- **Class fundamentals:** 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. As such, the class forms the basis for object-oriented programming in Java.

- **Define term class:** Class is a user defined data type.

- **The general form of a class:**

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or data, most real-world classes contain both.

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. The 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** or **data members.**

The code is contained within **methods**. Collectively, the methods and variables defined within a class are called **members** of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that determine how a class' data can be used.

All methods have the same general form as **main(),** which we have been using this far. However, most methods will not be specified as **static** or **public.**

Notice that the general form of class does not specify a **main()** method. Java classes do not need to have a **main().** You only specify one if that class is the starting point for your program. Further, the applets don't require a **main()** method at all.

❖ **Simple class:**

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width, height and depth.**

For example,

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

here class defines a new type of data is called **box.** You will use this name to declare objects of type **box.** It is important 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 mybox=new box();        //create a box object called mybox
```

After this statement executes, **mybox** will be an **object/instance** of box.

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. Therefore, 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 mybox.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 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);
  }}
```

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);
}}
```

- **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. There are **two** step processes to obtain objects of a class.

1. You must declare a variable of the class type. This variable does not define an object. It is simply a variable that can refer to an object.

2. 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 by allocated by **new**.

This reference is then stored in the variable. Thus, in all class objects must be dynamically allocated.

In the preceding sample programs, a line similar to the following is used to declare an object of type **box:**

```
box mybox=new box();
```

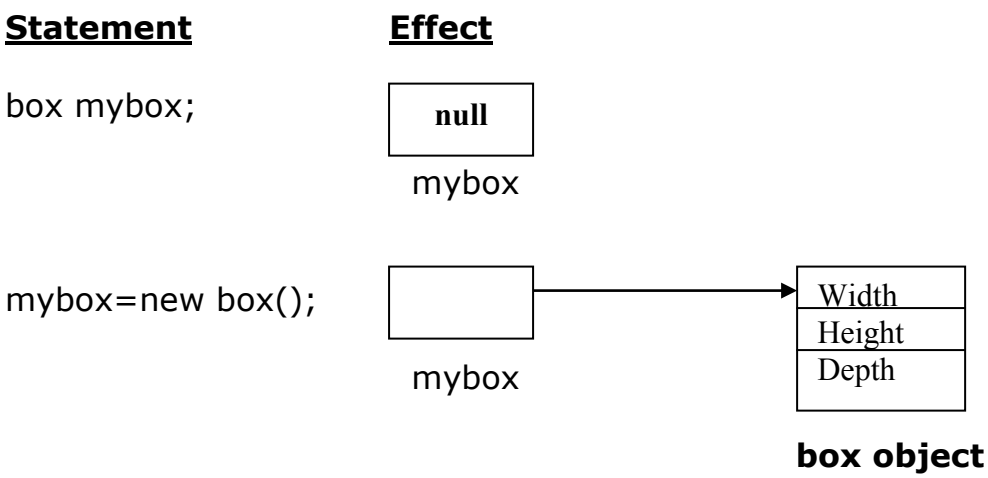
This statement can be rewritten like this to show each step more clearly;

```
box mybox;
mybox=new box();
```

- The first line declares mybox as reference to an object of type box. After this line executes, mybox contains the value null, which

indicates that it does not still point to object. Any attempt to use mybox at this point will result in a compile-time error.

- The next line allocates an actual 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 the memory address of the **actual box object**. The effect of these two lines of code is depicted in the following figure.



- **A closer Look at new:**

Or

- **Write a short note about new**

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

Syntax:

class-var=new classname();

Where
class-var = it is a variable of class type or an object of class type.
classname = it is the name of the class under which the object of that class
is being created.

The class name followed by parentheses specifies the default constructor of
the class.

For example

box mybox=new box(); [box() specifies the default constructor].

New allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However computer has fixed memory, so **new** may not be allocate enough memory to your program. If this happens, a run-time exception will occur. You can handle this situation by creating exception handling mechanism in your program.

- **Why do not need to use new for simple data types (int, char etc)?**

Java's simple types are not implemented as objects. Simple types are implemented as "normal" variables. This is required for the efficiency of variables. Because objects have many features and attribute that require java to take care of them differently than it treats the simple data types.

- **What is the difference between class and object?**

Class	Object
(1) Class is a logical framework	(1) object is a physical framework
(2) Class has not space of memory	(2) object has space of memory

- **How to assign object as reference variable?**

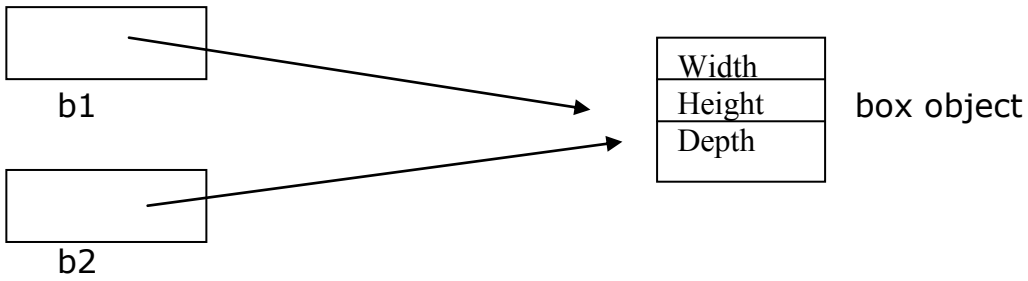
You can assign an object as reference variable to same class variable.

For example

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

In above statement both b1 and b2 will refer same object after execution. The assignment of b1 and 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. Therefore any changes occur by object b2 will affect the object b1, because they are the same object.

This situation is depicted here:



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;
```

❖ Introduction of methods or declaration of methods.

