UNIT 3 and UNIT 4

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***.)

```
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 –

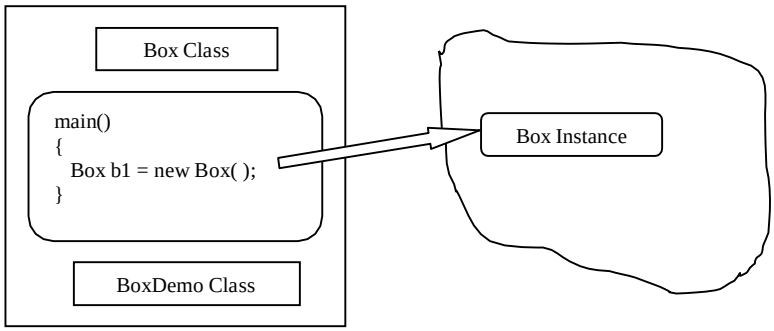
```
Volume of Box1 is 24.0
Volume of Box1 is 60.0
```

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 –



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;
```

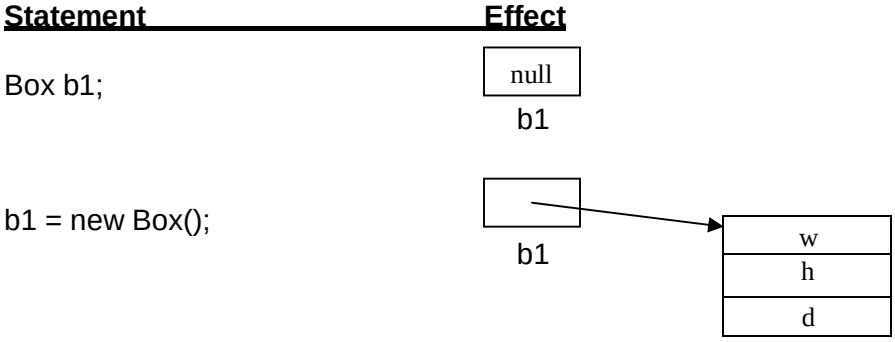
This statement will not actually create any physical object, but the object name *b1* can just ***refer*** to the actual object on the heap after memory allocation as follows –

```
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 –



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.

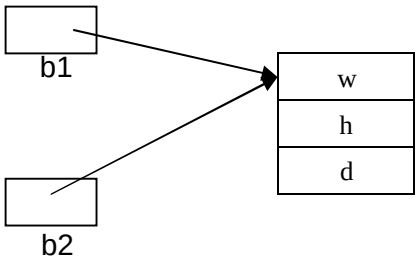
With the term *dynamic memory allocation*, we can understand that the keyword *new* allocates memory for the object during runtime. So, depending on the user's requirement memory will be utilized. This will avoid the problems with static memory allocation (either shortage or wastage of memory during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

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.

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 –

```
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.

```
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.

```
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.

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()           //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, the above program, 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.

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;
}
```

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.

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.

Overloading Methods

Having more than one method with a same name is called as method overloading. To implement this concept, the constraints are:

- the number of arguments should be different, and/or
- Type of the arguments must be different.

NOTE that, only the return type of the method is not sufficient for overloading.

```
class Overload
{
      void test()           //method without any arguments
      {
            System.out.println("No parameters");
      }

      void test(int a)      //method with one integer argument
      {
            System.out.println("Integer a: " + a);
      }

      void test(int a, int b) //two arguments
      {
        System.out.println("With two arguments : " + a + " " + b);
      }

      void test(double a)         //one argument of double type
      {
            System.out.println("double a: " + a);
      }
}

class OverloadDemo
{
      public static void main(String args[])
      {
            Overload ob = new Overload();

            ob.test();
            ob.test(10);
            ob.test(10, 20);
            ob.test(123.25);
      }
}
```

Overloading Constructors

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

```
class OverloadConstruct
{
      int a, b;
      OverloadConstruct()
      {
            System.out.println("Constructor without arguments");
      }

