Sri Sai Vidya Vikas Shikshana Samithi ®
SAI VIDYA INSTITUTE OF TECHNOLOGY
Approved by AICTE, New Delhi, Affiliated to VTU, Recognized by Govt. of Karnataka
Accredited by NBA
RAJANUKUNTE, BENGALURU 560 064, KARNATAKA
Phone: 080-28468191/96/97/98 ,Email: info@saividya.ac.in, URLwww.saividya.ac.in

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (CSE)**

# Module -2

## Introducing 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.
- The class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

## Class Fundamentals

- The classes created exist simply to encapsulate the **main( )** method, which has been used to demonstrate the basics of the Java syntax.
- A class is that it defines a new 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.

## The General Form of a Class

- A class is declared by use of the **class** keyword.
- The classes that have been used up to this point 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 methodname1(parameter-list) {
    // body of method
  }
  type methodname2(parameter-list) {
    // body of method
  }
  // ...
  type methodnameN(parameter-list) {
     // body of method
  }
}
```

- 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.
- The general rule, it is the methods that determine how a class' data can be used.
- 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.
- The general form of a class does not specify a **main( )** method.
- Java classes do not need to have a **main( )** method.
- You only specify one if that class is the starting point for your program.

## A 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;
}
```

A class defines a new type of data. In this case, the new data type is called **Box**. You will 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. Thus, the preceding code does not cause any objects of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, **mybox** will refer to an instance of **Box**. Thus, it will have "physical" reality

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:

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

In general, we use the dot operator to access both the instance variables and the methods within an object.

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

```
/* 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 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);
          }
        }
```

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**. The Java compiler 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. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively.

**output:**

```
Volume is 3000.0
```

Each object has its own copies of the instance variables. If you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables 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);
      }
    }
```

The output produced by this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

**mybox1**'s data is completely separate from the data contained in **mybox2**.

## Declaring Object:

- When you create a class, we are creating a new data type. Obtaining 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 (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, essentially, the address in memory of the object allocated by **new**.
- This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated

Declare an object of type **Box**:

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

This statement combines the two steps just described.

```
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**. At this point, **mybox** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds, in essence, the memory address of the actual **Box** object.

## A Closer Look at new

The **new** operator dynamically allocates memory for an object. In the context of an assignment, it has this general form:

*class-var* = new *classname* ( );

- Here, *class-var* is a variable of the class type being created.
- The *classname* is the name of the class that is being instantiated.
- The class name followed by parentheses specifies the *constructor* for the class.
- A constructor defines what occurs when an object of a class is created.
- Constructors are an important part of all classes and have many significant attributes.
- Most real-world classes explicitly define their own constructors within their class definition.
- However, if no explicit constructor is specified, then Java will automatically supply a default constructor.

