Chapter 3

Classes and Objects

A class, in the context of Java, are templates that are used to create objects, and to define object data types and methods. Core properties include the data types and methods that may be used by the object. All class objects should have the basic class properties. Classes are categories, and objects are items within each category.

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.

The General Form of a Class

A class is declared by use of the **class** keyword.

```
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. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class data can be used.

A simple class Example:

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

Class defines a new type of data. In this case, the new data type is called **Box**. Class only creates template but not an actual object.

Creation of object

A Java object is a combination of data and procedures working on the available data. An object has a state and behavior. The state of an object is stored in fields (variables), while methods (functions) display the object's behavior. Objects are created from templates known as classes. In Java, an object is created using the keyword **'new'** Object is an instance of a class.

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

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

Program demonstrating the declaration of class and creation of object

/* 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);
}
}
```

Each object has its own copies of the instance variables. This means that 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);
  }
}
```

Note: Obtaining objects of a class is a two-step process.

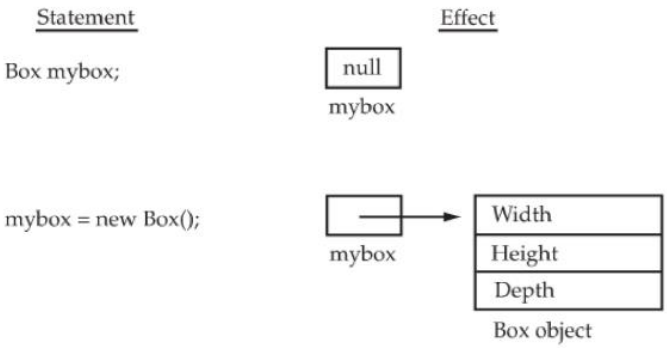
*First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.

*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.

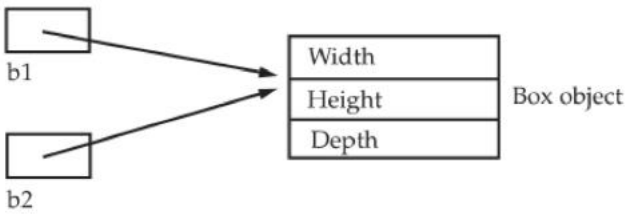
The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the objectallocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

Box mybox = new Box(); This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

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



Assigning Object Reference Variables
Box b1 = new Box();
Box b2 = b1;



Adding a Method to the Box Class:

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

Constructor

A constructor is a special member method which:

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

Note: Constructor doesn't have return type because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

A constructor can be of type :
i. default constructor (When no constructor are defined within a class Java calls the default constructor.)
ii. User defined constructor: When a constructor is explicitly defined by user. A user defined constructor can be parametrized or non parametrized(without any parameter).

The code below has non parametrized constructor.

/* 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);
}
}
```

Output:

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

Parameterized Constructors:

A constructor can have a parameters that can be used during object creation. Example:

```
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);
}
}
```

Output:

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

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. **this** can be used anywhere a reference to an object of the current class' type is permitted.

Example:

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

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

Using this to solve Instance Variable Hiding

When a local variable has the same name as an instance variable, the local variable hides the instance variable. Because this helps to refer directly to the object, it can be used to resolve any name space collisions that might occur between instance variables and local variables.
Example:

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

}
```

Garbage Collection

Garbage Collection in Java is a process by which the programs perform memory management automatically. The Garbage Collector(GC) finds the unused objects and deletes them to reclaim the memory. In Java, dynamic memory allocation of objects is achieved using the new operator that uses some memory and the memory remains allocated until there are references for the use of the object.

When there are no references to an object, it is assumed to be no longer needed, and the memory, occupied by the object can be reclaimed. There is no explicit need to destroy an object as Java handles the de-allocation automatically.

The finalize() Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization. By using finalization, you candefine specific actions that will occur when an object is just about to be reclaimed by the garbage collector. To add a finalizer to a class, you simply define thefinalize()method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside thefinalize()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 byany running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls thefinalize()method on the object.Thefinalize()method has this general form:

protected void finalize(){

// finalization code here

}

Stack Class

A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stacks are controlled through two operations traditionally called push and pop. To put an item on top of the stack, we will use push. To take an item off the stack, we will use pop.

```
// 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--];
  }
}
```