      OverloadConstruct(int x)
      {
            a=x;
            System.out.println("Constructor with one argument:"+a);
      }

      OverloadConstruct(int x, int y)
      {
            a=x;
            b=y;
            System.out.println("Constructor with two arguments:"+ a +"\t"+ b);
      }
}

class OverloadConstructDemo
{
      public static void main(String args[])
      {
            OverloadConstruct ob1= new OverloadConstruct();
            OverloadConstruct ob2= new OverloadConstruct(10);
            OverloadConstruct ob3= new OverloadConstruct(5,12);
      }
}
```

Output:

```
      Constructor without arguments
      Constructor with one argument: 10
      Constructor with two arguments: 5        12
```

Using Objects as Parameters

Just similar to primitive types, even object of a class can also be passed as a parameter to any method. Consider the example given below –

```
      class Test
      {
            int  a,  b;
            Test(int i, int j)
            {
```

```
            a = i;
            b = j;
      }

      boolean equals(Test ob)
      {
            if(ob.a == this.a && ob.b == this.b)
                  return true;
            else
                  return false;
      }
}

class PassOb
{
      public static void main(String args[])
      {
            Test ob1 = new Test(100, 22);
            Test ob2 = new Test(100, 22);
            Test ob3 = new Test(-1, -1);
            System.out.println("ob1 == ob2: " + ob1.equals(ob2));
            System.out.println("ob1 == ob3: " + ob1.equals(ob3));
      }
}
```

```
Output:
      ob1 == ob2: true
      ob1 == ob3: false
```

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)                  //observe this constructor
      {
            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);   //initialize b2 using b1
              b1.vol();
              b2.vol();
        }
  }
```

Output:

```
  Volume is 24
  Volume is 24
```

A Closer Look at 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 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)
      {
      *= 2;
      /= 2;
      }
}
class CallByRef
{
      public static void main(String args[])
      {
```

```
            Test ob = new Test(15, 20);
            System.out.println("before call: " + ob.a + " " + ob.b);
            ob.meth(ob);
            System.out.println("after call: " + ob.a + " " + ob.b);
      }
}
```

Output:
before call: 15 20
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
```

Introducing 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;
```

Consider a program given below –

```
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;        // inclusion of this line is Error!
            ob.setc(100);
            System.out.println("a, b, and c: " + ob.a + " " + ob.b + " "
                                                + ob.getc());
      }
}
```

Understanding *static*

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();

Consider the following program:

```
class StaticDemo
{
      static int a = 42;
      static int b = 99;

      static void callme()
      {
            System.out.println("Inside static method, a = " + a);
      }
}
```

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

Output:

Inside static method, a = 42
Inside main, b = 99

Arrays Revisited

Arrays have been discussed earlier. An important point to be noted with arrays is: arrays are implemented as objects in Java. Because of this, we can use a special instance variable ***length*** to know the size of an array.

```
class Test
{
      public static void main(String args[])
      {
            int  a1[]=new  int[10];
            int a2[]={1, 2, 3, 4, 5};
            int a3[]={3, 8, -2, 45, 9, 0, 23};

            System.out.println("Length of a1 is" + a1.length);
            System.out.println("Length of a2 is" + a2.length);
            System.out.println("Length of a3 is" + a3.length);
      }
}
```

Output:

Length of a1 is 10
Length of a2 is 5
Length of a3 is 7

Inheritance

Inheritance is one of the building blocks of object oriented programming languages. It allows creation of classes with hierarchical relationship among them. Using inheritance, one can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a ***superclass***. The class that does the inheriting is called a ***subclass***. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and add its own, unique elements. Through inheritance, one can achieve re-usability of the code.

In Java, inheritance is achieved using the keyword ***extends.*** The syntax is given below:

```
class A               //super class
{
      //members of class A
}

class B extends A     //sub class
{
      //members of B
}
```

Consider a program to understand the concept:

