2. INTRODUCTION TO CLASSES

❖ CLASS FUNDAMENTALS:-

- A class can be defined as a template/ blue print that describe the behaviors/states that object of its type support.
- A class is a user-defined data type with a template that serves to define its properties.
- once the class type has been defined, we can create "variable s" of that type using declarations that are similar to the basic type declarations.
- In java, these variables are termed as instances of classes, which are the actual objects.
- The basic form of a class definition is:

Class Syntax:-

```
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
        }
}
```

SIMPLE CLASS

- Here is a class called Box that defines three instance variables: width, height, and depth.
- Currently, Box does not contain any methods.

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

- As stated, a class defines a new type of data.
- It is important to remember that a class declaration only creates a template; it does not create an actual object.

- To actually create a Box object, you will use a statement like the following:
  - **Box b = new Box();** // create a **Box** object called **b**
- After this statement executes, **b** will be an instance of Box.
- Again, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class.
- Thus, every Box object will contain its own copies of the instance variables width, height, and depth.
- To access these variables, you will use the dot (.) operator. The dot operator links the name of the object with the name of an instance variable.
- For example, to assign the width variable of mybox the value 100, you would use the following statement:
  - **b.width = 100;**

```
/* A program that uses the Box class. Call this file 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[])
        {
                Box b = new Box();
                double vol;
                b.width = 10;
                b.height = 20;
                b.depth = 15;
                vol = b.width * b.height * b.depth;
                System.out.println("Volume is " + vol);
        }
}
```

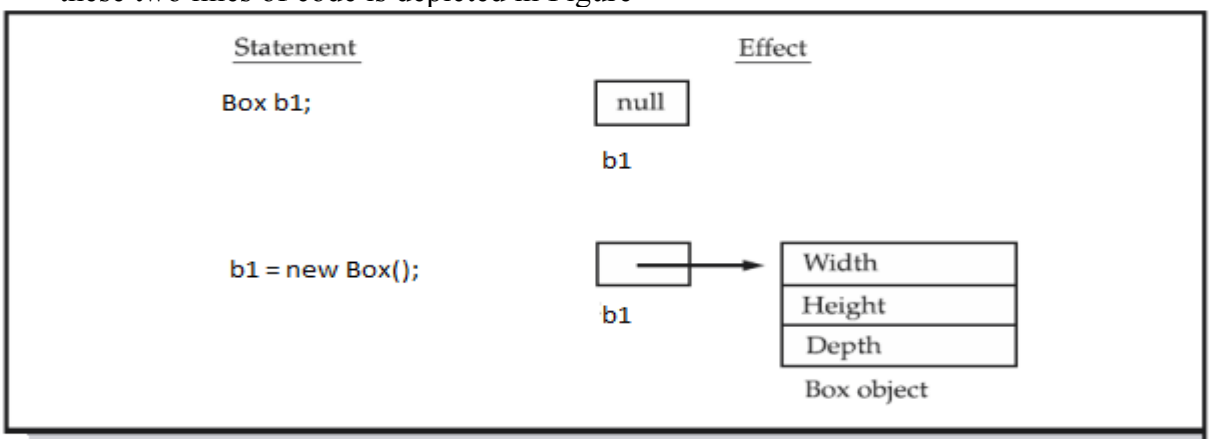
- You should call the file that contains this program BoxDemo.java, because the main( ) method is in the class called BoxDemo, not the class called Box.
- When you compile this program, you will find that two .class files have been created, one for Box and one for BoxDemo.
- When you do, you will see the following output:
  - Volume is 3000.0

❖ DECLARING OBJECTS:

- **Object: object is a basic run time entity of the class.**
- As pointed out earlier, an object in java is essentially a block of memory that contains space to store all the instance variables.
- Object in java are created using the **new** operator.
- The **new** operator creates an object of the specified class and returns a reference to that object.
- Here is an example of creating an object of type Box**.**

```
Box b1; // declare reference to object
B1 = new Box(); // allocate a Box object
```

