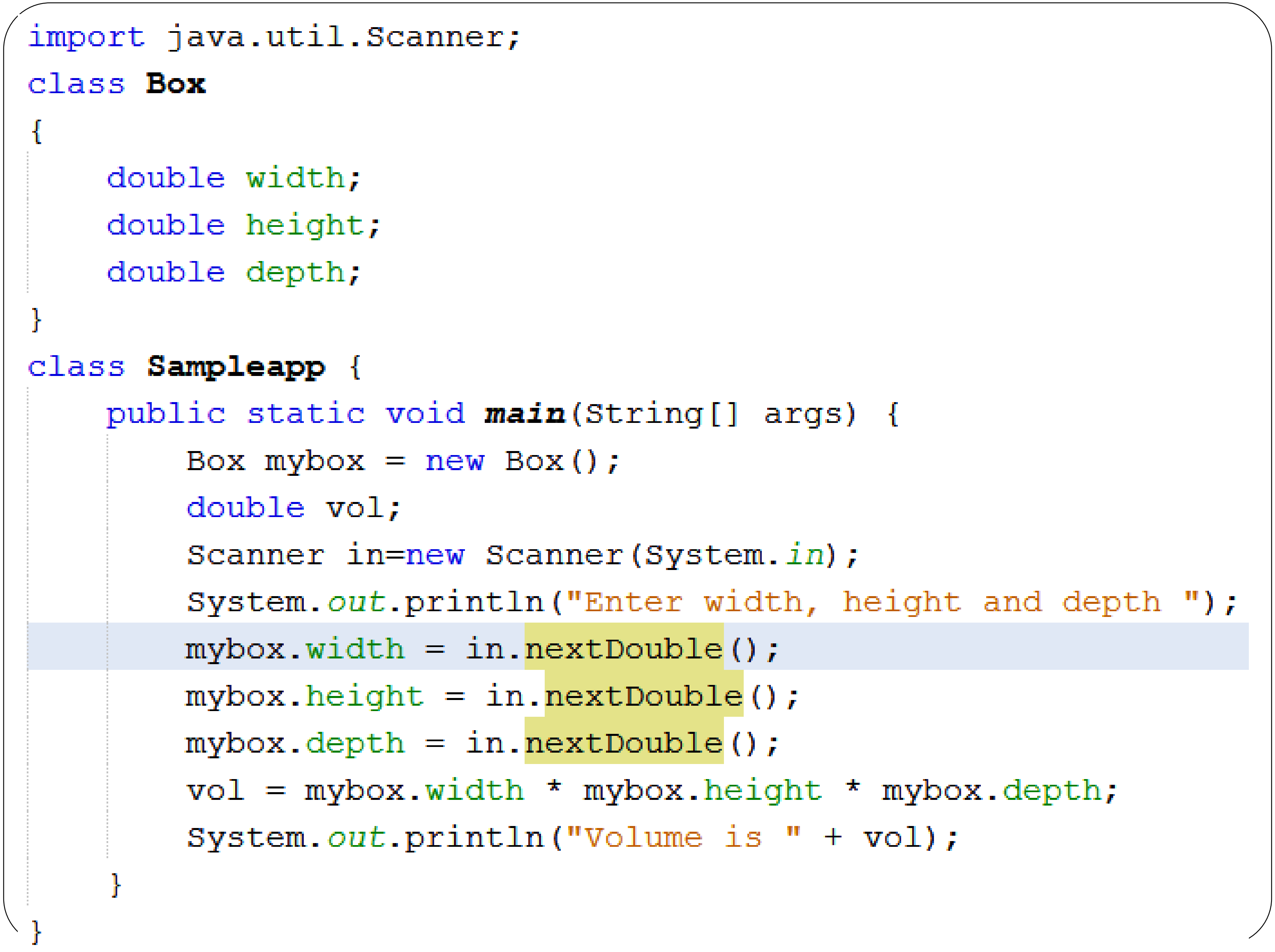
- To access the variables *dot* (.) operator is used.
- Example : assign the **width** variable of **mybox** the value 100

```
mybox.width = 100;
```

- If we have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. Changes to the instance variables of one object have no effect on the instance variables of another.

- Write a program to find the volume of a box, whose dimensions are given

```java
import java.io.*;
class Box
{
    double width;
    double height;
    double depth;
}
class Sampleapp {
    public static void main(String[] args) {
        Box mybox = new Box();
        double vol;
       // assign values to mybox's instance variables
        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);
    }
}
```



```java
import java.util.Scanner;
class Box
{
    double width;
    double height;
    double depth;
}
class Sampleapp {
    public static void main(String[] args) {
        Box mybox = new Box();
        double vol;
        Scanner in=new Scanner(System.in);
        System.out.println("Enter width, height and depth ");
        mybox.width = in.nextDouble();
        mybox.height = in.nextDouble();
        mybox.depth = in.nextDouble();
        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Volume is " + vol);
    }
}
```

Assigning Object Reference Variables

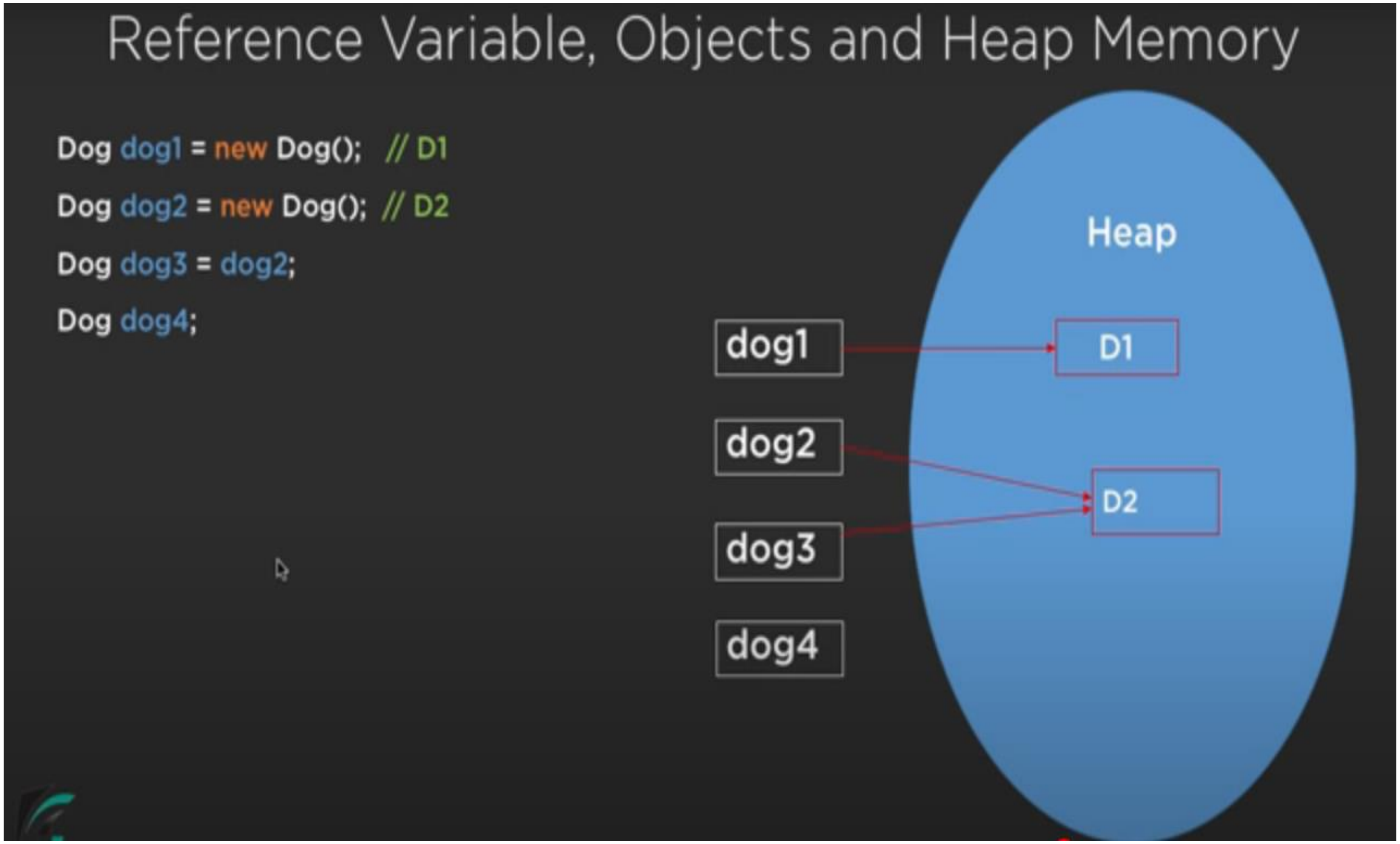
- Example:

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

- A subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**.
- Example:

