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.
- 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
        }
        ……….
}
```

- 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.

```
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);
    }
}
```

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.

Closer look at *new*

- The general form for object creation is –

  obj_name = new class_name();
- Here, ***class_name()*** is actually a constructor call.
- 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.
- If there is no enough memory in the heap when we use ***new*** for memory allocation, it will throw a run-time exception.

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.
- 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.

Introducing Methods

- A class can consist of instance variables and 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.

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

```
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();
    }
}
```

Returning a value

```
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());
    }
```

```
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());
  }
}
```

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.

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

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

    Box()
    {
        w=h=d=0;
    }

    Box(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(2,4,3);

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

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.

Instance variable Hiding

- As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- Interestingly, you 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 use following code snippet, we will not get an expected output –

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

- To avoid the problem, we can use –

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

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 only occurs once in a while during the execution of your 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.

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.
- This means that you cannot know when—or even if—**finalize( )** will be executed.
- Therefore, your program should provide other means of releasing system resources, etc., used by the object.
- It must not rely on **finalize( )** for normal program operation.

Method Overloading

- Having more than one method with a same name is called as method overloading.

- To implement this concept, the number of arguments and/or type of arguments for these methods should be different.

- Only the return type of the method is not sufficient for overloading.

Overloading Constructors

- One can have more than one constructor for a single class if the number and/or type of arguments are different.

Passing object as parameter

- Just similar to primitive types, even object of a class can also be passed as a parameter to any method.

```
class Equality
{
    int x,y;

    Equality(int a, int b)
    {       x=a;
            y=b;
    }

    public boolean test(Equality e)
    {
            if (e.x==x && e.y==y)
                    return true;
            else    return false;
    }

    public static void main(String args[])
    {
            Equality e1=new Equality(2,3);
            Equality e2=new Equality(2,3);
            Equality e3=new Equality(4,8);

            System.out.println("e1==e2: " + e1.test(e2));
            System.out.println("e1==e3: " + e1.test(e3));
    }
```

Using one object to initialize the other

- Sometimes, we may need to have a replica of one object.

- The usage of following statements **will not** serve the purpose.

  Box b1=new Box(2,3,4);

  Box b2=b1;

- In the above case, both b1 and b2 will be referring to same object, but not two different objects.

- So, we can write a constructor having a parameter of same class type to ***clone*** an object.

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

    Box(double ht, double wd, double dp)
    {
        h=ht; w=wd; d=dp;
    }
    Box (Box bx)
    {
        h=bx.h; w=bx.w; d=bx.d;
    }
    void vol()
    {
        System.out.println("Volume is " + h*w*d);
    }
    public static void main(String args[])
    {
        Box b1=new Box(2,3,4);

        Box b2=new Box(b1);

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

Argument Passing

- In Java, there are two ways of passing arguments to a method.
  - Call by value :
    - This approach copies the *value* of an argument into the formal parameter of the method.
    - Therefore, changes made to the parameter of the method have no effect on the argument.
  - Call by reference:
    - In this approach, a reference to an argument is passed to the parameter.
    - Inside the subroutine, this reference is used to access the actual argument specified in the call.
    - This means that changes made to the parameter will affect the argument used to call the subroutine.
- In Java, **when you pass a primitive type to a method, it is passed by value.**
- When you **pass an object to a method, they are passed by call-by-reference**.
- Keep in mind that when you create a variable of a class type, you are only creating a reference to an object.
- Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects are passed to methods by use of call-by-reference.
- Changes to the object inside the method *do* affect the object used as an argument.

```
class Test
{
    int a, b;
    Test(int i, int j)
    {
        a = i;
        b = j;
    }
    void meth(Test o)
    {
        o.a *= 2;
        o.b /= 2;
    }
}

class CallByRef
{
    public static void main(String args[])
    {
        Test ob = new Test(15, 20);
        System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
        ob.meth(ob);
        System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
    }
```

Output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

Returning Objects

- In Java, a method can return an object of user defined class.

```
class Test
{
    int a;
    Test(int i)
    {
        a = i;
    }

    Test incrByTen()
    {
        Test temp = new Test(a+10);
        return temp;
    }
}
```

```
class RetOb
{
  public static void main(String args[])
  {
    Test ob1 = new Test(2);
    Test ob2;

    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);

    ob2 = ob2.incrByTen();
    System.out.println("ob2.a after second
                increase: " + ob2.a);
  }
}
```

Output:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

Recursion

- A method which invokes itself either directly or indirectly is called as ***recursive*** *method.*

- Every recursive method should have a non-recursive terminating condition.

Access Control

- Encapsulation feature of Java provides a safety measure viz. ***access control.***

- Using ***access specifiers***, we can restrict the member variables of a class from outside manipulation.

- Java provides following access specifiers:
  - public
  - private
  - protected
- Along with above access specifiers, Java defines a *default access level*.

- Some aspects of access control are related to inheritance and package (a collection of related classes).

- The *protected* specifier is applied only when inheritance is involved.

- So, we will now discuss about only *private* and *public*.

- When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code.

- When a member of a class is specified as **private,** then that member can only be accessed by other members of its class.

- When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

- Usually, you will want to restrict access to the data members of a class—allowing access only through methods.

- Also, there will be times when you will want to define methods that are private to a class.

- An access specifier precedes the rest of a member's type specification.

- For example,

  public int x;
  private char ch;

  etc.

```
class Test
{   int a;
    public int b;
    private int c;
    void setc(int i)
    {  c = i;  }
    int getc()
    {  return c; }
}
class AccessTest
{
    public static void main(String args[])
    {
        Test ob = new Test();
        ob.a = 10;
        ob.b = 20;
        // ob.c = 100; // Error!
        ob.setc(100);
        System.out.println("a, b, and c: " + ob.a + " " + ob.b + " " + ob.getc());
    }
```

Static Members

- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- Instance variables declared as **static** are global variables.
- When objects of its class are declared, no copy of a **static** variable is made.
- Instead, all instances of the class share the same **static** variable.
- Methods declared as **static** have several restrictions:
  - They can only call other **static** methods.
  - They must only access **static** data.
  - They cannot refer to **this** or **super** in any way.
- If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded.

```
class UseStatic
{
    static int a = 3;
    static int b;

    static void meth(int x) //static method
    {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    static              //static block
    {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String args[])
    {
        meth(42);
    }
}
```

Output:

```
Static block initialized.
x = 42
a = 3
b = 12
```

- Outside of the class in which they are defined, **static** methods and variables can be used independently of any object.

- To do so, you need only specify the name of their class followed by the dot operator.

- The general form is –

  classname.method();

```
class StaticDemo
{
        static int a = 42;
        static int b = 99;
        static void callme()
        {
        System.out.println("a = " + a);
        }
}

class StaticByName
{
   public static void main(String args[])
   {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
   }
}
```

final Keyword:

- A variable can be declared as **final**.
- Doing so prevents its contents from being modified.
- This means that you must initialize a **final** variable when it is declared.
- For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

- It is a common coding convention to choose all uppercase identifiers for **final** variables.
- Variables declared as **final** do not occupy memory on a per-instance basis.
- Thus, a **final** variable is essentially a constant.