2. Introducing Classes

Class is a basis of OOP languages. It is a logical construct which defines shape and nature of an object. Entire Java is built upon classes.

2.1 Class Fundamentals

Class can be thought of as a user-defined data type. We can create variables (objects) of that data type. So, we can say that class is a ***template*** for an object and an object is an ***instance*** of a class. Most of the times, the terms *object* and *instance* are used interchangeably.

The General Form of a Class

A class contains data (member or instance variables) and the code (member methods) that operate on the data. The general form can be given as –

```
class classname
{
        type var1;
        type var2;
        …….
        type method1(para_list)
        {
                //body of method1
        }

        type method2(para_list)
        {
                //body of method2
        }
        ………..
}
```

Here, *classname* is any valid name given to the class. Variables declared within a class are called as ***instance variables*** because every instance (or object) of a class contains its own copy of these variables. The code is contained within ***methods***. Methods and instance variables collectively called as ***members*** of the class.

A Simple Class

Here we will consider a simple example for creation of class, creating objects and using members of the class. One can store the following program in a single file called ***BoxDemo.java***. (Or, two classes can be saved in two different files with the names ***Box.java*** and ***BoxDemo.java***.)

Program 2.1

```
class Box
{
        double w, h, d;
}

class BoxDemo
{
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();
                double vol;

                b1.w=2;
                b1.h=4;
                b1.d=3;

                b2.w=5;
                b2.h=6;
                b2.d=2;

                vol=b1.w*b1.h*b1.d;
                System.out.println("Volume of Box1 is " + vol);

                vol=b2.w*b2.h*b2.d;
                System.out.println("Volume of Box2 is " + vol);
        }
}
```

The output would be –

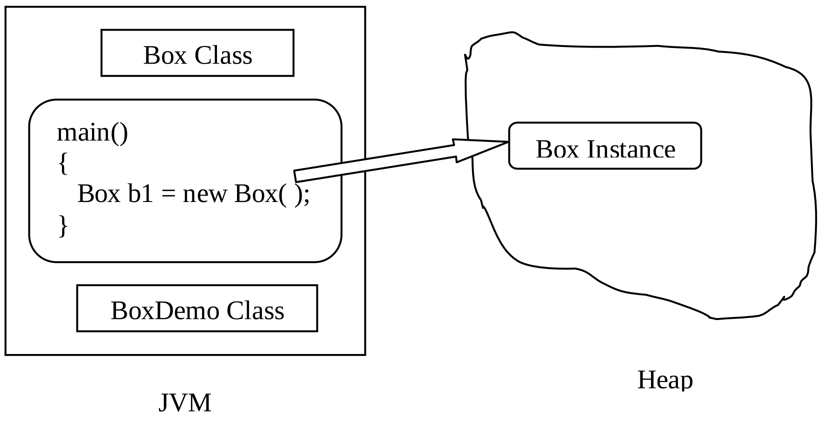
```
Volume of Box1 is 24.0
Volume of Box1 is 60.0
```

When you compile above program, two class files will be created viz. ***Box.class*** and ***BoxDemo.class***. Since *main()* method is contained in ***BoxDemo.class***, you need to execute the same.

In the above example, we have created a class ***Box*** which contains 3 instance variables *w, h, d***.**

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

The above statement creates a physical memory for one object of ***Box*** class. Every object is an instance of a class, and so, *b1* and *b2* will have their own copies of instance variables *w, h* and *d.* The memory layout for one object allocation can be shown as –



2.2 Declaring Objects

Creating a class means having a user-defined data type. To have a variable of this new data type, we should create an object. Consider the following declaration:

```
Box b1;
```

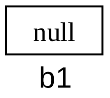
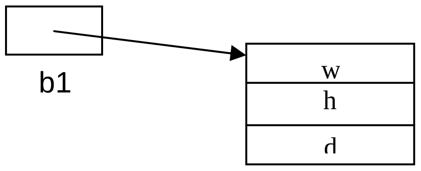
This statement will not actually create any physical object, but the object name *b1* can just ***refer*** to the actual object on the heap after memory allocation as follows –

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

We can even declare an object and allocate memory using a single statement –

Box b1=new Box();

Without the usage of *new*, the object contains **null.** Once memory is allocated dynamically, the object *b1* contains the address of real object created on the heap. The memory map is as shown in the following diagram –

Statement	**Effect**
Box b1;	
b1 = new Box();	

Closer look at *new*

The general form for object creation is –

obj_name = new class_name();

Here, ***class_name()*** is actually a constructor call. A ***constructor*** is a special type of member function invoked automatically when the object gets created. The constructor usually contains the code needed for object initialization. If we do not provide any constructor, then Java supplies a ***default constructor.***

Java treats primitive types like *byte, short, int, long, char, float, double* and *boolean* as ordinary variables but not as an object of any class. This is to avoid extra overhead on the heap memory and also to increase the efficiency of the program. Java also provides the *class-version* of these primitive types that can be used only if necessary. We will study those types later in detail.

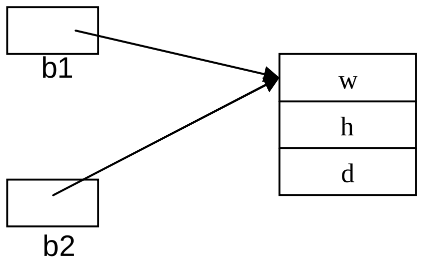
With the term *dynamic memory allocation*, we can understand that the keyword *new* allocates memory for the object during runtime. So, depending on the user's requirement memory will be utilized. This will avoid the problems with static memory allocation (either shortage or wastage of memory during runtime). If there is no enough memory in the heap when we use ***new*** for memory allocation, it will throw a run-time exception.

2.3 Assigning Object Reference Variables

When an object is assigned to another object, no separate memory will be allocated. Instead, the second object refers to the same location as that of first object. Consider the following declaration –

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

Now both b1 and b2 refer to same object on the heap. The memory representation for two objects can be shown as –



Thus, any change made for the instance variables of one object affects the other object also. 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.

NOTE that when you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

2.4 Introducing Methods

A class can consist of instance variables and methods. We have seen declaration and usage of instance variables in Program 2.1. Now, we will discuss about methods. The general form of a method is –

```
ret_type method_name(para_list)
{
        //body of the method
        return value;
}
```

Here, ***ret_type***	specifies the data type of the variable returned by the method. It may be any primitive type or any other derived type including name of the same class. If the method does not return any value, the ***ret_type*** should be specified as ***void***.
method_name	is any valid name given to the method
para_list	is the list of parameters (along with their respective types) taken the method. It may be even empty also.
body of method	is a code segment written to carryout some process for which the method is meant for.
return	is a keyword used to send ***value*** to the calling method. This line will be absent if the ***ret_type*** is void.

Adding Methods to *Box* class

Though it is possible to have classes with only instance variables as we did for ***Box*** class of Program 2.1, it is advisable to have methods to operate on those data. Because, methods acts as interface to the classes. This allows the class implementer to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself. Consider the following example –

Program 2.2

```
class Box
{
        double w, h, d;

        void volume()
        {
                System.out.println("The volume is " + w*h*d);
        }
}

class BoxDemo
{
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();

                b1.w=2;
                b1.h=4;
                b1.d=3;

                b2.w=5;
                b2.h=6;
                b2.d=2;

                b1.volume();
                b2.volume();
        }
}
```

The output would be –

```
The volume is 24.0
The volume is 60.0
```

In the above program, the ***Box*** objects *b1* and *b2* are invoking the member method *volume()* of the ***Box*** class to display the volume. To attach an object name and a method name, we use dot (**.**) operator. Once the program control enters the method *volume()*, we need not refer to object name to use the instance variables *w, h* and *d.*

Returning a value

In the previous example, we have seen a method which does not return anything. Now we will modify the above program so as to return the value of *volume* to *main()* method.

Program 2.3

```
class Box
{
        double w, h, d;

        double volume()
        {
                return w*h*d;
        }
}

class BoxDemo
{
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();
                double vol;
                b1.w=2;
                b1.h=4;
                b1.d=3;

                b2.w=5;
                b2.h=6;
                b2.d=2;

                vol = b1.volume();
                System.out.println("The volume is " + vol);
                System.out.println("The volume is " + b2.volume());
        }
}
```

The output would be –

```
        The volume is 24.0
        The volume is 60.0
```

As one can observe from above example, we need to use a variable at the left-hand side of the assignment operator to receive the value returned by a method. On the other hand, we can directly make a method call within print statement as shown in the last line of above program.

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.

Adding Methods that takes Parameters

Having parameters for methods is for providing some input information to process the task. Consider the following version of ***Box*** class which has a method with parameters.

Program 2.4

```
class Box
{
        double w, h, d;

        double volume()
        {
                return w*h*d;
        }

        void set(double wd, double ht, double dp)
        {
                w=wd;
                h=ht;
                d=dp;
        }
}

class BoxDemo
```

```
{
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();

                b1.set(2,4,3);
                b2.set(5,6,2);

                System.out.println("The volume of b1 is " + b1.volume());
                System.out.println("The volume of b2 is " + b2.volume());
        }
}
```

The output would be –

```
        The volume of b1 is 24.0
        The volume of b2 is 60.0
```

In the above program, the ***Box*** class contains a method ***set()*** which take 3 parameters. Note that, the variables ***wd, ht*** and ***dp*** are termed as ***formal parameters*** or just ***parameters*** for a method. The values passed like 2, 4, 3 etc. are called as ***actual arguments*** or just ***arguments*** passed to the method.

2.5 Constructors

Constructor is a special type of member method which is invoked automatically when the object gets created. Constructors are used for object initialization. They have same name as that of the class. Since they are called automatically, there is no return type for them. Constructors may or may not take parameters.

Program 2.5

```
        class Box
        {
                double w, h, d;

                double volume()
                {
                        return w*h*d;
                }

                Box()               //ordinary constructor
                {
                        w=h=d=5;
                }

                Box(double wd, double ht, double dp)          //parameterized constructor
                {
                        w=wd;
                        h=ht;
                        d=dp;
                }
        }

        class BoxDemo
        {
```

```
        public static void main(String args[])
        {
                Box b1=new Box();
                Box b2=new Box();
                Box b3=new Box(2,4,3);

                System.out.println("The volumeof b1 is " + b1.volume());
                System.out.println("The volumeof b2 is " + b2.volume());
                System.out.println("The volumeof b3 is " + b3.volume());
        }
}
```

The output would be –

```
The volume of b1 is 125.0
The volume of b2 is 125.0
The volume of b3 is 24.0
```

When we create two objects ***b1*** and ***b2***, the constructor with no arguments will be called and the all the instance variables ***w, h*** and ***d*** are set to 5. Hence volume of ***b1*** and ***b2*** will be same (that is 125 in this example). But, when we create the object ***b3***, the ***parameterized constructor*** will be called and hence volume will be 24.

Few points about constructors:

- Every class is provided with a ***default constructor*** which initializes all the data members to respective *default values*. (Default for numeric types is zero, for character and strings it is null and default value for Boolean type is false.)
- In the statement

  classname ob= new classname();

  the term *classname()* is actually a constructor call.
- If the programmer does not provide any constructor of his own, then the above statement will call default constructor.
- If the programmer defines any constructor, then default constructor of Java can not be used.
- So, if the programmer defines any parameterized constructor and later would like to create an object without explicit initialization, he has to provide the default constructor by his own.

  For example, in **Program 2.5**, if we remove ordinary constructor, the statements like

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

  will generate error. To avoid the error, we should write a default constructor like –

  ```
  Box(){ }
  ```

  Now, all the data members will be set to their respective default values.

2.6 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 which invokes the method call. For example, in the **Program 2.5,** the method *volume()* can be written as –

```
double volume()
{
        return this.w * this.h * this.d;
}
```

Here, usage of ***this*** is not mandatory as it is implicit. But, in some of the situations, it is useful as explained in the next section.

Instance Variable Hiding

As we know, in Java, we can not have two local variables with the same name inside the same or enclosing scopes. (Refer **Program 1.7** and a **NOTE** after that program from Chapter 1, Page 16 & 17). But we 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**. That is, if we write following code snippet for a constructor in **Program 2.5**, we will not get an expected output –

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

Here note that, formal parameter names and data member names match exactly. To avoid the problem, we can use –

```
Box(double w, double h, double d)
{
        this.w=w;        //this.w refers to data member name and w refers to formal parameter
        this.h=h;
        this.d=d;
}
```

2.7 Garbage Collection

In C and C++, dynamically allocated variables/objects must be manually released using ***delete*** operator. But, in Java, this task is done automatically and is called as ***garbage collection.*** 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. Garbage collection occurs once in a while during the execution of the program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection.

2.8 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 can define 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 the ***finalize()*** method. The Java run time calls that method whenever it is about to recycle an object of that class.

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. Note that finalize( ) is only called just prior to garbage collection. It is not called when an object goes out-of-scope. So, we can not know when ***finalize()*** method is called, or we may be sure whether it is called or not before our program termination. Therefore, if at all our program uses some resources, we should provide some other means for releasing them and must not depend on ***finalize()*** method.

2.9 A Stack Class

To summarize the concepts of encapsulation, class, constructor, member initialization etc, we will now consider a program to implement stack operations.

Program 2.6

```
class Stack
{
        int st[] = new int[5];
        int top;

        Stack()
        {
                top = -1;
        }


        void push(int item)
        {
                if(top==4)
                        System.out.println("Stack is full.");
                else
                        st[++top] = item;
        }

        int pop()
        {
                if(top==-1)
                {
                        System.out.println("Stack underflow.");
                        return 0;
                }
                else
                        return st[top--];
        }
}

class StackDemo
{
        public static void main(String args[])
        {
                Stack mystack1 = new Stack();
                Stack mystack2 = new Stack();

                for(int i=0; i<5; i++)
                        mystack1.push(i);
                for(int i=5; i<10; i++)
                        mystack2.push(i);

                System.out.println("Contents of mystack1:");
                for(int i=0; i<5; i++)
                        System.out.println(mystack1.pop());

                System.out.println("Contents of mystack2:");
                for(int i=0; i<5; i++)
                        System.out.println(mystack2.pop());
        }
}
```