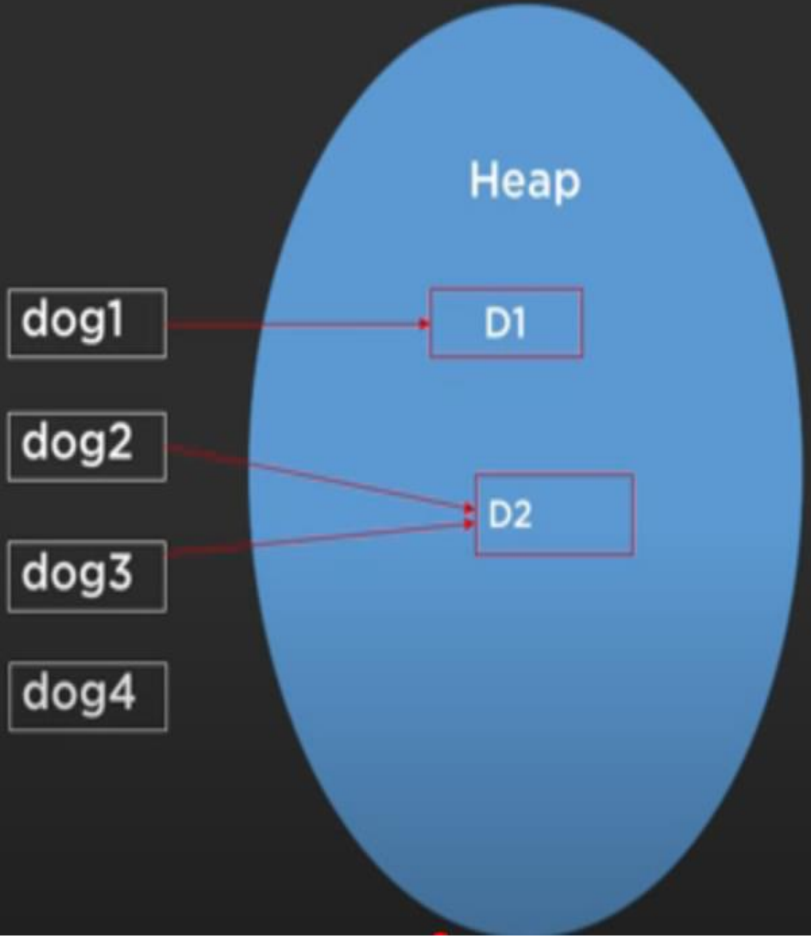
```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

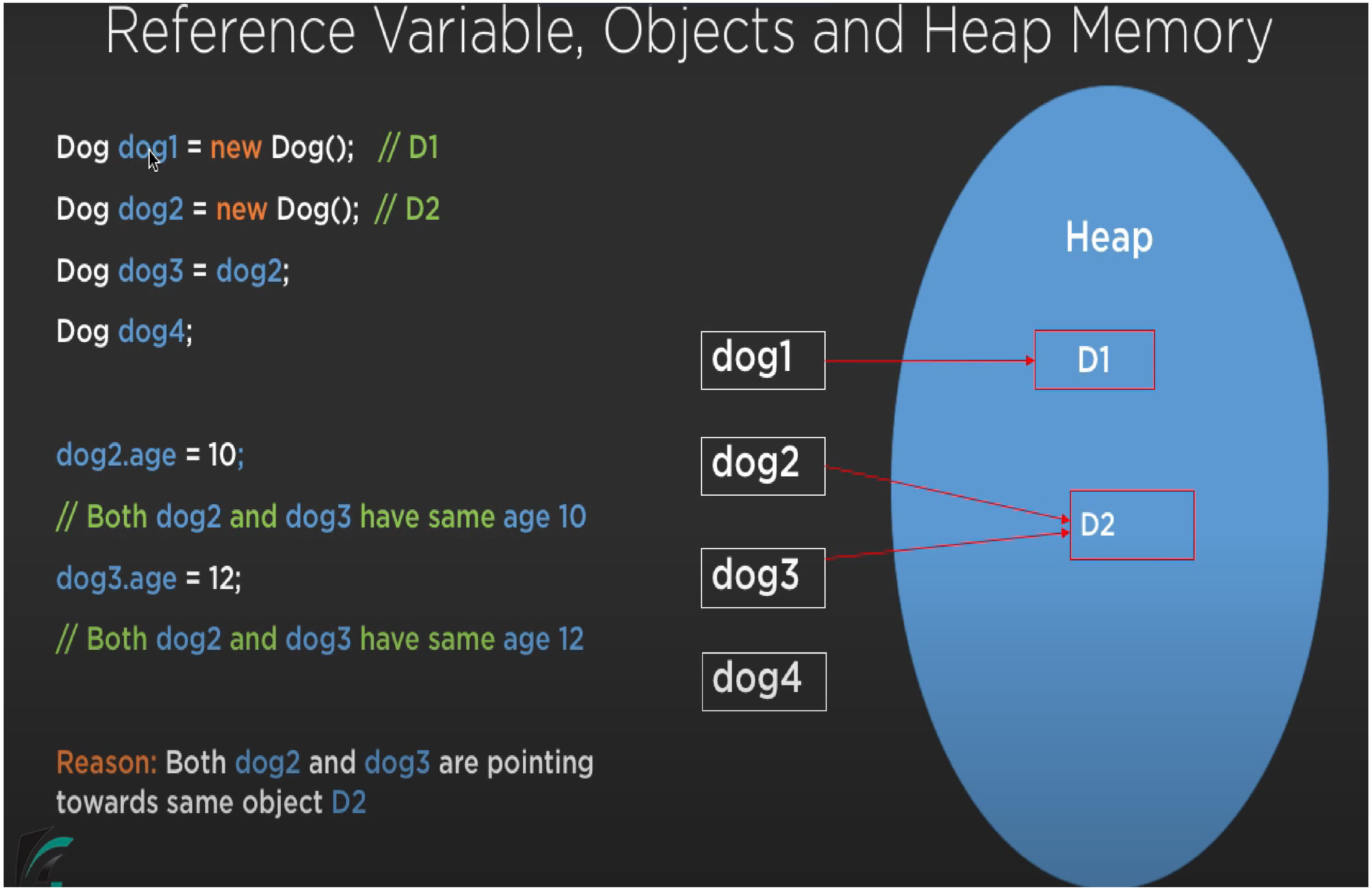
Here, **b1** has been set to **null**, but **b2** still points to the original object.



Reference Variable, Objects and Heap Memory

```
Dog dog1 = new Dog();   // D1
Dog dog2 = new Dog();  // D2
Dog dog3 = dog2;
Dog dog4;
```





Reference Variable, Objects and Heap Memory

```
Dog dog1 = new Dog();   // D1

Dog dog2 = new Dog();  // D2

Dog dog3 = dog2;

Dog dog4;


dog2.age = 10;

// Both dog2 and dog3 have same age 10

dog3.age = 12;

// Both dog2 and dog3 have same age 12
```

Reason: Both dog2 and dog3 are pointing towards same object D2

```
class Sampleapp {
    public static void main(String[] args) {
        Box mybox = new Box();
        Box b2=mybox;
        System.out.println("Enter width, height and depth ");
        mybox.width = 2;
        mybox.height =2;
        mybox.depth = 3;
       b2.width=12;
        System.out.println("Width of box mybox : "+mybox.width);
        System.out.println(“Depth of box mybox : "+b2.depth);
    }
//output
Width of box mybox :12
Depth of box mybox : 3
```

Methods

- Syntax :

```
type name(parameter-list)
{
        // body of method
}
```

- Methods that have a return type other than **void** return a value to the calling routine
  - Syntax :

    ```
    return value;
    ```

  - The type of data returned by a method must be compatible with the return type specified by the method.
  - The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

- The name of the method should be a legal identifier.

Q1)Write a program to find the volume of a box using method.

Q2)Write a program to find the volume of a box (whose dimensions are given) using method with a return type.

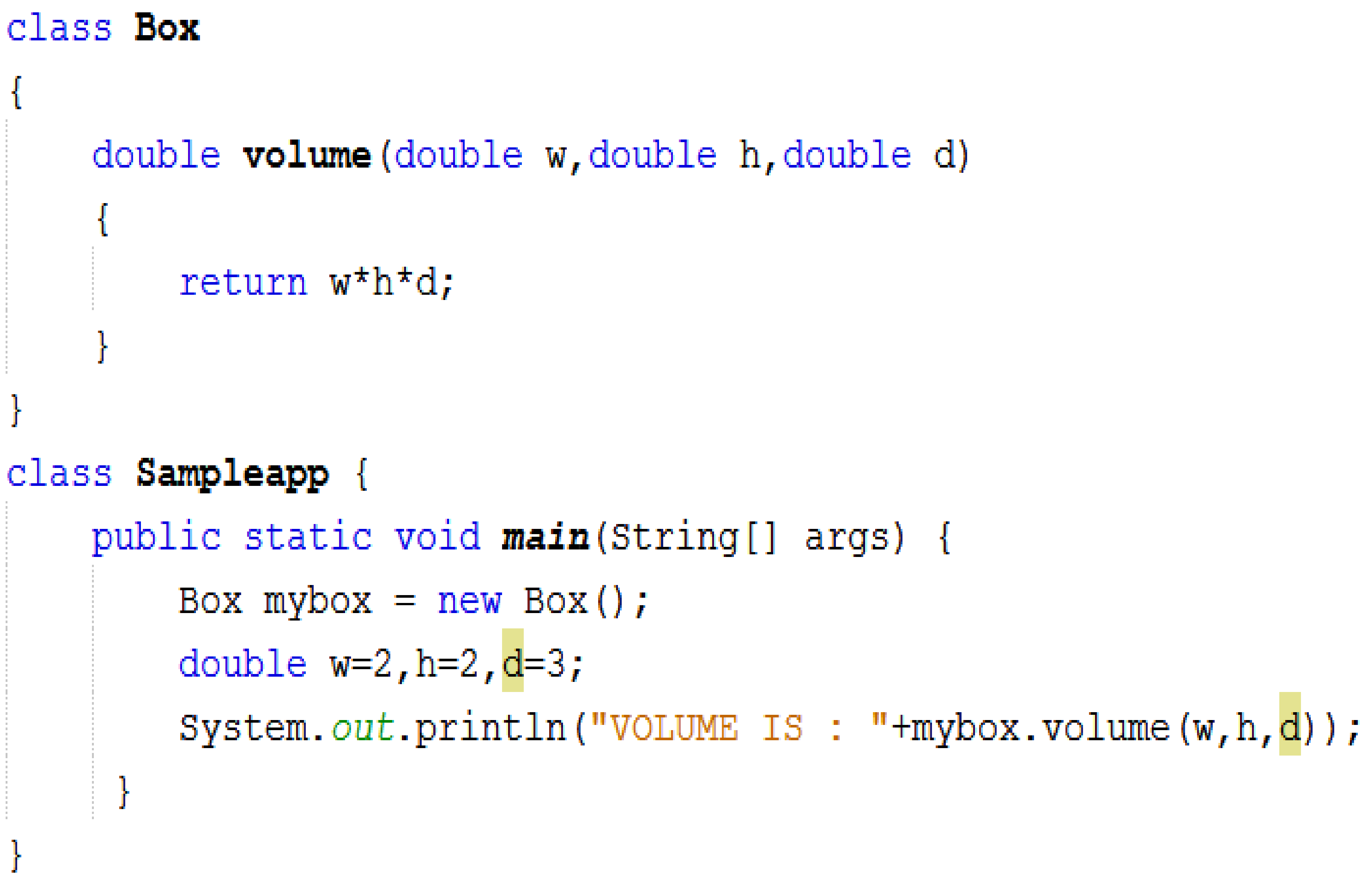
Q3)Write a program to find the volume of a box(whose dimensions are given) using parameterized method.

```java
import java.util.Scanner;
class Box
{
    double width;
    double height;
    double depth;
    void volume()
    {
        double vol;
        vol=width*height*depth;
        System.out.println("VOLUME : "+vol);
    }
}
 class Sampleapp {
     public static void main(String[] args) {
         Box mybox = new Box();
         Scanner in=new Scanner(System.in);
         System.out.println("Enter width, height and depth ");
         mybox.width = in.nextDouble();
         mybox.height = in.nextDouble();
         mybox.depth = in.nextDouble();
         mybox.volume();
     }
 }
```

- Q2

```
class Box
{
    double width; double height; double depth;
    double volume()
    {
        double vol;
        vol=width*height*depth;
        return vol;
    }
}
class Sampleapp {
    public static void main(String[] args) {
        Box mybox = new Box();
        System.out.println("Enter width, height and depth ");
        mybox.width = 2;  mybox.height = 2;
        mybox.depth = 1;
        System.out.println("VOLUME IS : "+mybox.volume());
     }
}
```

- Q3



```
class Box
{
    double volume(double w,double h,double d)
    {
        return w*h*d;
    }
}
class Sampleapp {
    public static void main(String[] args) {
        Box mybox = new Box();
        double w=2,h=2,d=3;
        System.out.println("VOLUME IS : "+mybox.volume(w,h,d));
     }
}
```

TUTORIAL 3

1. Create a class circle with data field radius. Calculate the area and perimeter using methods with return type.

2. Create a class student with fields rollno, name and marks of three subjects. Calculate the total marks and percentage obtained by the student.

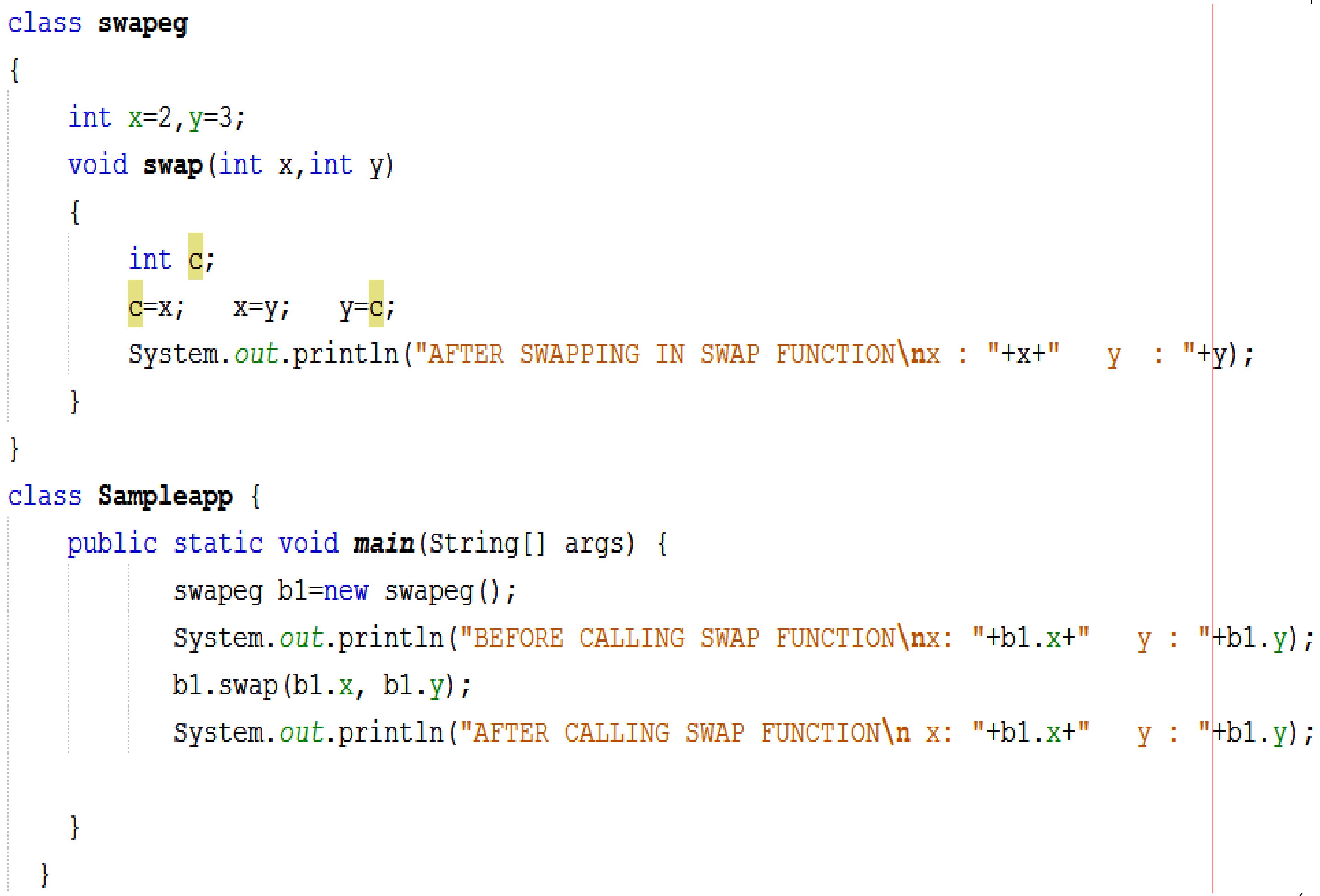
Argument/Parameter Passing

- Two ways to pass an argument to a method :
  - **call-by-value** (or pass by value)
  - **call-by-reference** (or pass by reference)

- Call by value : If we call a method by passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

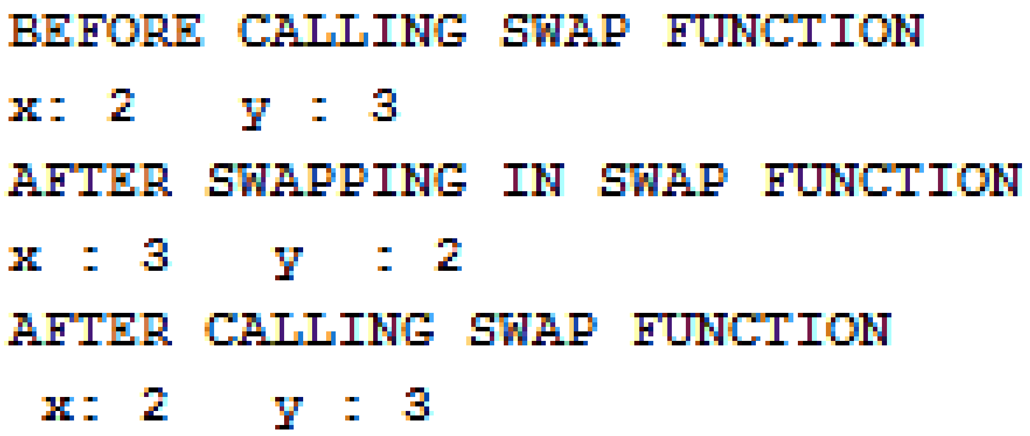
- Call by Reference: Here original value is changed if changes are made in the called method. If we pass object in place of any primitive value, original value will be changed.

Example for Pass by value



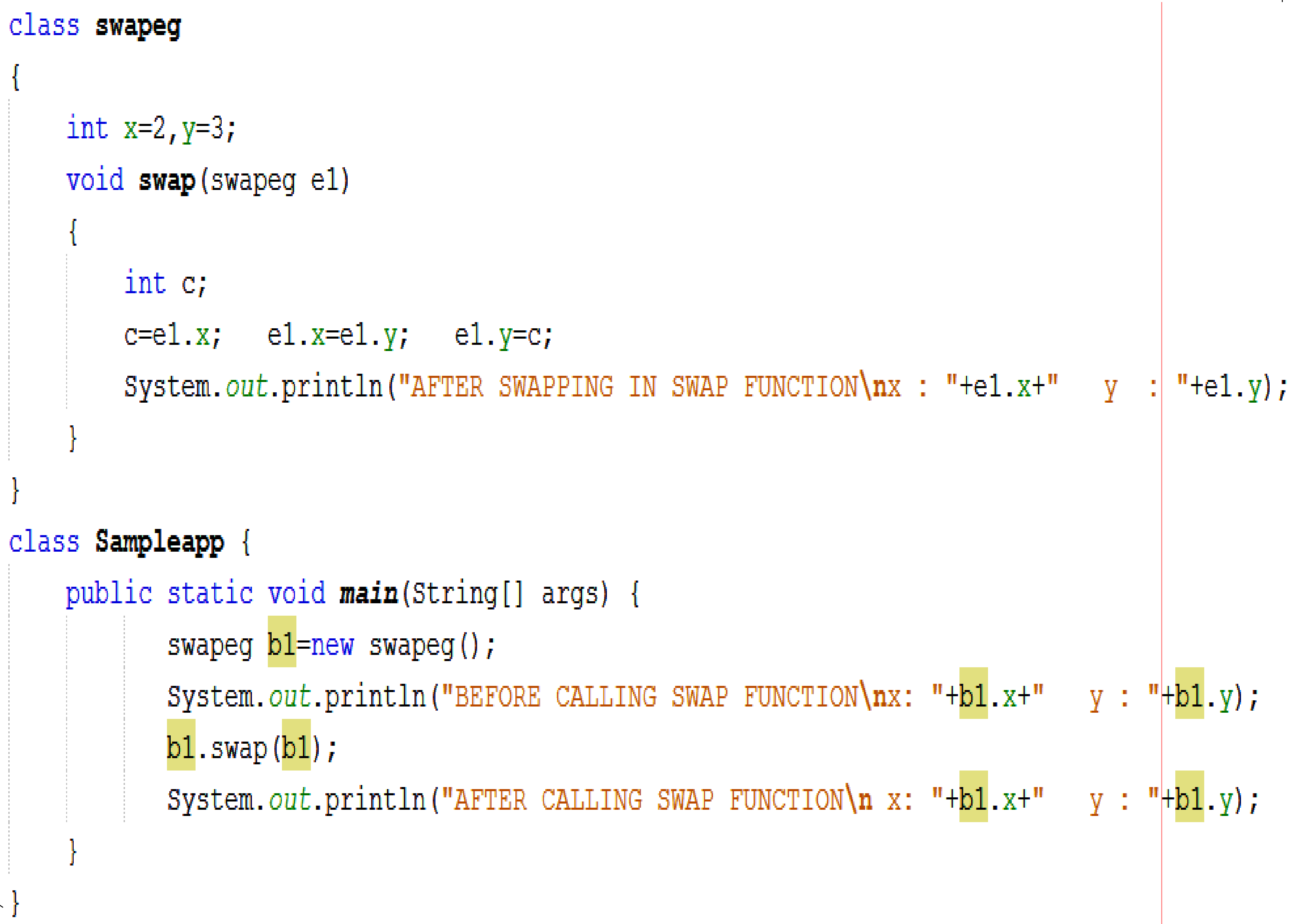
```java
class swapeg
{
    int x=2,y=3;
    void swap(int x,int y)
    {
        int c;
        c=x;   x=y;   y=c;
        System.out.println("AFTER SWAPPING IN SWAP FUNCTION\nx : "+x+"   y  : "+y);
    }
}
class Sampleapp {
    public static void main(String[] args) {
            swapeg b1=new swapeg();
            System.out.println("BEFORE CALLING SWAP FUNCTION\nx: "+b1.x+"   y : "+b1.y);
            b1.swap(b1.x, b1.y);
            System.out.println("AFTER CALLING SWAP FUNCTION\n x: "+b1.x+"   y : "+b1.y);

    }
 }
```

Output:



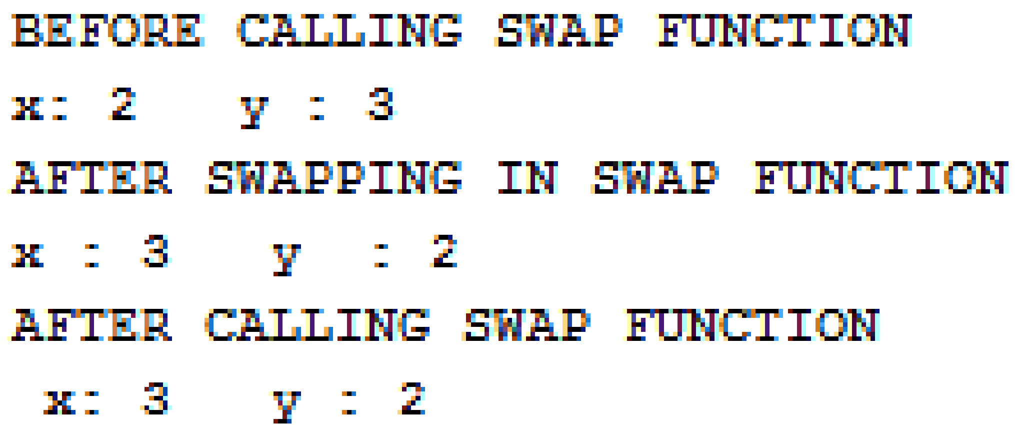
```
BEFORE CALLING SWAP FUNCTION
x: 2    y : 3
AFTER SWAPPING IN SWAP FUNCTION
x : 3    y  : 2
AFTER CALLING SWAP FUNCTION
 x: 2    y : 3
```

- Example for Call By Reference (here object is passed as parameter)



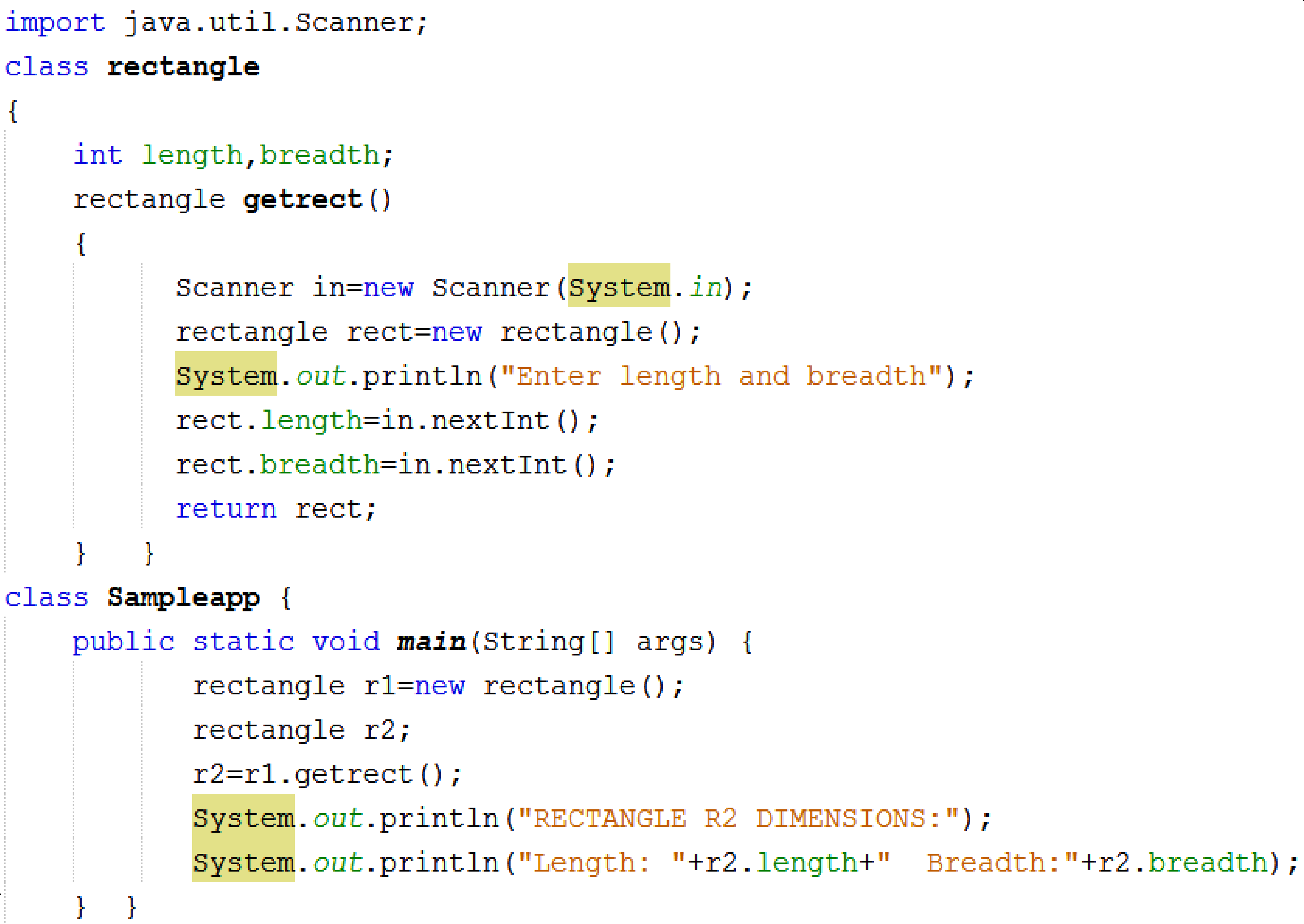
```
class swapeg
{
    int x=2,y=3;
    void swap(swapeg e1)
    {
        int c;
        c=e1.x;   e1.x=e1.y;   e1.y=c;
        System.out.println("AFTER SWAPPING IN SWAP FUNCTION\nx : "+e1.x+"    y  :  "+e1.y);
    }
}
class Sampleapp {
    public static void main(String[] args) {
            swapeg b1=new swapeg();
            System.out.println("BEFORE CALLING SWAP FUNCTION\nx: "+b1.x+"    y : "+b1.y);
            b1.swap(b1);
            System.out.println("AFTER CALLING SWAP FUNCTION\n x: "+b1.x+"    y : "+b1.y);
    }
}
```

Output



```
BEFORE CALLING SWAP FUNCTION
x: 2    y : 3
AFTER SWAPPING IN SWAP FUNCTION
x : 3    y  : 2
AFTER CALLING SWAP FUNCTION
 x: 3    y : 2
```

Returning Objects in Methods



```java
import java.util.Scanner;
class rectangle
{
    int length,breadth;
    rectangle getrect()
    {
            Scanner in=new Scanner(System.in);
            rectangle rect=new rectangle();
            System.out.println("Enter length and breadth");
            rect.length=in.nextInt();
            rect.breadth=in.nextInt();
            return rect;
    }   }
class Sampleapp {
    public static void main(String[] args) {
             rectangle r1=new rectangle();
             rectangle r2;
             r2=r1.getrect();
             System.out.println("RECTANGLE R2 DIMENSIONS:");
             System.out.println("Length: "+r2.length+"  Breadth:"+r2.breadth);
    }  }
```

Constructors

- Used to initialize objects when they are created.
- It has the same name as the class in which it resides.
- It is syntactically similar to a method.
- The constructor is automatically called immediately after the object is created.
- Constructors have no return type.

- Java supports 2 types of constructors:
- Default Constructor - does not have an argument.
- Parameterized Constructor - A constructor having parameter is called parameterized constructor.

Default Constructor:

- A default constructor does **not have an argument**.
- It provides **default values** to the object.
- If no constructor is designed by the user explicitly, java compiler implicitly provides a default constructor.
    - JVM calls that system defined default constructor at the time of object creation.
- If we define a default constructor, JVM will call the user-defined default constructor.

Parameterized Constructor:

- The parameterized constructor **has parameters**.
  - The values for these parameters must be provided when the constructor is called.
- This type of constructor is used to provide **different values** for the data members of different objects.
- The programmer has to explicitly define such a constructor. Java compiler never provides any parameterized constructor, unlike default constructor.
- Calling the parameterized constructor without defining it will generate a compile-time error.

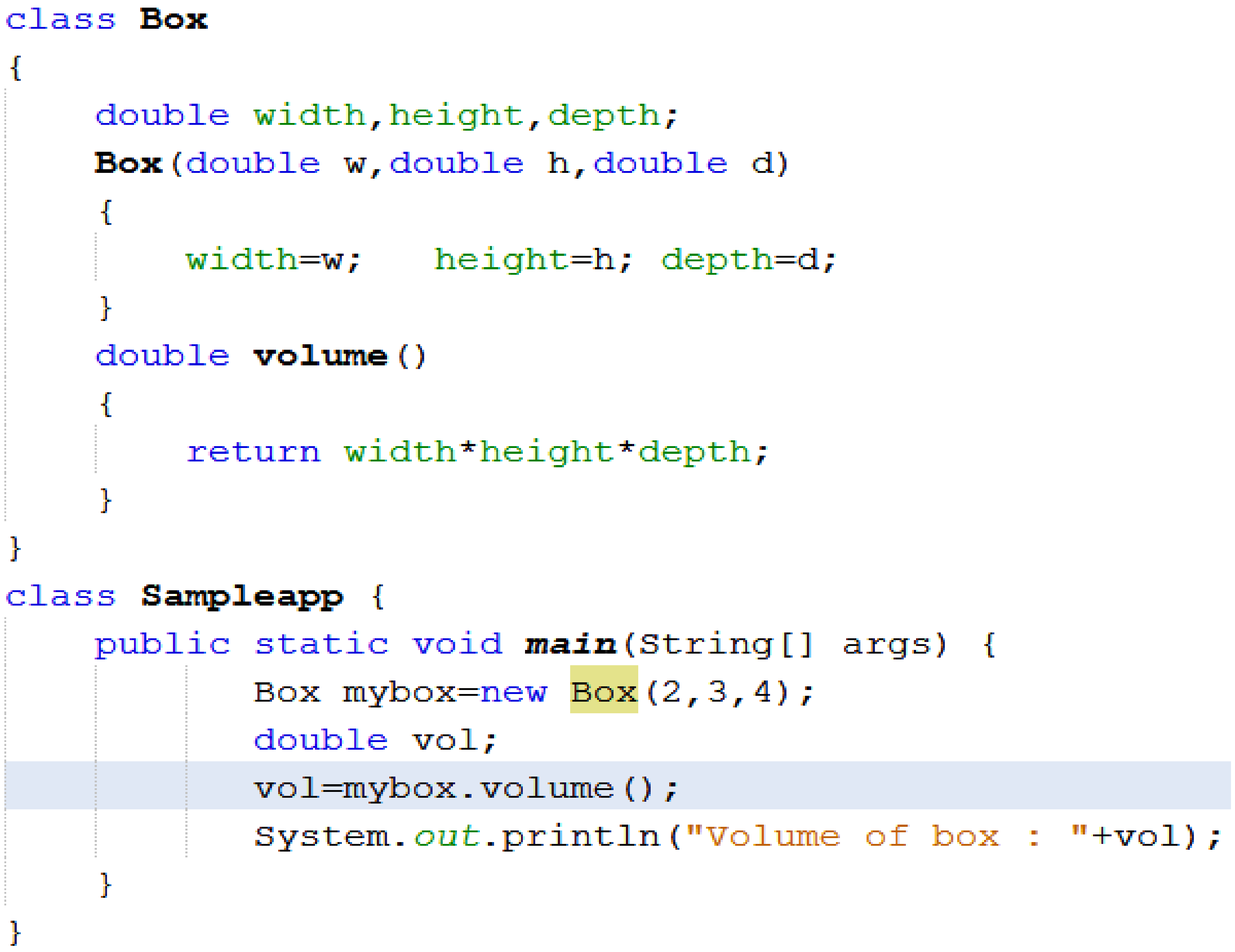
Q1. Write a program to find the volume of a box using constructor

Q2. Write a program to find the volume of a box using parameterized constructor

Q1 (Example of Default Constructor)

```
class Box
{
    double width,height,depth;
    Box()
    {
        width=2;    height=2; depth=3;
    }
    double volume()
    {
        return width*height*depth;
    }
}
class Sampleapp {
    public static void main(String[] args) {
            Box mybox=new Box();
            double vol;
            vol=mybox.volume();
            System.out.println("Volume of box :  "+vol);
    }
}
```

Q2 (Example of Parameterized Constructor)



```
class Box
{
    double width,height,depth;
    Box(double w,double h,double d)
    {
        width=w;   height=h; depth=d;
    }
    double volume()
    {
        return width*height*depth;
    }
}
class Sampleapp {
    public static void main(String[] args) {
            Box mybox=new Box(2,3,4);
            double vol;
            vol=mybox.volume();
            System.out.println("Volume of box : "+vol);
    }
}
```

Overloading

- **Overloading** refers to the ability of a class to have multiple constructors or multiple methods with the same name but with different number or type of arguments list.

- Overloading is a way by which java implements polymorphism.

- There are two types :
  - Method Overloading
  - Constructor loading

Method Overloading

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

- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.

- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

- Q)Write a program to display a character, integer number and a floating point number using the concept of method overloading

```
class DisplayOverloading2  {
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(int c)
    {
       System.out.println(c );
    }
    public void disp(float c)
    {
       System.out.println(c );
    }   }
class Sample2{
    public static void main(String args[])
    {
        DisplayOverloading2 obj = new DisplayOverloading2();
        obj.disp('a');
        obj.disp(5);                                    o/p  a
        obj.disp(2.5f);                                      5
    }    }                                                   2.5
```

Constructor Overloading

- Constructor methods can also be overloaded

```
class Box
{     double width, height,depth;
Box(double w, double h, double d)
{       width = w;  height = h;  depth = d;
}
Box()
{       width = -1;  height = -1;  depth = -1;
}
Box(double len)
{        width = height = depth = len;
}
double volume()
{       return width * height * depth;
}
}
```