```
class A
{
      int i, j;

      void showij()
      {
            System.out.println("i and j: " + i + " " + j);
      }
}

class B extends A
{
      int k;
      void showk()
      {
            System.out.println("k: " + k);
      }
      void sum()
      {
            System.out.println("i+j+k: " + (i+j+k));
      }
}

class SimpleInheritance
{
      public static void main(String args[])
      {
            A superOb = new A();
            B subOb = new B();
            superOb.i = 10;
            superOb.j  =  20;
            System.out.println("Contents of superOb: ");
            superOb.showij();

            subOb.i = 7;
            subOb.j = 8;
            subOb.k = 9;
            System.out.println("Contents of subOb: ");
```

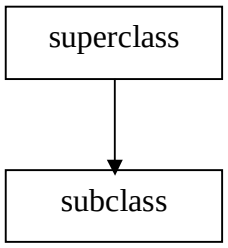
```
                subOb.showij();
                subOb.showk();

                System.out.println("Sum of i, j and k in subOb:");
                subOb.sum();
        }
}
```

Note that, private members of the super class can not be accessed by the sub class. The subclass contains all non-private members of the super class and also it contains its own set of members to achieve specialization.

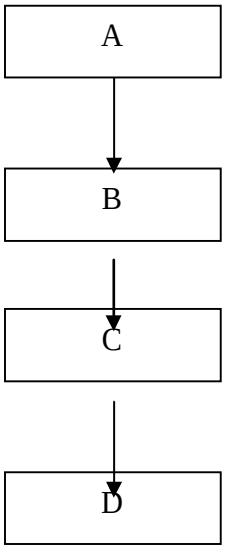
Type of Inheritance

- **Single Inheritance:** If a class is inherited from one parent class, then it is known as single inheritance. This will be of the form as shown below –



The previous program is an example of single inheritance.

- **Multilevel Inheritance:** If several classes are inherited one after the other in a hierarchical manner, it is known as multilevel inheritance, as shown below –



A Superclass variable can reference a subclass object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. Consider the following for illustration:

```
class Base
{
      void dispB()
      {
            System.out.println("Super class " );
      }
}
class Derived extends Base
{
      void dispD()
      {
            System.out.println("Sub class ");
      }
}

class Demo
{
      public static void main(String args[])
      {
            Base b = new Base();
            Derived d=new Derived();

            b=d;       //superclass reference is holding subclass object
            b.dispB();
            //b.dispD();    error!!
      }
}
```

Note that, the ***type of reference variable*** decides the members that can be accessed, ***but not the type of the actual object.*** That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.

Using *super*

In Java, the keyword *super* can be used in following situations:

- To invoke superclass constructor within the subclass constructor
- To access superclass member (variable or method) when there is a duplicate member name in the subclass

Let us discuss each of these situations:

- **To invoke superclass constructor within the subclass constructor:** Sometimes, we may need to initialize the members of super class while creating subclass object. Writing such a code in subclass constructor may lead to redundancy in code. For example,

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

```
        Box(double wd, double ht, double br)
        {
            w=wd; h=ht; b=br;
        }
    }
    class ColourBox extends Box
    {
        int colour;
        ColourBox(double wd, double ht, double br, int c)
        {
            w=wd; h=ht; b=br;          //code redundancy
            colour=c;
        }
    }
```

Also, if the data members of super class are private, then we can't even write such a code in subclass constructor. If we use super() to call superclass constructor, then **it must be the first statement** executed inside a subclass constructor as shown below –

```
class Box
{
    double w, h, b;
    Box(double wd, double ht, double br)
    {
        w=wd; h=ht; b=br;
    }
}

class ColourBox extends Box
{
    int colour;
    ColourBox(double wd, double ht, double br, int c)
    {
        super(wd, ht, br);          //calls superclass constructor
        colour=c;
    }
}