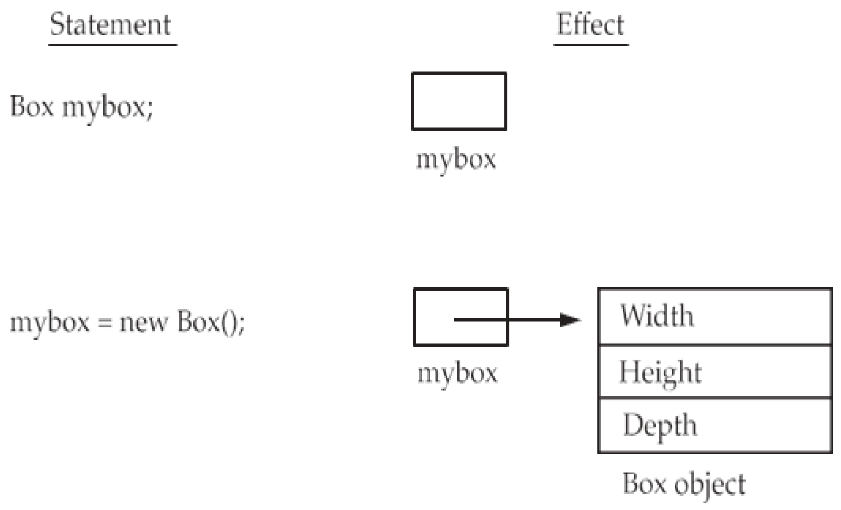


Figure: Declaring an object of type **Box**

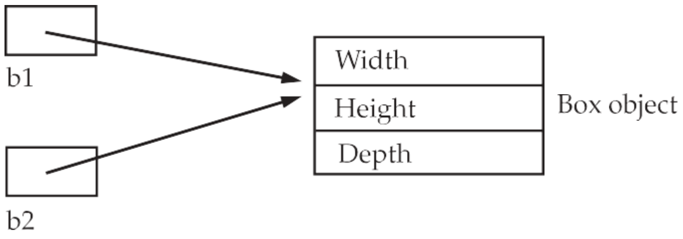
## Assigning Object Reference Variables

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;
```

**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:



Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**.

For 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.

# Introducing Methods

Classes usually consist of two things: instance variables and methods, to add methods to your classes.

**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 type must be **void**.
- The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope.
- 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.

## Adding a Method to the Box Class

Adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// This program includes a method inside the box class.

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

  // display volume of a box
  void volume() {
    System.out.print("Volume is ");
    System.out.println(width * height * depth);
  }
}

class BoxDemo3 {
  public static void main(String[] args) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    // 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;

         // display volume of first box
         mybox1.volume();

         // display volume of second box
         mybox2.volume();
       }
     }
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

```
Here,  mybox1.volume();

       mybox2.volume();
```

The first line here invokes the **volume( )** method on **mybox1**. That is, it calls **volume( )** relative to the **mybox1** object, using the object's name followed by the dot operator.

Thus, the call to **mybox1.volume( )** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume( )** displays the volume of the box defined by **mybox2**. Each time **volume( )** is invoked, it displays the volume for the specified box.

## Returning a Value

A better way to implement **volume( )** is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
// 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 BoxDemo4 {
          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;

            // 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);
          }
        }
```

There are two important things to understand about returning values:

The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.

The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

The call to **volume( )** could have been used in the **println( )** statement directly, as shown here:

```
System.out.println("Volume is" + mybox1.volume());
```

In this case, when **println( )** is executed, **mybox1.volume( )** will be called automatically and its value will be passed to **println( )**.

## Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()
{
  return 10 * 10;
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make **square( )** much more useful.

```
int square(int i)
{
  return i * i;
}
```

Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

```
// This program uses a parameterized method.

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

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

          // sets dimensions of box
          void setDim(double w, double h, double d) {
            width = w;
            height = h;
            depth = d;
               }
             }

             class BoxDemo5 {
               public static void main(String[] args) {
                 Box mybox1 = new Box();
                 Box mybox2 = new Box();
                 double vol;

                 // initialize each box
                 mybox1.setDim(10, 20, 15);
                 mybox2.setDim(3, 6, 9);

                 // 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);
               }
             }
```

As you can see, the **setDim( )** method is used to set the dimensions of each box.

## Constructors

- Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.
- 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 when 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.

**Types of Constructors in Java**

- Default Constructor
- Parameterized Constructor

### 1. Default Constructor in Java

A constructor that has no parameters is known as default the constructor. A default constructor is invisible.

### 2. Parameterized Constructor in Java

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then we use a parameterized constructor.

You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim( )** with a constructor.

This version is shown here:

```
/* 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.
  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();
    Box mybox2 = new Box();

    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);
  }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

## Parameterized Constructors

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then we use a parameterized constructor.

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

```
/* 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;
```

```
        // 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);
  }
}
```

The output from this program is shown here:

```
   Volume is 3000.0
   Volume is 162.0
```

## The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines 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;
}
```

## The this Keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

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;
}
```

## Instance Variable Hiding

- Instance variable hiding refers to a state when instance variables of the same name are present in superclass and subclass.
- 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.
- Because **this** lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables.
- For 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

- Java takes an approach for deallocation of memory 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 need to explicitly destroy objects.
- 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.

## A Stack Class

- A *stack* stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used.
- Stacks are controlled through two operations traditionally called *push* and *pop*.
- To put an item on top of the stack, you will use push.
- To take an item off the stack, you will use pop. As you will see, it is easy to encapsulate the entire stack mechanism.
- Here is a class called **Stack** that implements a stack for up to ten integers:

```
// This class defines an integer stack that can hold 10 values

class Stack {
  int[] stck = new int[10];
  int tos;

  // Initialize top-of-stack
  Stack() {
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
       System.out.println("Stack underflow.");
```

```
            return 0;
          }
          else
            return stck[tos--];
        }
      }
        public static void main(String[] args) {
          Stack mystack1 = new Stack();
          Stack mystack2 = new Stack();

          // push some numbers onto the stack
          for(int i=0; i<10; i++) mystack1.push(i);
          for(int i=10; i<20; i++) mystack2.push(i);

          // pop those numbers off the stack
          System.out.println("Stack in mystack1:");
          for(int i=0; i<10; i++)
             System.out.println(mystack1.pop());

          System.out.println("Stack in mystack2:");
          for(int i=0; i<10; i++)
             System.out.println(mystack2.pop());
        }
      }
```

This program generates the following output:

```
 Stack in mystack1:
          9
          8
          7
          6
          5
          4
          3
          2
          1
          0

Stack in mystack2:
        19
        18
        17
        16
        15
        14
        13
        12
        11
        10
```