```
class OverloadCons
{
    public static void main(String args[])
    {
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
         double vol;
         vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " +
vol);
          vol = mybox2.volume();
         System.out.println("Volume of mybox2 is " +
vol);
        vol = mycube.volume();
        System.out.println("Volume of mycube is " +
vol);     /* o/p Volume of mybox1 is 3000.0
}                Volume of mybox2 is -1.0
}                Volume of cube is 343.0 */
```

this Keyword

- In Java, “**this**” is a reference variable that always refers to the current object. “**this**” can be used to resolve any name space collisions that might occur between instance variables and local variables.

We can use “this” keyword as follows:

- in the current class to
  1. refer to the instance variable
  2. invoke the constructor
  3. invoke the method
- can be passed as an argument in a method call
- can be passed as an argument in a constructor call
- to return the current class instance

1) this: to refer current class instance variable (Problem if we don't use 'this')

```
class Student{
int rollno;
String name;
Student(int rollno,String name){
        rollno=rollno;
        name=name;          }
void display(){System.out.println(rollno+" "+name);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit");
Student s2=new Student(112,"sumit");
s1.display();                                  o/p  0  null
s2.display();                                       0  null
}}
```

1) this: to refer current class instance variable (Solution using 'this')

```
class Student{
int rollno;
String name;
Student(int rollno,String name){
        this.rollno=rollno;
        this.name=name;     }
void display(){System.out.println(rollno+" "+name);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit");
Student s2=new Student(112,"sumit");
s1.display();                                   o/p  111  ankit
s2.display();                                        112  sumit
}}
```

2) this: to invoke current class method (Example 1)

```
class Test {
    void display()
    {
        // calling function show()
        this.show();


       System.out.println("Inside display function");
    }
    void show() {
        System.out.println("Inside show funcion");
    }
    public static void main(String args[]) {
       Test t1 = new Test();
        t1.display();
    }
}
```

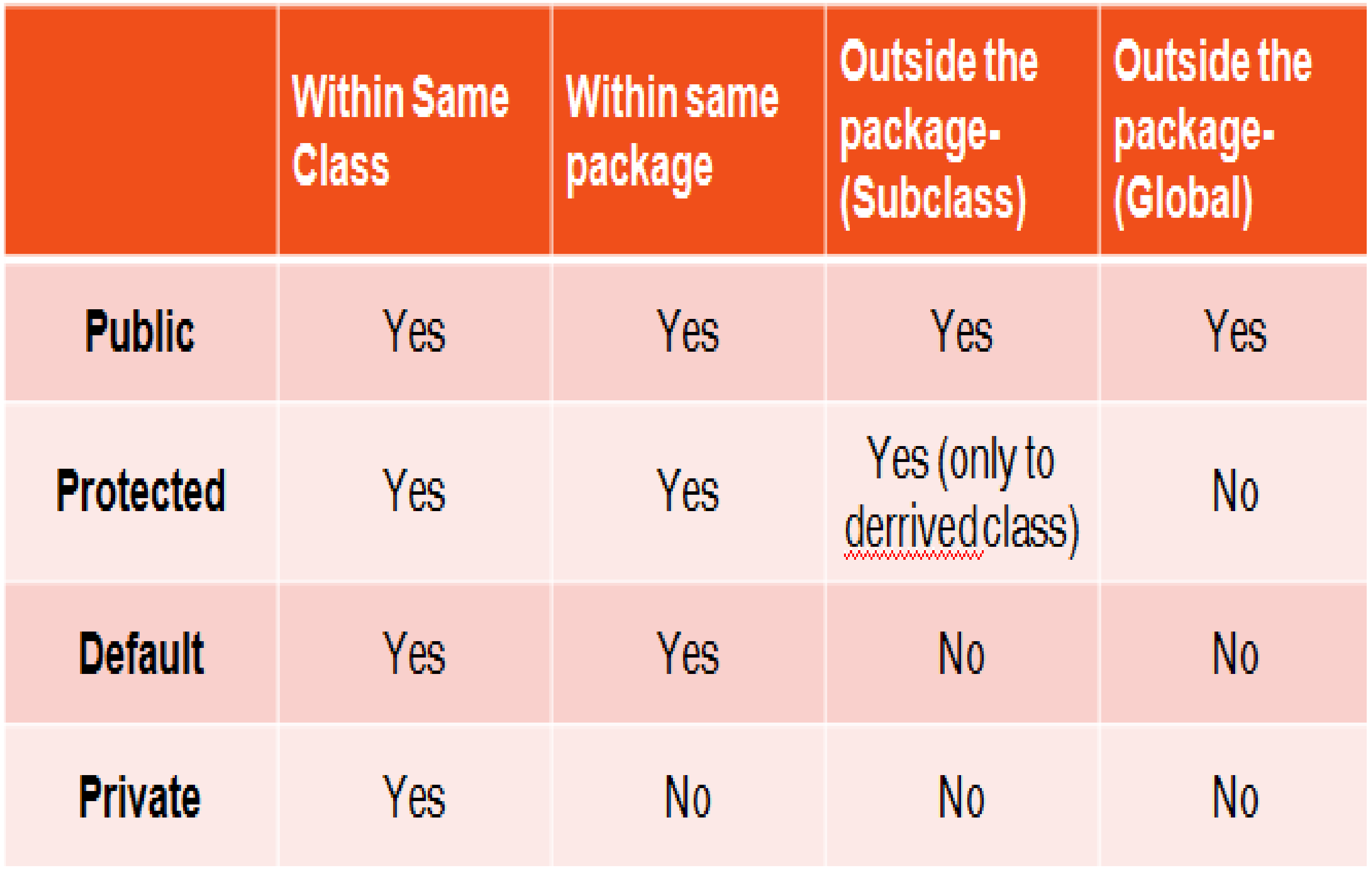
o/p   Inside show funcion
Inside display function

3) this() : to invoke current class constructor (Example 1)

```
/*The this() constructor call can be used to invoke the current class constructor.
   It is used to reuse the constructor. In other words, it is used for constructor
   chaining.*/
class A{
        A()
        {  System.out.println("hello a");  }            o/p   hello a
        A(int x){                                              10
        this();
        System.out.println(x);
        }
    }
class TestThis5{
                public static void main(String args[]){
                A a=new A(10);
            }}
```

Access Control

- The job of access specifier is to specify the **scope of a variable** (data member), function (method), constructor or any class.
- Access specifiers in Java are :
  - **Public :** This member can be accessed by any other code
  - **Private :** This member can only be accessed by other members of its class.
  - **Protected** : It is applied only when inheritance is involved.
  - Default: Members of the same class as well as members of a different class within the same package can access the default members.
- When no access specifier is used, then by default the member of a class is **public** within its own package, but cannot be accessed outside of its package.



	Within Same Class	Within same package	Outside the package- (Subclass)	Outside the package- (Global)
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes (only to derrived class)	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

```
class Test
{
    int a;
    public int b;
    private int c;
    void setc(int i)
    {
        c = i;
    }
    int getc()
    {
        return c;
    }
}
```

```
class AccessTest
{
    public static void main(String args[])
    {
        Test ob = new Test();
        // These are OK, a and b may be accessed directly
        ob.a = 10;
        ob.b = 20;
        // This is not OK and will cause an error
        // ob.c = 100; // Error!
        // You must access c through its methods
        ob.setc(100); // OK
        System.out.println("a, b, and c: " + ob.a + " "
                                      +  ob.b + " " + ob.getc());
    }
}
```

Tutorial 4

1. **Write a java program to design a class to represent a bank account. Include the following members:**

   Data members: name of depositor, Account number, Type of Account, Balance amount in account.

   Methods : To assign initial values(use constructor), To deposit an amount, To withdraw an amount after checking balance, To display the name and balance.

2. **Is there error in the following code. If yes then correct it.**

```
public class Figure
{
    public String draw(String s){
        return "figure drawn"; }
    public void draw(String s){}
    public void draw(double f){}
}
```

static keyword

- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- Example : **main( )**

- Variables, methods, blocks and nested classes can be declared as **static**.
- Example : static int i;

  static void add(){ ……. }
- 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 class share the same **static** variable.

Advantage – Makes the program memory efficient.

Advantage of static variable (Memory efficient)

```
class Student{
    int rollno;
    String name;
    String college=“VJC";  }
Student(int r, String n,String c)
        {
  rollno = r;
  name = n;
  college=c;
           }
                  }
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,“Amy“,”VJC”)
 Student s2 = new Student(222,"Anu“,”VJC”);
  }}
```

```
class Student{
    int rollno;
    String name;
    static  String college=“VJC";  }
Student(int r, String n)
        {
  rollno = r;
  name = n;
           }
                  }
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,“Amy“);
 Student s2 = new Student(222,"Anu“);
  }}
```

- Properties of **static** Methods :
  - They can only call other static methods.
  - They must only access static data.
  - They cannot refer to this or super in any way.

- Syntax for calling static methods and variables from outside of the class is : *classname.method( );* **//access static methods**

  classname.var_name ; **//access static variables**

- **Static block** : It is defined inside a class and can be used to initialize static variables, which gets executed exactly once, when the class is first loaded before main.

- Syntax :

```
static
{…………
}
```

Example 1

```
class StaticDemo {
     static int a = 42;
       static int b ;
       static
       {       b=a*2;
        }
        static void callme()
       {     System.out.println("a = " + a);
        }              }
class StaticByName
{     public static void main(String args[])
       {
              StaticDemo.callme();
              System.out.println("b = " + StaticDemo.b);
        }
}
```

Example 2