- The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to the variable.
- But in reality, b1 simply holds the memory address of the actual Box object. The effect of these two lines of code is depicted in Figure

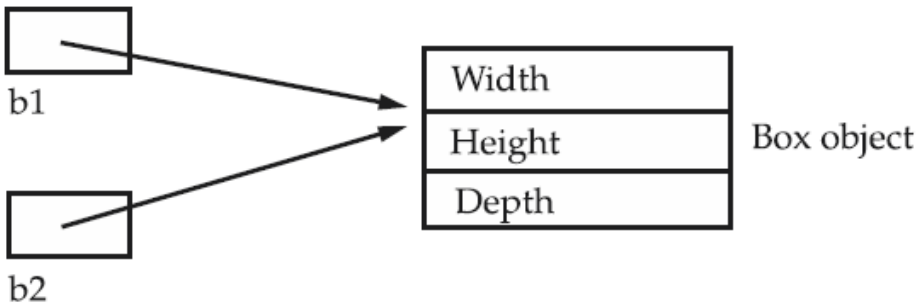


❖ ASSIGNING OBJECT REFERENCE VARIABLES

- 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;
- 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 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.
- This situation is depicted here:



❖ INTRODUCING METHODS:

- A class with only data fields has no life.
- The data contained in the class.
- Methods(functions) are declared inside the body of the class but immediately after the declaration of instance variables.
- The general form of a methods declaration is

  return type methodname (parameter-list)

  {

  Method-body;

  }
- Method declarations have four basic parts:
  - The name of the method(method name)

- The type of the value the method returns(type)
- A list of parameters(parameter-list)
- The body of the method

- The type specifies the **type of value** the method would return**.**
- It could any data type or void, if the method does not return any value then its data type will be **void**.
- The ***method name*** is a valid identifier.
- The **parameter list** is always enclosed in parentheses.

- **Example:**

```
// This program includes a method inside the box class.
class Box
{
        double width;
        double height;
        double depth;
        void volume()
        {
                System.out.print("Volume is ");
                System.out.println(width * height * depth);
        }
}
class BoxClass
{
        public static void main(String args[])
        {
                Box b = new Box();
                b.width = 10;
                b.height = 20;
                b.depth = 15;
                b.volume();
        }
}
```

- **Example:**

```
// Now, volume() returns the volume of a box.
class Box
{
    double width;
    double height;
    double depth;
    // compute and return volume
    double volume()
    {
            return width * height * depth;
    }
}
class BoxClass
{
    public static void main(String args[])
    {
        Box b = new Box();
```

```
        double vol;
        b.width = 10;
        b.height = 20;
        b.depth = 15;
        vol = b.volume();
        System.out.println("Volume is " + vol);
    }
}
```

- **Example:**

// **method-1**:-This program uses a parameterized method.

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

    void setDim(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
}
class DemoBox
{
    public static void main(String args[])
    {
        Box b1 = new Box();
        Box b2 = new Box();
        double vol;
        b1.setDim(10, 20, 15);
        b2.setDim(3, 6, 9);
        vol = b1.volume();
        System.out.println("Volume is " + vol);
        vol = b2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

OUTPUT of Method-1 and Method-2:-

Volume is 3000.0
Volume is 162.0

❖ CONSTRUCTORS:

- This automatic initialization is performed through the use of a constructor.
- A constructor will call initializes an object immediately upon creation.
- It has the same name as the class and syntactically similar to a method.

- Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
- Constructors have no return type, not even void.
- This is because the implicit return type of a class' constructor is the class type itself.
- Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:
- **Example:**

```
/* Here, Box uses a constructor to initialize the dimensions of a box.*/
class Box
{
        double width;
        double height;
        double depth;
        // This is the constructor for Box with no parameter.
        Box()
        {
                width = 10;
                height = 10;
                depth = 10;
        }
        // compute and return volume
        double volume()
        {
                return width * height * depth;
        }
}
class DemoBox
{
        public static void main(String args[])
        {
                Box b1 = new Box();
                Box b2 = new Box();
                double vol;

                vol = b1.volume();
                System.out.println("Volume is " + vol);

                vol = b2.volume();
                System.out.println("Volume is " + vol);
        }
}
```

OUTPUT:-

```
Volume is 1000.0
Volume is 1000.0
```

❖ PARAMETERIZED CONSTRUCTORS

- What is needed is a way to construct Box objects of various dimensions.
- The easy solution is to add parameters to the constructor. As you can probably guess, this makes them much more useful.

- For example, the following version of Box defines a parameterized constructor which sets the dimensions of a box as specified by those parameters. Pay special attention to how Box objects are created.
- **Example:**

/* 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 DemoBox
{
    public static void main(String args[])
    {
            Box b1 = new Box(10, 20, 15);
            Box b2 = new Box(3, 6, 9);
            double vol;
            vol = b1.volume();
            System.out.println("Volume is " + vol);
            vol = b2.volume();
            System.out.println("Volume is " + vol);
    }
}
```

❖ 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.
- 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;
}
```

- This version of Box( ) operates exactly like the earlier version.

- The use of this is redundant,but perfectly correct.
- Inside Box( ), this will always refer to the invoking object.
- **Example:**

```
class A
{
    int a,b,c;
    void input(int a,int b)
    {
      this.a=a;
      this.b=b;
    }
    int mul()
    {
        return a*b;
    }
}
class This1
{
    public static void main(String[] args)
    {
        A a=new A();
        a.input(2,3);
        int x;
        x=a.mul();
        System.out.println("X Value:"+x);
    }
}
```

❖ GARBAGE COLLECTION:

- Garbage collection is one of the most important features of Java.

- The purpose of garbage collection is to identify and delete objects(or allocated memory of objects) that are not needed by a program so that their resources can be reused.
- Garbage collection is also called automatic memory management.
- Garbage collection is an automatic process.
- **Example:-2**

  - Run the garbage collector using System class

```
public class Main
{
    public static void main(String[] args)
    {
        System.gc();
    }
}
```

❖ THE finalize() METHOD:

- Finalize method will use when object will need to perform some action when it is destroyed.
- For example, if an object has some non-java resource such as a file handle or window character font, then you have free those resources for future use.
- To handle such situations, java provides a mechanism called finalization.
- Right before an asset is freed, the java run time calls the **finalize()** method on the object.
- 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.

- It is important to understand that **finalize()** is only called just prior to garbage collection.
- It is not called when an object goes out-of-scope, for example. This means program should provide other means of releasing system resources, etc., used by the object. It must not rely on finalize() for normal program operation.

❖ Wrapper Class

- Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object (with **java.io.File**), an address of a system can be seen as an object (with **java.util.URL**), an image can be treated as an object (with **java.awt.Image**) and a simple data type can be converted into an object (with **wrapper classes**). This tutorial discusses wrapper classes. **Wrapper classes** are used to convert any data type into an object
- As the name says, a wrapper class wraps (encloses) around a data type and gives it an object appearance. Wherever, the data type is required as an object, this object can be used. Wrapper classes include methods to unwrap the object and give back the data type. It can be compared with a chocolate. The manufacturer wraps the chocolate with some foil or paper to prevent from pollution. The user takes the chocolate, removes and throws the wrapper and eats it.
- The name of the wrapper class corresponding to each primitive data type, along with the arguments its constructor accepts, is listed below:

Primitive datatype	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

- **Example**

```
public class WrappingUnwrapping
{
        public static void main(String args[])
        {
                int marks = 50;
                Integer m1 = new Integer(marks);
                System.out.println("Integer object m1: " + m1);
                int iv = m1.intValue();
                System.out.println("int value, iv: " + iv);
        }
}
```

OUTPUT:

Integer object m1: 50

int value, iv: 50