JAVA PROGRAMMING

UNIT - III

Dr Yogish H K, Professor, Dept. of ISE, RIT, Bengaluru

UNIT - III

Syllabus

Introducing Classes: Class Fundamentals, Declaring Objects, A Closer Look at new, Assigning Object Reference Variables, Introducing Methods, Constructors, Parameterized Constructors, The this Keyword, Instance Variable Hiding, Garbage Collection, The finalize Method.

A Closer Look at Methods and Classes: Overloading Methods, Using Objects as Parameters, A Closer Look at Argument Passing, Returning Objects, Introducing Access Control, Understanding static, Introducing final

Classes and Objects

- Classes and objects are the two main aspects of object-oriented programming.
- Java is True object-oriented programming language and therefore all programs in Java are written by classes.
- The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- A **class** is a user defined data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a ***template*** for an object, and an **object** is an *instance* of a class.

Classes and Objects ….

- Look at the following illustration to see the difference between class and objects:
- **Class - Fruit**
- **Objects - Apple, Mango, Banana**
- **Class - Car**
- **Objects – Maruti, Audi, Toyota**
- So, a class is a template for objects, and an object is an instance of a class.
- When the individual objects are created, they inherit all the variables and methods from the class.

A class is declared by use of the **class** keyword.
A simplified general form of a class definition is:


```
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
                   }
}
```

Continue …

- The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*.
- Collectively, the methods and variables defined within a class are called *members* of the class.

- Variables defined within a class are called *instance variables* because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

A Simple Class

- Here is a class called Box that defines three instance variables: width, height, and depth.

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

- As stated, a class defines a new type of data. In this case, the new data type is called **Box**.
- Use this name to declare **objects** of type **Box**. It is important to remember that *a class declaration only creates a template; it does not create an actual object*.

Declaring Objects

- When you create a class, you are creating a new data type. You can use this type to declare objects of that type.
- Declaring objects of a class is a two-step process.
- First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator. The new operator dynamically allocates memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new.

Syntax:

- The **new** operator dynamically allocates memory for an object. It has the following general form:

  class-var = new classname( );

- *class-var* is a variable of the class. i.e. Object of class.
- The *classname* is the name of the class that is being instantiated.
- The class name followed by parentheses specifies the *constructor* for the class.

Example:

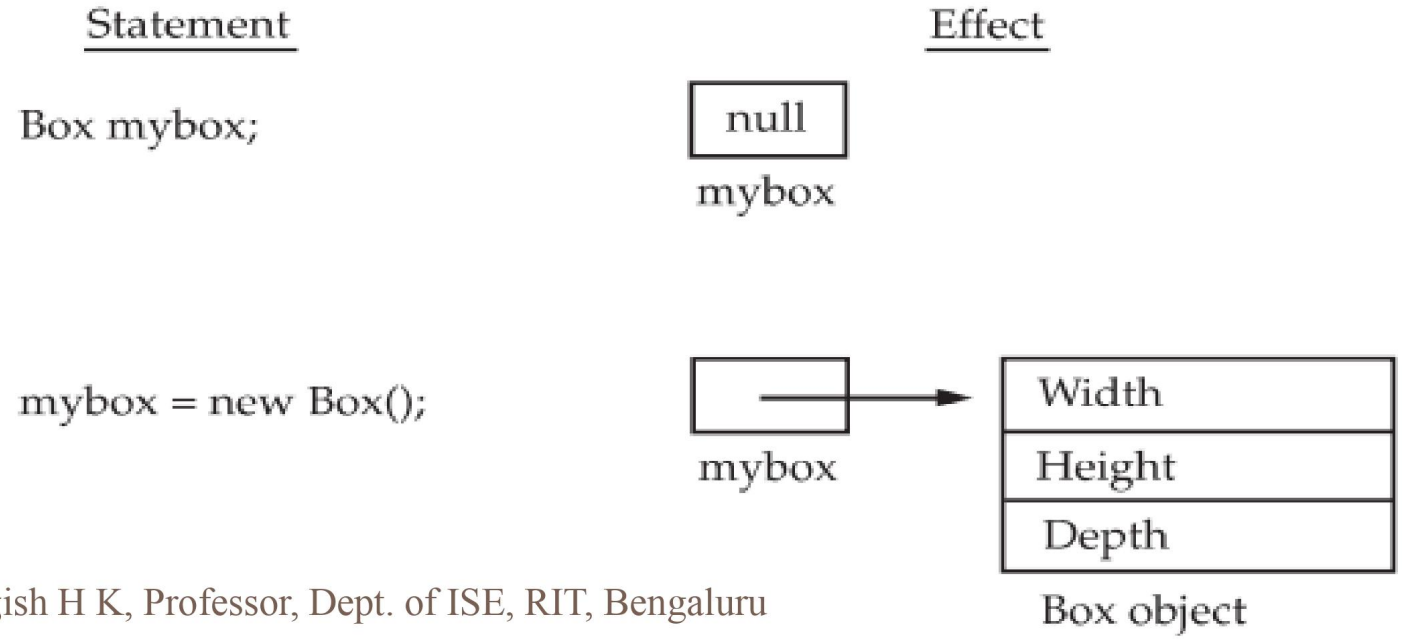
- Creates the Box object:

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

- The first line declares mybox as a reference to an object of type Box. After this line executes, mybox contains the value null, which indicates that it does not yet point to an actual object.
- Below statement combines the two steps.
- `Box mybox = new Box();` // create a Box object called mybox

new

- It is an unary operator
- It dynamically allocates memory for an object.
- It has the general form: class-var = new classname( );
- Example : declaring an object of type **Box**



Assigning Object Reference Variables

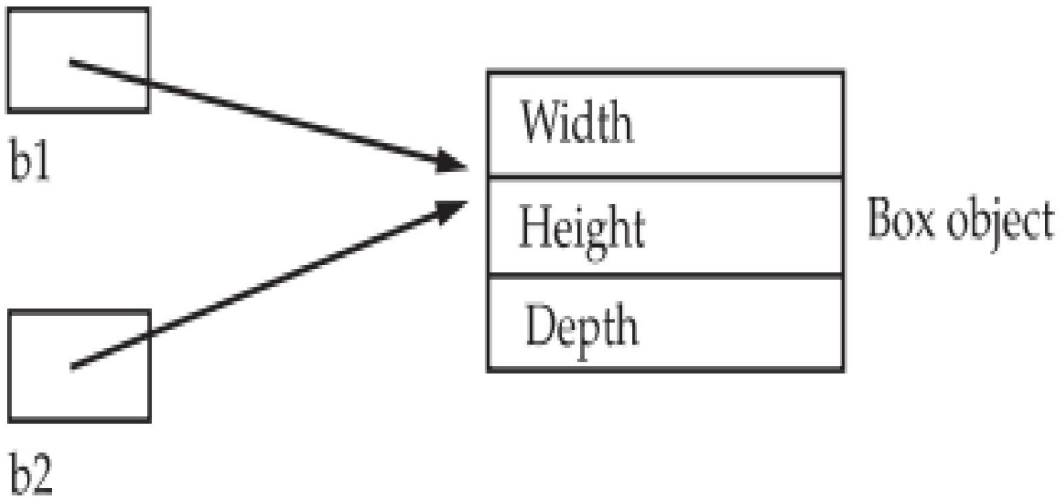
- Consider the following code fragment:

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

- b2 is being assigned a reference to a copy of the object referred to by b1. Where 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 the original object.
- It simply makes b2 refer to the same object as does b1.
- Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

Assigning Object Reference Variables …

- The above situation is depicted as shown below:



Methods

- **Methods** are declared inside the body of the class but immediately after the declaration of instance variables.
- The general form of a method declaration is:

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

Methods …

- Here, type specifies the type of data returned by the method. This can be any valid type, including class type.
- If the method **does not return a value**, its return type must be **void.**
- The name of the method is specified by methodname. This can be any legal identifier.

Methods …

- The parameter-list is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has 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 using the following form of the return statement:

  return value;

- Here, value is the value returned.

Example - Adding a Method to the Box Class


```
class Box
{
    double width, height, depth;
    void volume()
     {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}
```

Program to find Volume of Box – Without return value


```
class Box {
                    double width, height, depth;
                    void volume() {
                                        System.out.print("Volume is ");
                                        System.out.println(width * height * depth);
                    }
    }
class BoxDemo {
            public static void main(String args[]) {
            Box mybox1 = new Box();
            mybox1.width = 10;
            mybox1.height = 20;
            mybox1.depth = 15;
            mybox1.volume();
   }
}
```

Program to find Volume of Box – With return value


```
class Box {
        double width, height, depth;
        double  volume() {
                return(width * height * depth);
        }
}
class BoxDemo {
        public static void main(String args[]) {
        Box mybox1 = new Box();
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        double res=mybox1.volume();
        System.out.println("Volume of box is " + res);
    }
}
```

Program to find Volume of Box – With return value and method with arguments


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

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

Program to find Volume of Box – With return value and method with arguments


```
class BoxDemo {
        public static void main(String args[]) {
        Box mybox1 = new Box();
       double res=mybox1.volume(10,20,15);
         System.out.println("Volume of box is " + res);
    }
}
```

Constructors

- A constructor initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the object is created, before the new operator.
- Constructors - have no return type, not even void.

Box uses a
constructor to initialize the dimensions of a box


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

        Box() {
                System.out.println("Constructing Box");
                width = 10;
                height =10;
                depth = 10;
            }
        double volume() {
                        return width * height * depth;
            }
}
```

Box uses a constructor to initialize the dimensions of a box…


```
class BoxDemo {
        public static void main(String args[]) {
          // declare, allocate, and initialize Box objects
          Box mybox1 = new Box();
          System.out.println("Volume is " + mybox1.volume());
    }
}
```

Parameterized Constructors – Having parameters


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

    Box(double w, double h, double d) {
        System.out.println("Constructing Box");
        width = w;
        height = h;
        depth = d;
    }
     double volume() {
            return width * height * depth;
        }
}
```

Parameterized Constructors – Having parameters …


```
class BoxDemo {
        public static void main(String args[]) {
          // declare, allocate, and initialize Box objects
          Box mybox1 = new Box(10,20,5);
          System.out.println("Volume is " + mybox1.volume());
    }
}
```

The 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.
- Example

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

- it is similar to

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

- Inside Box( ), this will always refer to the invoking object.

Instance Variable Hiding

- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- But you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.
- However, when a local variable has the same name as an instance variable, the local variable hides the instance variable.
- This is why width, height, and depth were not used as the names of the parameters to the Box( ) constructor inside the Box class.
- If they had been, then width would have referred to the formal parameter, hiding the instance variable width.
- While it is usually easier to simply use different names or can use **this** to resolve any name space collisions that might occur between instance variables and local variables.

Instance Variable Hiding …

- Example, here is another version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name:
- Use ***this*** to resolve name-space collisions.

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

Garbage Collection

- Since objects are dynamically allocated by using the new operator. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called **garbage collection**.
- It works like this: When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

The finalize( ) Method

- It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called finalize( ), and it can be used to ensure that an object terminates cleanly.
- For example, you might use finalize( ) to make sure that an open file owned by that object is closed. To add a finalizer to a class, you simply define the finalize( ) method.
- The Java runtime calls that method whenever it is about to reclaimed an object of that class. Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.

The finalize( ) Method …

- The finalize( ) method has this general form:

```
protected void finalize( )
{
        // finalization code here
}
```

- the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class.
- finalize( ) method is a method of a class which is called just before the destruction of an object by the garbage collector.
- finalize() method called only once by garbage collector.
- After finalize( ) method gets executed completely, the object automatically gets destroyed.
- The purpose of calling finalize method is to perform activities related to clean up, resource deallocation etc.

Example


```
class Gclass{
                protected void finalize()    {
                        System.out.println("Garbage Collected ");
    }
}
class Main{
        public static void main(String[] args) {
        Gclass obj = new Gclass();
        obj=null;
        System.gc();
     }
 }
```

Student Class


```
class Student{
        int id;
        String name;
 }
class Teststudent{
        public static void main(String args[]){
                Student s=new Student();
                System.out.println(s.id  +  s.name);
  }
 }
```

initialize object – reference


```
class Student{
        int id;
        String name;
   }
class Teststudent{
        public static void main(String args[]){
        Student s=new Student();
        s.id=120;
        s.name="Viraat";
        System.out.println(s.id  +  s.name);
 }
}
```

initialize object – method


```
class Student{
        int id;
        String name;
         void getd(int num, String s ) {
                id=num;
                name=s;
          }
       void disp() {
                System.out.println(id + name);
         }
}
class Teststudent{
        public static void main(String args[]){
          Student s=new Student();
          s.getd(120,"Viraat");
          s.disp();
    }
}
```

initialize object – constructor


```
class Student{
                int id;
                String name;
                Student(int num, String s ) {
                                id=num;
                                name=s;
                        }
        void disp() {
           System.out.println(id + name);
      }
}
class Teststudent{
                public static void main(String args[]){
                Student s=new Student(120,"Viraat");
                s.disp();
       }
}
```

Overloading Methods

- Two or more methods within the same class that share the same name, but differ by parameter.
- overloaded methods must differ in the type and/or number of their parameters.
- Method overloading is one of the ways that Java supports polymorphism.
- 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.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Overloading Methods …Example


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

        void test(int a, int b) {    // Overload test for two integer parameters
                System.out.println("a and b: " + a + " " + b);
            }

        void test(double a) {   // overload test for a double parameter
                System.out.println("double a: " + a);
            }
}
```

Overloading Methods …Example


```
class Overload {
        public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        ob.test();
        ob.test(10, 20);
        ob.test(123.25);
        ob.test(10);
    }
}
```

Overloading Constructors

- Like a normal methods, we can also overload constructor methods.
- Example:

```
class Box {
        double width, height, depth;
    Box() {
        width = -1;      // use -1 to indicate
        height = -1;     // an uninitialized
        depth = -1;      // box
    }
```

Continue …


```
    // constructor used when cube is created
    Box(double len) {
            width = height = depth = len;
    }
    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
            width = w;
            height = h;
            depth = d;
    }
    double volume() {
            return width * height * depth;
    }
}
```

Continue …


```
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 mybox3 = new Box(7);
        double vol;
        vol =mybox1.volume();          // get volume of first box
        System.out.println("Volume of mybox1 is " + vol);

        vol = mybox2.volume();         // get volume of second box
        System.out.println("Volume of mybox2 is " + vol);

        vol = mybox3.volume();        // get volume of cube
        System.out.println("Volume of mycube is " + vol);
    }
}
```

Using Objects as Parameters

- So far, we have only been using simple types as parameters to methods. However, it is to pass objects to methods.

```
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
            }
        // return true if obj is equal to the invoking object
        boolean equals(Test obj) {
            if(a == obj.a && b == obj.b)
                return true;
            else
                return false;
        }
}
```

Using Objects as Parameters …


```
class PassOb {
        public static void main(String args[]) {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);
        System.out.println("ob1 == ob2: " + ob1.equals(ob2));
        System.out.println("ob1 == ob3: " + ob1.equals(ob3));
    }
}
```

Argument Passing

- Two ways to pass an argument to a function or method.
- The first way is : **call-by-value**. This approach copies the value of an argument into the formal parameter of the method. Therefore, changes made to the parameter of the method have no effect on the argument.
- The second way is : **call-by-reference.** In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the method, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the method.

Call by value …example

- In Java, when you pass a primitive type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

```
class Test {
        void fun(int i, int j) {
            i *= 2;
            j /= 2;
        }
}
class CallByValue {
        public static void main(String args[]) {
                Test ob = new Test();
                int a = 15, b = 20;
                System.out.println("a and b before call: " + a + " " + b);
                ob.fun(a, b);
                System.out.println("a and b after call: " +  a + " " + b);
            }
}
```

Call by value …example…

The output from this program is shown here:

a and b before call: 15 20

a and b after call: 15 20

Call by reference …example…

- Pass an object to a method
- When you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference.
- Changes to the object inside the method do affect the object used as an argument.

Call by reference …example…


```
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }
        // pass an object
        void fun(Test obj) {
                obj.a *= 2;
                obj.b /= 2;
        }
}
```

Call by reference …example…


```
class CallByRef {
    public static void main(String args[]) {
    Test ob = new Test(15, 20);
    System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
    ob.fun(ob);
     System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
   }
}
```

This program generates the following output:

ob.a and ob.b before call: 15 20

ob.a and ob.b after call: 30 10

Introducing Access Control

- As you know, encapsulation **links data with the code** that manipulates it. However, encapsulation provides another important attribute: **access control**.
- Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse.
- Java's access specifiers are **public, private,** and **protected**. Java also defines a *default access level*. protected applies only when inheritance is involved.
- When a member of a class is modified by the public specifier, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed by other members of its class.
- Now you can understand why main( ) has always been preceded by the public specifier. It is called by code that is outside the program—that is, by the Java run-time system. 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.

Introducing Access Control …


```
class Test {
        int a;              // default access
        public int b;       // public access
        private int c;      // private access
                            // methods to access c
        void setc(int i) {          // set c's value
                            c = i;
        }
        int getc() {                // get c's value
                            return c;
                }
}
```

Introducing Access Control …


```
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
ob.a = 10;                  // These are OK, a and b may be accessed directly
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());
}
}
```

Static Variables and Methods

- **Static Variables** : The static keyword is used to create variables that will exist independently of any instances created for the class.
- Only one copy of the static variable exists regardless of the number of instances of the class. *Static variables are also known as class variables*.

Static Variables and Methods

- **Static Methods**
- The **static** keyword is used to create methods that will exist independently of any instances created for the class.
- **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

Example: shows a class that has a static method, some static variables, and a static initialization block:


```
class UseStatic {
        static int a = 3;
        static int b;
        static void fun(int x) {
                                        System.out.println("x = " + x);
                                        System.out.println("a = " + a);
                                        System.out.println("b = " + b);
                }
        static {
                System.out.println("Static block initialized.");
                b = a * 4;
         }
        public static void main(String args[]) {
        fun(42);
      }
}
```

Continue…

As soon as the UseStatic class is loaded, all of the static statements are run.

First, a is set to 3,

then the static block executes, which prints a message and then initializes b to a*4 or 12.

Then main( ) is called, which calls fun( ), passing 42 to x. The three println( ) statements refer to the two 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

Continue…

- Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator.
- For example, if you wish to call a static method from outside its class, you can do so using the following general form:

  classname.method( ) ;

- Here, classname is the name of the class in which the static method is declared.

Continue… Example


```
class StaticDemo {
        static int a = 42;
        static int b = 99;
        static void callme() {
                        System.out.println("a = " + a);
                }
}
class StaticByName {
                        public static void main(String args[]) {
                        StaticDemo.callme();
                        System.out.println("b = " + StaticDemo.b);
                }
}
```

Introducing final

- A variable can be declared as final.
- Doing so prevents its contents from being modified.
- You must initialize a final variable when it is declared. For example:
- final int SIZE= 10;
- final double PI= 3.14;
- final int QUIT = 5;
- Subsequent parts of your program can now use SIZE, PI etc.,

Students details


```
class Student{
        int rollno;
        String name;
        Student(int r, String n){
                        rollno=r;    name=n;
                }
 void display(){
    System.out.println(rollno+" "+name);
   }
}
```

```
class Main{
public static void main(String[] args) {
 Student s[]=new Student [2];
  s[0]=new Student(120,"Karan");
  s[1]=new Student(222,"Arjun");
   s[0].display();
   s[1].display();
  }
  }
```

Java User Input – from key board

- The Scanner class is used to get user input, and it is found in the java.util package.

```
import java.util.Scanner;                          // Import the Scanner class
class Main {
  public static void main(String[] args) {
    Scanner  in = new Scanner(System.in);          // Create a Scanner object
    System.out.println("Enter Number");
    int num= in.nextInt();                         // Read user input
    System.out.println("User  num is: " + num);  // Output user input
  }
}
```

Java User Input – from key board

- Input Types

Method	Description
nextBoolean()	Reads a boolean
nextByte()	Reads a byte
nextShort()	Reads a short
nextInt()	Reads a int
nextFloat()	Reads a float
nextDouble()	Reads a double
nextLong()	Reads a long
nextLine()	Reads a String
Next().charAt(0)	Reads a Character

THANK YOU