```
class Staticdemo{
public static void main(String args[])
{
Counter c1=new Counter();
c1.incrementCounter();
Counter c2=new Counter();
c2.incrementCounter();
}                    }
class Counter{
static int c=0;
public void incrementCounter(){
c++;
System.out.println(c);                    o/p  1
}                  }                             2
```

Example 3(static block)

```
class Staticblock {
static int a=10;
static int b;
static{   //static block
System.out.println("Static block");
b=a*4;
}
public static void main(String args[])
{
System.out.println("from main");
System.out.println("Value of a : "+a+ "and b : "  +b);
}
}
```

o/p

```
Static block
from main
Value of a: 10 and b: 40
```

Example 4(static nested class)

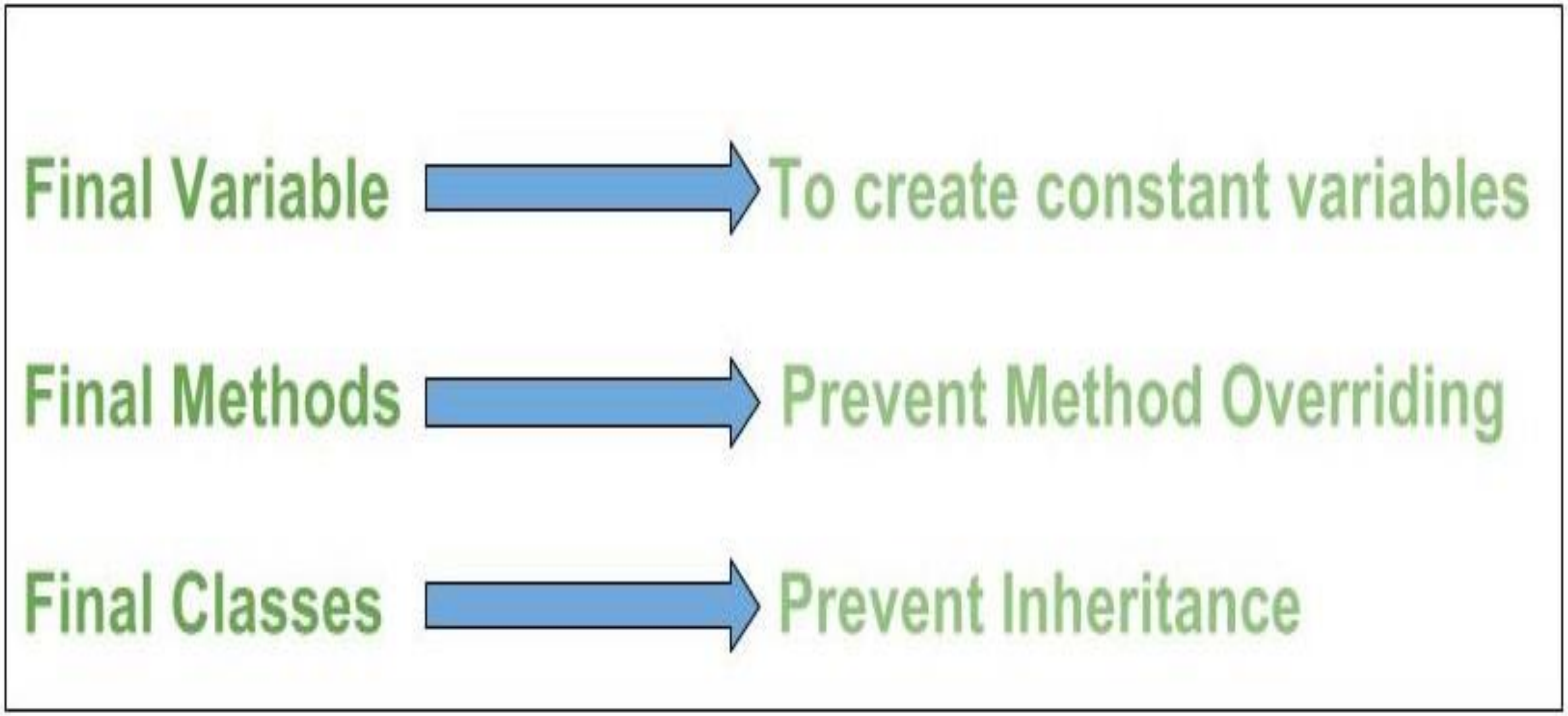
```
public class Outer {
static class Nested_Demo {
   public void my_method()
  {
 System.out.println("This is my nested class");
}
}                                              o/p This is my nested class
public static void main(String args[])
{
 Outer.Nested_Demo nested = new Outer.Nested_Demo();
   nested.my_method();
} } /*You cannot use the static keyword with a class unless it is an
   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. */
```

final Variables

- The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. **variable**
2. **method**
3. **class**

A final variable that have no value is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.


Final Variable
To create constant variables
Final Methods
Prevent Method Overriding
Final Classes
Prevent Inheritance

Final variables

- If a variable is declared with the final keyword, its value cannot be changed once initialized.

Example

```
class ExFinalVariable
 {
          final int var = 50;
          var = 60 //This line would give an error
   }
```

Final methods

A method, declared with the final keyword, cannot be *overridden* or *hidden* by subclasses.

Example

```
class Base{
    public final void finalMethod(){
    System.out.print("Base");
  }     }
  class Derived extends Base{
 public final void finalMethod() { //Overriding the final method throws an
                                                                    error
    System.out.print("Derived");
  }
   }
```

Final classes

- A class declared as a final class, cannot be subclassed

Example

```
// declaring a final class
final class FinalClass {
 //...
         }
class Subclass extends FinalClass
{ //attempting to subclass a final class throws an error

 //...
}
```

Final Variables, Methods and Classes

- The keyword **final** has three uses.
  - **Equivalent to named constants** :
    - The value of the final variable can never be changed.
    - Final variables behave like class variables and they do not take any space on individual objects of the class.
    - Example : **final int size=100;**
  - **To Prevent Overriding** :
    - To prevent the subclasses from overriding the members of the super class, we can declare them as final in its super class using the keyword **final**.
  - **To Prevent Inheritance :**
    - Precede the class declaration with **final**.
    - Declaring a class as **final** implicitly declares all of its methods as **final**, too.
    - It is illegal to declare a class as both **abstract** and **final**

Inner classes

- In Java, just like methods, a class too can have another class as its member. The class written within is called the **nested class**, and the class that holds the inner class is called the **outer class**.
- Nested classes are divided into two types −
- **Inner classes or Non-static nested classes** − These are the non-static members of a class.
- **Static nested classes** − These are the static members of a class.

Example – inner classes(non-static nested classes)

```
class Outer {
  // Simple nested inner class
          class Inner {
     public void show() {
         System.out.println("In a nested class method");
                                  }
                        }
                 }
class Nested {
  public static void main(String[] args) {
     Outer.Inner in = new Outer().new Inner();
     in.show();
  }       }
```

o/p In a nested class method

Java Command line argument

- The command line argument is the argument that passed to a program during runtime.

- It is the way to pass argument to the main method in Java.

- These arguments store into the String type **args** parameter which is main method parameter.

- To access these arguments, you can simply traverse the args parameter in the loop or use direct index value because args is an array of type String.

Example

```
class Cmdline
{
 public static void main(String[] args)
{
for(int i=0;i< args.length;i++)
 {
System.out.println(args[i]);
 }          //To run the program -    java Cmdline hello world
}                             o/p        hello
}                                         world
```

Variable Length Argument (Varargs)

- The varargs allows the method to accept zero or muliple arguments.

- Variable length arguments are most useful when the number of arguments to be passed to the method is not known beforehand.

- Variable number of arguments are represented by three dotes (…)

Advantage of Varargs:

- We don't have to provide overloaded methods, so less code.

Example 1

```
class Varagex1 {
int sum(int…num)  {
int total=0;
for(int x:num)
{
total=total+x;
  }
return total;
                              }
public static void main(String args[]){
int s;
Varagex1 e=new Varagex1();
s=e.sum(1,2,3);
System.out.println("The sum is " +s);
}                 }
```

o/p The sum is 6

Example 2

```
class Demo{
public static void Varar(String…str)
{
System.out.println("num of args" +str.length);
System.out.println("The arguments are ");
for(String a:str)
System.out.println(a);
}
public static void main(String args[])
{
Varar("Apple","mango");
Varar("Pear");
}
}
```

o/p

```
num of args 2
The arguments are
Apple
mango
num of args 1
The arguments are
Pear
```

Recursion in Java

- Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.
- It makes the code compact.

Example 1: Factorial

```
public class RecursionExample {
static int factorial(int n){
 if (n == 1)
return 1;
else
 return(n * factorial(n-1));
 }
public static void main(String[] args) {
System.out.println("Factorial of 5 is: "+factorial(5));
}
}
```

Working of above program:

```
factorial(5)
   factorial(4)
           factorial(3)
                     factorial(2)
                               factorial(1)
                               return 1
                    return 2*1 = 2
          return 3*2 = 6
   return 4*6 = 24
return 5*24 = 120
```

Example 2: Fibonacci Series

```
public class RecursionExample {
   static int n1=0,n2=1,n3=0;
    static void printFibo(int count)
{
      if(count>0){
          n3 = n1 + n2;
          n1 = n2;
          n2 = n3;
          System.out.print(" "+n3);

          printFibo(count-1);
       }
    }
```

```
public static void main(String[] args) {
    int count=5;
      System.out.print(n1+" "+n2);//printing 0
 and 1
      printFibo(count-2);
//count-
2 because 2 numbers are already printed
}
}
```

o/p   0 1 1 2 3

Arrays

- An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index.
- **One-Dimensional Arrays**
- A one-dimensional array is a list of like-typed variables. In Java all arrays are dynamically allocated.
- Declaration and memory allocation can be done in Two steps
- First, declare a variable of the desired array type. Second, allocate the memory that will hold the array, using **new** (special operator that allocates memory), and assign it to the array variable.
- **Syntax 1:** **Example**

***type var-name*[ ];**	int month_days[];
***var-name* = new *type*[*size*];**	month_days = new int[12];

- **Syntax 2:** **Example**

  ***type[] var-name*;** int[] month_days;

  ***var-name* = new *type*[*size*];** month_days = new int[12];

- It is possible to combine the declaration of the array variable with the allocation ofthe array itself
- **Syntax 1:** **Example**

***type var-name*[ ] = new *type*[*size*];** int month_days[] = new int[12];

- **Syntax 2**:

***type[] var-name* = new *type*[*size*];** int[] month_days = new int[12];

- Arrays can be initialized when they are declared. An *array initializer* is a list of comma-separated expressions surrounded by curly braces.

Eg) int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

Example

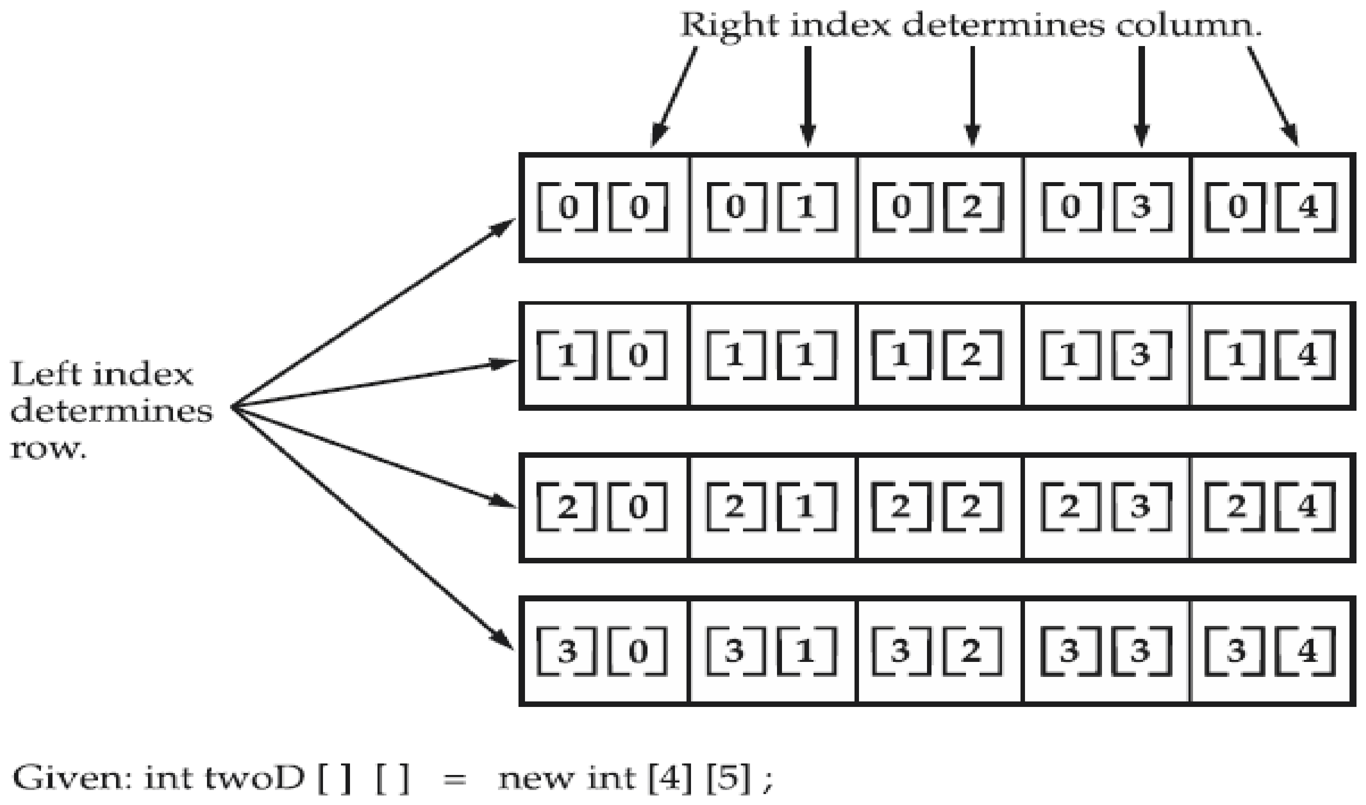
```
// Average an array of values.
class Average
{
public static void main(String args[])
{
double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
double result = 0;
int i;
for(i=0; i<5; i++)
result = result + nums[i];
System.out.println("Average is " + result / 5);
}
}
```

Multidimensional Arrays

- In Java, *multidimensional arrays* are actually arrays of arrays.
- Declaration Example :

int twoD[][] = new int[4][5]; **OR** int[][] twoD = new int[4][5];

- This allocates a 4 by 5 array and assigns it to **twoD**.



Example: Print the elements in the matrix

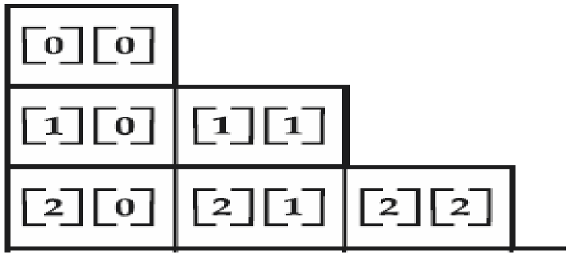
```
class TwoDArray{
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;                    }
for(i=0; i<4; i++)   {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();   }
}  }
```

- When you allocate memory for a multidimensional array, we need only specify the memory for the first (leftmost) dimension. we can allocate the remaining dimensions separately.

Example

```
int twoD[][] = new int[3][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
```

This is how the array looks like



- It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces.
- Expressions as well as literal values can be used inside array initializer.

Example

```
int m[][] = { { 0, 1, 2, 3 },
              { 4, 5, 6, 7 },
              {8,9,10,11} };
```

Vector class

- Vector implements a dynamic array that means it can grow or shrink as required.
- Like an array, it contains components that can be accessed using an integer index.
- It can hold objects of any type and any number.
- Vector class is contained in java.util package.
- It is similar to ArrayList, but with two differences –
  1. Vector is synchronized.
  2. Vector contains many legacy methods that are not part of the collections framework.
- It extends **AbstractList** and implements **List** interfaces.

- Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

<u>Constructors of Vector</u> – Vector class provides 4 constructors

1. **Vector()**: Creates a default vector of initial capacity is 10.
2. **Vector(int size):** Creates a vector whose initial capacity is specified by size.
3. **Vector(int size, int incr):** Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time that a vector is resized upward.
4. **Vector(Collection c):** Creates a vector that contains the elements of collection c.

Important points regarding Increment of vector capacity:

If increment is specified, Vector will expand according to it in each allocation cycle but if increment is not specified then vector's capacity get doubled in each allocation cycle. Vector defines three protected data member:

- **int capacityIncreament:** Contains the increment value.
- **int elementCount:** Number of elements currently in vector stored in it.
- **Object elementData[]:** Array that holds the vector is stored in it.

Methods in Vector:

boolean add(Object obj): This method appends the specified element to the end of this vector.

Example

```
import java.util.*;
class Vector_demo {
    public static void main(String[] arg)
    {
        Vector v = new Vector(); // create default vector
        v.add(1);
        v.add(2);
        v.add("hello");
        v.add("world");
        v.add(3);
        System.out.println("Vector is " + v);
    }   }
```

o/p Vector is [1, 2, hello, world, 3]

void add(int index, Object obj): This method inserts the specified element at the specified position in this Vector.

Example

```
import java.util.*;
class Vector_demo {
    public static void main(String[] arg)   {
        Vector v = new Vector();  // create default vector
        v.add(0, 1);
        v.add(1, 2);
        v.add(2, “hello");
        v.add(3, “world");
        v.add(4, 3);
        System.out.println("Vector is " + v);
    }
}
```

Output:

```
Vector is: [1, 2, hello, world, 3]
```

Commonly used methods of Vector Class:

- **void addElement(Object element):** It inserts the element at the end of the Vector.
- **int capacity():** This method returns the current capacity of the vector.
- **int size():** It returns the current size of the vector.
- **void setSize(int size):** It changes the existing size with the specified size.
- **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.
- **boolean containsAll(Collection c):** It returns true if all the elements of collection c are present in the Vector.
- **Object elementAt(int index):** It returns the element present at the specified location in Vector.
- **Object firstElement():** It is used for getting the first element of the vector.
- **Object lastElement():** Returns the last element of the array.
- **Object get(int index):** Returns the element at the specified index.

Contd..

- **boolean isEmpty():** This method returns true if Vector doesn’t have any element.
- **boolean removeElement(Object element):** Removes the specifed element from vector.
- **boolean removeAll(Collection c):** It Removes all those elements from vector which are present in the Collection c.
- **void setElementAt(Object element, int index):** It updates the element of specifed index with the given element.