Classes usually consist of two things: Instance variables and methods.
The objects created by such a class cannot respond to any messages. We must therefore add methods that are necessary for manipulating the data contained in the class. Methods are declared inside the body of the class but immediately after the declaration of data member or instance variable.

This is the general form of a method:

```
type methodname(parameter list)
{
```

```
        //body of method
}
```

Method declarations have four basic parts:

1. The name of the method (methodname)
2. The type of the value the method returns (type)
3. A list of parameters (parameter-list)
4. The body of the method

Here, type specifies the type of value the returned by method. This can be any valid simple data type such as int, float as well as any class type. If the method does not return a value, its return type must be void.
The method name is a valid identifier. The parameter-list is always enclosed in parentheses. This list contains variable names and types of all the values we want to give to the method as input. The variables in the list are separated by commas. In the case where no input data are required, the declaration must retain the empty parentheses.

For example,

```
        Setval(int x, float y, double z)      //three parameters
        getdata()                             //empty list
```

let us consider the box class again and add a method setdim() to it.

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

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

- **How to add method to the box class?**

Most of the case you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes.

The following program shows you how to add method to the box class.

```
// 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

❖ How to return value by method to the box class?

The following example is an improved version of the preceding program.
// 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.

Note: The preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. The call to **volume()** could have been used in the **println()** statement directly, as shown here:

system.out.println("volume is " +mybox1.volume());

- **How to add method that takes parameters?**

A parameterized method can operate on a variety of data and be used in a number of slightly different situations. Here is a method that returns the square of the number 10:

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

Here , use of this method is very limited. However, you can 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;
}
```

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=square(9);y=2;x=square(y);
```

A **parameter** is a variable defined by a method that receives a value when the method is called. For example, in **square()**, i is a parameter. An **argument** is a value that is passed to a method when it is called.

For example **square(100)** passes 100 as an argument. Inside **square()**, the parameter i receives that value.

//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);
  }
}
```

❖ CONSTRUCTORS:

A **constructor** is a special method which has same name as class name that is used to initialize the instance variable of class when we create an object of class.

Constructor is automatically called when we create an object of class. Constructor looks like a strange because they have no return type, not even **void**.

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

❖ Parameterized constructors:

While the **box()** constructor in the preceding example does initialize a **Box** object, it is not very useful because of all boxes have the same dimensions.

However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are known as **parameterized constructors**.

For example, the following version of **Box** defines a parameterized constructor which sets the dimensions of a box as specified by those parameters.

```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
```

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

  // This is the constructor for Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

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

class BoxDemo7 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

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

❖ This keyword:

Sometimes a method will need to refer to the object that called it. To allow this, java defines the **this** keyword. **this** can be used inside any method to refer to the current object. It means **this** is always a reference to the object on which the method was called. You can use **this** anywhere a reference to an object of the current class type is permitted.

Consider the following example to understand 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:**

Sometimes the name of instance variable (class data member) and the variable name of parameterized method may be same. When a local variable of parameterized method has the same name as an instance variable, the local variable hides the **instance variable.**

We can use **this** keyword to resolve the problem between instance variable and local variable by using **this** keyword. Because **this** directly refer the current object of class.

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:

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

  // This is the constructor for Box.
  Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
  }
  // 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);
  }
```

❖ Garbage collection:

Objects are dynamically allocated by using the **new** operator and destroyed the objects to release the memory for later reallocation in C++ language.
In C++ dynamically allocated objects must be manually released by use of **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 forcefully need to destroy objects as in C++.

When one or more objects exist and it is not used for longer time, garbage collection occurs infrequently during the execution of your program.

❖ The finalize() method:

Sometimes an object will need to perform some action when it is destroyed. **For example**, if an object is type of some non-Java resource such as close connection object of database file, record set file, file handle (open, read and write) or window character font, then you need to free these resources before an object is destroyed. To handle such situations, Java provides a mechanism called finalization.

Finalization method will be called when object need to be destroyed by the garbage collection.

To add a finalize method to a class, you simply define the **finalize()** method. Inside the **finalize()** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects.

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

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

// This Java program will fail after 5 calls to f().

```
class X {

    static int count = 0;

  // constructor
  X() {
   if(count<MAX) {
     count++;
    }
    else  {
      System.out.println("Error -- can't construct");
      System.exit(1);
    }
  }

  // finalization
  protected void finalize() {
    count--;
  }

  static void fun1()
  {
    X ob = new X(); // allocate an object
    // destruct on way out
  }

  public static void main(String args[]) {
    int i;

    for(i=0; i < 5; i++) {
      fun1();
      System.out.println("Current count is: " + count);
    }
  }
}
```