class Demo
{
    public static void main(String args[])
    {
        ColourBox b=new ColourBox(2,3,4, 5);
    }
}
```

Here, we are creating the object *b* of the subclass `ColourBox` . So, the constructor of this class is invoked. As the first statement within it is **`super(wd, ht, br)`**, the constructor of superclass `Box` is invoked, and then the rest of the statements in subclass constructor `ColourBox` are executed.

- **To access superclass member variable when there is a duplicate variable name in the subclass:** This form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
class A
{
      int a;
}

class B extends A
{
      int a;     //duplicate variable a

      B(int x, int y)
      {
            super.a=x;       //accessing superclass a
            a=y;             //accessing own member a
      }

      void disp()
      {
            System.out.println("super class a: "+ super.a);
            System.out.println("sub class a: "+ a);
      }
}

class SuperDemo
{
   public static void main(String args[])
   {
      B ob=new B(2,3);
      ob.disp();
   }
}
```

Creating Multilevel Hierarchy

Java supports multi-level inheritance. A sub class can access all the non-private members of all of its super classes. Consider an illustration:

```
class A
{     int a;
}

class B extends A
{     int b;
}

class C extends B
{     int c;
```

```
      C(int x, int y, int z)
      {
            a=x;  b=y;  c=z;
      }
      void disp()
      {
            System.out.println("a= "+a+ " b= "+b+" c="+c);
      }
}

class MultiLevel
{
   public static void main(String args[])
  {
      C ob=new C(2,3,4);
      ob.disp();
   }
}
```

When Constructors are called

When class hierarchy is created (multilevel inheritance), the constructors are called in the order of their derivation. That is, the top most super class constructor is called first, and then its immediate sub class and so on. If *super* is not used in the sub class constructors, then the default constructor of super class will be called.

```
class A
{
      A()
      {
            System.out.println("A's constructor.");
      }
}

class B extends A
{
      B()
      {
            System.out.println("B's constructor.");
      }
}
class C extends B
{
      C()
      {
            System.out.println("C's constructor.");
      }
}
class CallingCons
{
  public static void main(String args[])
```

```
    {
      C c = new C();
    }
}
```

Output:

```
A's constructor
B's constructor
C's constructor
```

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to ***override*** the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

```
class A
{
      int  i,  j;
      A(int a, int b)
      {
            i = a;
            j = b;
      }
      void show()      //suppressed
      {
            System.out.println("i and j: " + i + " " + j);
      }
}
class B extends A
{
      int k;
      B(int a, int b, int c)
      {
            super(a, b);
            k = c;
      }
      void show()      //Overridden method
      {
            System.out.println("k: " + k);
      }
}
class Override
{
    public static void main(String args[])
    {
            B subOb = new B(1, 2, 3);
            subOb.show();
     }
}
```

Output:

k: 3

Note that, above program, only subclass method *show()* got called and hence only *k* got displayed. That is, the ***show()*** method of super class is suppressed. If we want superclass method also to be called, we can re-write the ***show()*** method **in subclass** as –

```
void show()
{
      super.show();    // this calls A's show()
      System.out.println("k: " + k);
}
```

Method overriding occurs *only* when the names and the type signatures of the two methods (one in superclass and the other in subclass) are identical. If two methods (one in superclass and the other in subclass) have same name, but different signature, then the two methods are simply overloaded.

Dynamic Method Dispatch

Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch.* Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Java implements run-time polymorphism using dynamic method dispatch. We know that, a superclass reference variable can refer to subclass object. Using this fact, Java resolves the calls to overridden methods during runtime. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

```
class A
{
      void callme()
      {
         System.out.println("Inside A");
      }
}
class B extends A
{
      void callme()
      {
         System.out.println("Inside B");
      }
}

class C extends A
{
      void callme()
      {
```

```
            System.out.println("Inside C");
        }
}
class Dispatch
{
   public static void main(String args[])
   {
      A a = new A();
      B b = new B();
      C c = new C();

      A r;        //Superclass reference
      r = a;      //holding subclass object
      r.callme();
      r = b;
      r.callme();
      r = c;
      r.callme();
    }
}
```

Why overridden methods?

Overridden methods are the way that Java implements the “one interface, multiple methods” aspect of polymorphism. superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses. Dynamic, run-time polymorphism is one of the most powerful mechanisms that objectoriented design brings to bear on code reuse and robustness.

Using Abstract Classes

Sometimes, the method definition will not be having any meaning in superclass. Only the subclass (specialization) may give proper meaning for such methods.In such a situation, having a definition for a method in superclass is absurd. Also, we should enforce the subclass to override such a method. A method which does not contain any definition in the superclass is termed as ***abstract method***. Such a method declaration should be preceded by the keyword ***abstract***. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass.

A class containing at least one abstract method is called as ***abstract class.*** Abstract classes can not be instantiated, that is one cannot create an object of abstract class. Whereas, a reference can be created for an abstract class.

```
abstract class A
{
      abstract void callme();
      void callmetoo()
      {
            System.out.println("This is a concrete method.");
      }
}
```

```
class B extends A
{
      void callme() //overriding abstract method
      {
            System.out.println("B's implementation of callme.");
      }
}

class AbstractDemo
{
      public static void main(String args[])
      {
            B b = new B(); //subclass object
            b.callme();      //calling abstract method
            b.callmetoo();  //calling concrete method
      }
}
```

Example: Write an abstract class *shape,* which has an abstract method *area()*. Derive three classes *Triangle, Rectangle* and *Circle* from the *shape* class and to override *area()*. Implement run-time polymorphism by creating array of references to supeclass. Compute area of different shapes and display the same.

Solution:

```
abstract class Shape
{
      final double PI= 3.1416;
      abstract double area();
}

class Triangle extends Shape
{
      int b, h;
      Triangle(int x, int y)      //constructor
      {
            b=x;
            h=y;
      }

      double area()    //method overriding
      {
            System.out.print("\nArea of Triangle is:");
            return 0.5*b*h;
      }
}

class Circle extends Shape
{
      int r;
```

```
        Circle(int rad)       //constructor
        {
              r=rad;
        }

        double area()         //overriding
        {
              System.out.print("\nArea of Circle is:");
              return PI*r*r;
        }
  }

  class Rectangle extends Shape
  {
        int a, b;
        Rectangle(int x, int y)     //constructor
        {
              a=x;
              b=y;
        }
        double area()         //overriding
        {
              System.out.print("\nArea of Rectangle is:");
              return a*b;
        }
  }

  class AbstractDemo
  {
     public static void main(String args[])
    {
       Shape r[]={new Triangle(3,4), new Rectangle(5,6),new Circle(2)};

        for(int i=0;i<3;i++)
              System.out.println(r[i].area());
     }
  }
```

Output:

```
Area of Triangle is:6.0
Area of Rectangle is:30.0
Area of Circle is:12.5664
```

Note that, here we have created array *r*, which is reference to *Shape* class. But, every element in *r* is holding objects of different subclasses. That is, r[0] holds *Triangle* class object, r[1] holds *Rectangle* class object and so on. With the help of array initialization, we are achieving this, and also, we are calling respective constructors. Later, we use a *for-loop* to invoke the method *area()* defined in each of these classes.

Using *final*

The keyword *final* can be used in three situations in Java:

- To create the equivalent of a named constant.
- To prevent method overriding
- To prevent Inheritance

To create the equivalent of a named constant: 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.

To prevent method overriding: Sometimes, we do not want a superclass method to be overridden in the subclass. Instead, the same superclass method definition has to be used by every subclass. In such situation, we can prefix a method with the keyword *final* as shown below –

```
class A
{
      final void meth()
      {
            System.out.println("This is a final method.");
      }
}
class B extends A
{
      void meth()  // ERROR! Can't override.
      {
            System.out.println("Illegal!");
      }
}
```

To prevent Inheritance: As we have discussed earlier, the subclass is treated as a specialized class and superclass is most generalized class. During multi-level inheritance, the bottom most class will be with all the features of real-time and hence it should not be inherited further. In such situations, we can prevent a particular class from inheriting further, using the keyword *final*. For example –

```
final class A
{
      // ...
}
class B extends A      // ERROR! Can't subclass A
{
      // ...
}
```

Note:

- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations