BCA PART- III

PAPER – XIX

TOPIC: OBJECT CLASSES AND METHODS

PREPARED BY: DR. KIRAN PANDEY

(SCHOOL OF COMPUTER SCIENCE)

EMAIL ID: kiranpandey.nou@gmail.com

Introduction

Java is a true object-oriented language. Anything we wish to represent in a Java program must be encapsulated in a class that defines the *state* and *behavior* of the basic program components known as *objects*. Classes create objects and objects use methods to communicate between them. That is all about object-oriented programming.

Classes provide a convenient method for packing together a group of logically related data items and functions that work on them. In Java, the data items are called *fields* and the functions are called *methods*. Calling a specific method in an object is described as sending the object a message.

A class is essentially a description of how to make an object that contains fields and methods. It provides a sort of *template* for an object and behaves like a basic data type such as **int**. It is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic OOP concepts such as *encapsulation*, *inheritance* and *polymorphism*.

Defining a Class

As stated earlier, a class is a user-defined data type with a template that serves to define its properties. Once the class type has been defined, we can create "variables" of that type using declaration that are similar to the basic type declarations. In Java, these variables are termed as *instances* of classes, which are the actual *objects*. The basic form of a class definition is:

```
class   classname    [extends   superclassname]
{
        [ fields declaration; ]
        [ methods declaration; ]
}
```

Everything inside the square brackets is optional. This means that the following would be a valid class definition:

```
class  Empty
{
}
```

Because the body is empty, this class does not contain any properties and therefore cannot do anything. We can, however, compile it and even create objects using it, C++ *programmers may note that there is no semicolon after closing brace*.

classname and *superclassname* are any valid Java identifiers. The keyword **extends** indicates that the properties of the *superclassname* class are extended to the *classname* class. This concept is known as *inheritance* and is discussed in Section 6.11. Fields and methods are declared inside the body.

6.3 Fields Declaration

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called *instance variables* because they are created whenever an object of the class is instantiated. We can declare the instance variables exactly the same way as we declare local variables. Example:

```
class  Rectangle
{
        int  length;
        int  width;
}
```

The class **Rectangle** contains two integer type instance variables. It is allowed to declare them in one line as

```
int  length,  width;
```

Remember these variables are only declared and therefore no storage space has been created in the memory. Instance variables are also known as *member variables*.

Methods Declaration

A class with only data fields (and without methods that operate on that data) has no life. The objects created by such a class cannot respond to any message. We must therefore add methods that are necessary for manipulating the data contained in the class. Methods are declared inside the body of the class but immediately after the declaration of instance variables. The general form of a method declaration is

```
type  methodname  (parameter.list)
{
        method-body;
}
```

Method declarations have four basic parts:

- The name of the method (*methodname*)
- The type of the value the method returns (*type*)
- A list of parameters (*parameter-list*)
- The body of the method

The *type* specifies the type of value the method would return. This could be a simple data type such as **int** as well as any class type. It could even be **void** type, if the method does not return any value. The *methodname* is a valid identifier. The *parameter* list is always enclosed in parentheses. This list contains variable names and types of all the value we want to give to the method as input. The variables in the list are separated by commas. In the case where no input data required, the declaration must retain the empty parentheses. Examples:

```
(int m, float x, float y)    // Three parameters
( )                          // Empty list
```

The *body* actually describes the operations to be performed on the data. Let us consider the **Rectangle** class again and add a method **getData ( )** to it.

```
class Rectangle
{
      int length;
      int width;
      void getData (int x, int y) // Method declaration
      {
```

```
                length = x;
                length = y;
            }
        }
```

Note that the method has a return type of **void** because it does not return any value. We pass two integer values to the method which are then assigned to the instance variables **length** and **width**. The **getData** method is basically added to provide values to the instance variables. Notice that we are able to use directly **length** and **width** inside the method.

Let us add some more properties to the class. Assume that we want to compute the area of the rectangle defined by the class. This can be done as follows:

```
        class  Rectangle
        {
            int  length,  width;  // Combined declaration
            void  getData(int x, int y)
            {
                length = x;
                width = y;
            }
            int  recArea( ) // Declaration of another method
                int area = length * width;
                return (area);
            }
        }
```

The new method **rectArea( )** computes area of the rectangle and returns the result. Since the result would be an integer, the return type of the method has been specified as **int.** Also note that the parameter list is empty.

Remember that while the declaration of instance variables (and also local variables) can be combined as

```
        int  length,  width;
```

the parameter list used in the method header should always be declared independently separated by commas. That is,

```
        void  getData  (int x, y) // Incorrect
```

is illegal.

Now, our class **Rectangle** contains two instance variables and two methods. We can add more variables and methods, if necessary.

Most of the times when we use classes, we will have many methods and variables within the class. Instance variables and methods in classes are accessible by all the methods in the class but a method cannot access the variables declared in other methods.

Example:

```
        class Access
        {
            int x;
            void method( )
            {
                int y = ;
```

```
            x = 10 ;          // legal
            y = x ;           // legal
        }
        void method2( )
        {
            int z ;
            x = 5 ;           // legal
            z = 10 ;          // legal
            y = 1 ;           // illegal
        }
    }
```

Creating Objects

As pointed out earlier, an object in Java is essentially a block of memory that contains space to store all the instance variables. Creating an object is also referred to as *instantiating* an object.

Objects in Java are created using the **new** operator. The **new** operator creates an object of the specified class and returns a reference to that object. Here is an example of creating an object of type **Rectangle.**

```
Rectangle rect1;              // declare the object
Rect1 = new Rectangle( );     // instantiate the object
```

The first statement declares a variables to hold the object reference and the second one actually assigns the object reference to the variable. The variable **rect1** is now an object of the **Rectangle** class (see Fig. 6.1)

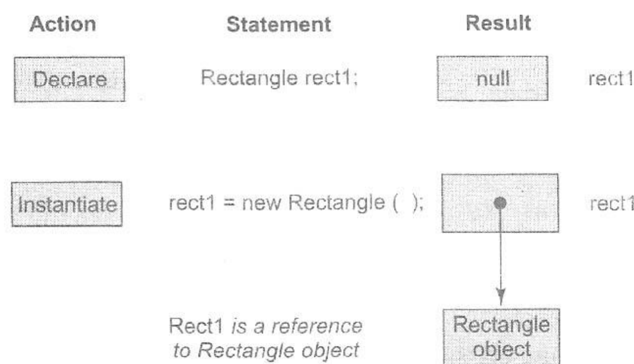

Fig. 6.1 Creating object references

Both statements can be combined into one as shown below:

```
Rectangle rect1 = new Rectangle( );
```

The method **Rectangle**( ) is the default constructor of the class. We can create any number of objects of **Rectangle**. Example:

```
Rectangle rect1 = new Rectangle( );
Rectangle rect2 = new Rectangle( );
```

and so on.

It is important to understand that each object has its own copy of the instance variables of its class. This means that any changes to the variables of one object have no effect on the variables of another. It is also possible to create two or more references to the same object (see Fig. 6.2)

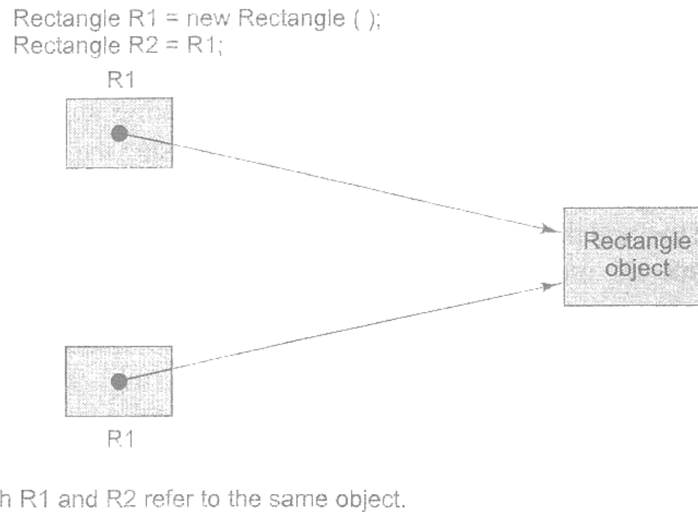

Fig. 6.2 Assigning one object reference variable to another

Accessing Class Members

Now that we have created objects, each containing its own set of variables, we should assign values to these variables in order to use them in our program. Remember, all variables must be assigned values before they are used. Since we are outside the class, we cannot access the instance variables and the methods directly. To do this, we must use the concerned object and the dot operator as shown below:

```
objectname.variablename = value;
objectname.methodname(paramewter-list);
```

Here *objectname* is the name of the object, *variablename* is the name of the instance variable inside the object that we wish to access, *methodname* is the method that we wish to call, and *parameter-list* is a comma separated list of "actual values" (or expression) that must match in type and number with the parameter list of the methodname declared in the class. The instance variables of the **Rectangle** class may be accessed and assigned values as follows:

```
rect1.length     = 15;
rect1.width      = 10;
rect2.length     = 20;
rect2.width      = 12;
```

Note that the two objects **rect1** and **rect2** store different values as shown below:

	rec1		**rec2**
rect1.length	15	rec2.length	20
rect1.width	10	rec2.width	12

This is one way of assigning values to the variables in the objects. Another way and more convenient way of assigning values to the instance variables is to use a method that is declared inside the class.

In our caste, the method **getData** can be used to do this work. We can call the **getData** method on any **Rectangle** object to set the values of both **length** and **width**. Here is the code segment to achieve this.

```
Rectangle rect1 = new Rectangle( ); // Creating an object
rect1.getData(15, 10); // Calling the method using the object
```

This code creates **rect1** object and then passes in the values 15 and 10 for the **x** and **y** parameters of the method **getData**. This method then assigns these values to **length** and **width** variables respectively. For the sake of convenience, the method is again shown below:

```
void getData  (int x, int y)
{
    length = x;
    width  = y;
}
```

Now that the object **rect1** contains values for its variables, we can compute the area of the rectangle represented by **rect1**. This again can be done in two ways.

- The first approach is to access the instance variables using the dot operator and compute the area. That is,

  ```
  int area1`= rect1.length * rect1.width;
  ```

- The second approach is to call the method **rectArea** declared inside the class. That is,

  ```
  int area1 = rect1.rectArea( );     // Calling the method
  ```

Program 6.1 illustrates the concepts discussed so far.

Program 6.1 Application of classes and objects

```
class Rectangle
{
   int length, Width;                  //   Declaration of variables
   void getData x, int y)              //   Definition of method
   {
       length = x;
       widht  = y;
   }
   int rectArea( )                     //   Definition of another method
   {
       int area = length * width;
       return (area);
   }
}
class RectArea                         //   Class with main method
{
   public static void main (String args[ ])
   {
       int area1, area2;
       Rectangle rect1 = new Rectangle();     //  Creating objects
       Rectangle rect2 = new Rectangle( );
       Rect1.length = 15;                     //  Accessing variables
       area1 = rect1.length * rect1.width;
       rect2.getData (20, 12);                //  Accessing methods
       area2 = rect2.rectArea( );
       System.out.println("Area1 = " + area1);
       System.out.println("Area2 = " + area2);
   }
}
```

Program 6.1 would output the following:

```
Area1 = 150
Area2 = 240
```

Constructors

We know that all objects that are created must be given initial values. We have done this earlier using two approaches. The first approach uses the dot operator to access the instance variables and then assigns values to them individually. It can be a tedious approach to initialize all the variables of all the objects.

The second approach takes the help of a method like **getData** to initialize each object individually using statements like,

```
Rect1.getData(15, 10);
```

It would approach be simpler and more concise to initialize an object when it is first created. Java supports a special type of method, called a *constructor*, that enables an object to initialize itself when it is created.

Constructors have the same name as the class itself. Secondly, they do not specify a return type, not even **void**. This is because they return the instance of the class itself.

Let us consider our **Rectangle** class again. We can now replace the **getData** method by a constructor method as shown below:

```
class Rectangle
{
    int length;
    int width;
    Rectangle(int x, int y)    //   Constructor method
    {
        length = x;
        width  = y;
    }
    int recArea( )
    {
        return (length * width);
    }
}
```

Program 6.2 illustrates the use of a constructor method to initialize an object at the time of its creation.

Program 6.2 Application of constructors

```
class Rectangle
{
    int length, width;
    Rectangle (int x, int y)                //    Defining constructor
    {
        length = x;
        width  = y;
    }
    int rectArea( )
    {
        return (length * width);
    }
}
class RectangleArea
{
        public static void main (string args[ ])
    {
        Rectangle rect1 = new Rectangle(15, 10);  // Calling constructor
        int area1 = rect1.rectArea( );
        System.out.println("Area = "+ arae1);
    }
}
```

Output of Program 6.2

```
Area1 = 150
```

Methods Overloading

In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called *method overloading*. Method overloading is used when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, Java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as *polymorphism*.

To create an overloaded method, all we have to do is to provide several different method definitions in the class, all with the same name, but with different parameter lists. The difference may either be in the number or type of arguments. That is, each parameter list should be unique. Not that the method's return type does not play any role in this. Here is an example of creating an overloaded method.

```
class Room
{
    float length;
    float breadth;
    Room(float x, float y)          //  consturctor1
    {
        length = x;
        length  = y;
    }
    Room(float x);                  //  constructor2
    {
        length = breadth = x;
    }
    int area( )
    {
        return (length * breadth);
    }
}
```

Here, we are overloading the constructor method **Room( )**. An object representing a rectangular room will be created as

```
Room  room1  =  new  Room(25.0, 15.0);  //  using constructor1
```

On the other hand, if the room is square, then we may create the corresponding object as

```
Room  room2  =  new  Room(20.0);      //  using constructor1
```

Static Members

We have seen that a class basically contains two sections. One declares variables and the other declares methods. These variables and methods are called *instance* and *instance methods*. This is because every time the class is instantiated, a new copy of each of them is created. They are accessed using the objects (with dot operator).

Let us assume that we want to define a member that is common to all the objects and accessed without using a particular object. That is, the member belongs to the class as a whole rather than the objects created from the class. Such members can be defined as follows:

```
static  int count;
static  int max(int x, int y);
```

The members that are declared **static** as shown above are called *static members*. Since these members are associated with the class itself rather than individual objects, the static variables and static methods are often referred to as *class variables* and *class methods* in order to distinguish them from their counterparts, instance variables and instance methods.

Static variables are used when we want to have a variable common to all instances of a class. One of the most common examples is to have a variable that could keep a count of how many objects of a class have been created. Remember, Java creates only one copy for a static variable which can be used even if the class is never actually instantiated.

Like static variables, static methods can be called without using the objects. They are also available for use by other classes, methods that are of general utility but do not directly affect an instance of that class are usually declared as class methods. Java class libraries contain a large number of class methods. For example, the **Math** class of Java library defines many static methods to perform math operations that can be used in any program. We have used earlier statements of the types.

```
float x = Math.sqrt(25.0);
```

The method **sqrt** is a class method (or static method) define in **Math** class.

We can define our own static methods as shown in Program 6.3.

Program 6.3 Defining and using static members

```
class Matheoperation
{
      Static float mul(float x, float y)
      {
            return x*y;
      }
      static  float  divide (float x,  float y)
      {
            return x/y;
      }
}
Class MathApplication
{
      public void static main(string args[ ])
      {
            float a = MathOperation.mul(4.0, 5.0);
            float b = MathOperation.divide(a, 2.0);
            System.out.println("b = "+ b);
      }
}
```

Output of Program 6.3

```
b = 10.0
```

Note that the static methods are called using class names. In fact, no objects have been created for use. Static methods have several restrictions:

1. They can only call other **static** methods.

2. They can only access **static** data.

3. They cannot refer to **this** or **super** in any way.

Nesting of Methods

We discussed earlier that a method of a class can be called only by an object of that class (or class itself, in the case of static methods) using the dot operator. However, there is an exception to this. A method can be called by using only its name by another method of the same class. This is known as *nesting of methods*.

Program 6.4 illustrates the nesting of methods inside a class. The class **Nesting** defines one constructor and two methods, namely **largest( )**. The method **display**( ) calls the method **largest**( ) to determine the largest of the two numbers and then displays the result.

Program 6.4 Nesting of methods

```
class  Nesting
{
   int m,  n;
   Nesting (int x, int y)         //   constructor method
      {
         m = x;
         n = y;
      }
      int largest( )
      {
         if (m >= n)
             returns(m);
         else
             returns(n);
      }
      void display( )
      {
         int  large = largest( ); //   calling a method
         System.out.println (“Largest value = ” + large);
      }
}
class  NestingTest
{
   public  static  void  main(String  args[  ])
   {
      Nesting  nest  =  new  Nesting(50, 40);
      Nest.display(  );
   }
}
```

Output of Program 6.4 would be:

```
Largest  value  =  50
```

A method can call any number of methods. It is also possible for a called method to call another method. That is, **method1** may call **method2**, which in turn may call **method**3.

Inheritance : Extending a Class

Reusability is yet another aspect of OOP paradigm. It is always nice if we could reuse something that already exists rather than creating the same all over again. Java supports concept. Java classes can be reused in several ways. This is basically done by creating new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called *inheritance*. The old class is known as the *base class* or *super class* or *parent class* and the new one is called the *subclass* or *derived class* or *child class*.

The inheritance allows subclasses to inherit all the variables and methods of their parent classes. Inheritance may take different forms:

- Single inheritance (only one super class)
- Multiple inheritance (several super classes)
- Hierarchical inheritance (one super class, many subclasses)
- Multilevel inheritance (Derived from a derived class)

These forms of inheritance are shown in Fig. 6.3. Java does not directly implement multiple inheritance. However, this concept is implemented using a secondary inheritance path in the form of *interfaces*.

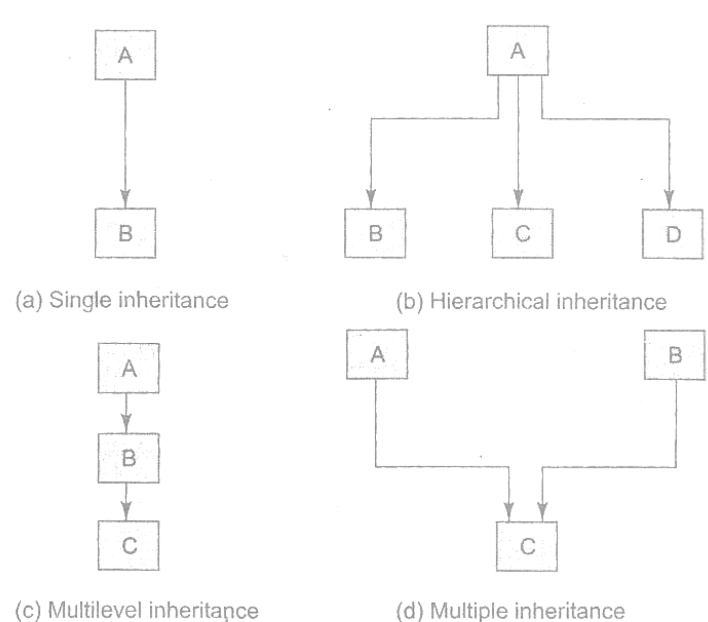

Fig. 6.3 Forms of inheritance

Defining a Subclass

A subclass is define as follows:

```
class  subclassname  extends  superclassname
{
      Variables  declaration;
      methods  declaration;
}
```

The keyword **extends** signifies that the properties of the *superclassname* are extended to the *subclassname*. The subclass will now contain its own variables and methods as well those of the superclass. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it. Program 6.5 illustrates the concept of single inheritance.

Program 6.5 Application of single inheritance

```
class  Room
{
       int  length;
       int  breadth;
       Room(int x, int y)
       {
           length  = x;
           breadth = y;
       }
       int  area( )
       {
           return (length * breadth);
       }
}
class  Bedroom  extend  Room              //   Inheriting Room
       int  height;
       BedRoom(int x, int y, int z)
       {
           Super(x, y)                    //   pass values to superclass
       }
       int  volume( )
       {
           return  (length * breadth * height);
       }
}
class  InherTest
{
       public static void main(String args[ ])
       {
           BedRoom room1 = new BedRoom(14,12,10);
           int area1 = room1.area( );              //   superclass method
           int volume1 = room1.volume( );         //   baseclass method
           System.out.println("Area1 = "+ area1);
           System.out.println("Volume ="+ volume);
       }
}
```

The output of Program 6.5

```
Area1 = 168
Volume1 = 1680
```

The program defines a class **Room** and extends it to another class **BedRoom**. Note that the class **BedRoom** defines its own data members and methods. The subclass BedRoom now includes three instance variables, namely, **length, breadth** and **height** and two methods, **area** and **volume**.

The constructor in the derived class uses the **super** keyword to pass values that are required by the constructor. That statement

```
BedRoom room1 = new BedRoom(14,12,10);
```

Calls first the **BedRoom** constructor method, which in turn class the **Room** constructor method by using the **super** keyword.

Finally, the object **room1** of the subclass **BedRoom** calls the method **area** defined in the super class as well as the method volume defined in the subclass itself.

Subclass Constructor

A subclass constructor is used to construct the instance variables of both the subclass and the superclass. The subclass constructor uses the keyword **super** to invoke the constructor method of the superclass. The keyword **super** is used subject to the following conditions.

- **super** may only be used within a subclass constructor method
- The call to superclass constructor must appear as the first statement within the subclass constructor
- The parameters in the **super** call must match the order and type of the instance variable declared in the superclass.

Program 6.5 illustrated the use of **super( )** method for passing parameters to a superclass.

Multilevel Inheritance

A common requirement in object-oriented programming is the use of a derived class as a super class. Java supports this concept and uses it extensively in building its class library. This concept allows us to build a chain of classes as shown in Fig. 6.4.

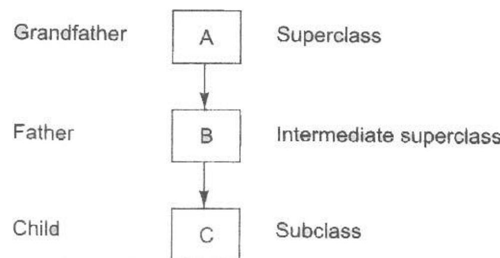

Fig. 6.4 Multilevel inheritance

The class **A** serves as a base class for the derived class **B** which in turn serves as a base class for the derived class **C**. The chain ABC is known as *inheritance path*.

A derived class with multilevel base classes is declared as follows.

```
class A
{
    ............
    ............
}
class B extends A  // First level
{
    ............
    ............
}
class C extends B  // Second level
{
    ............
    ............
}
```

This process may be extended to any number of levels. The class **C** inherit the members of both **A** and **B** as shown in Fig. 6.5

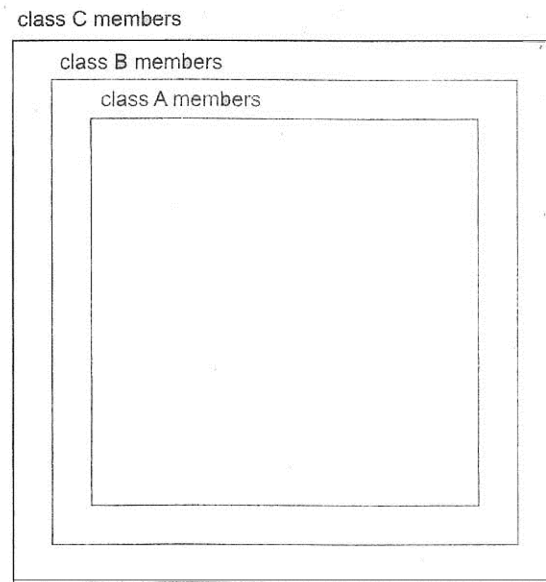

Fig. 6.5 C contains B which contains A

Hierarchical Inheritance

Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below the level. As an example, Fig. 6.6 shows a hierarchical classification of accounts in a commercial bank. This is possible because all the accounts possess certain common features.

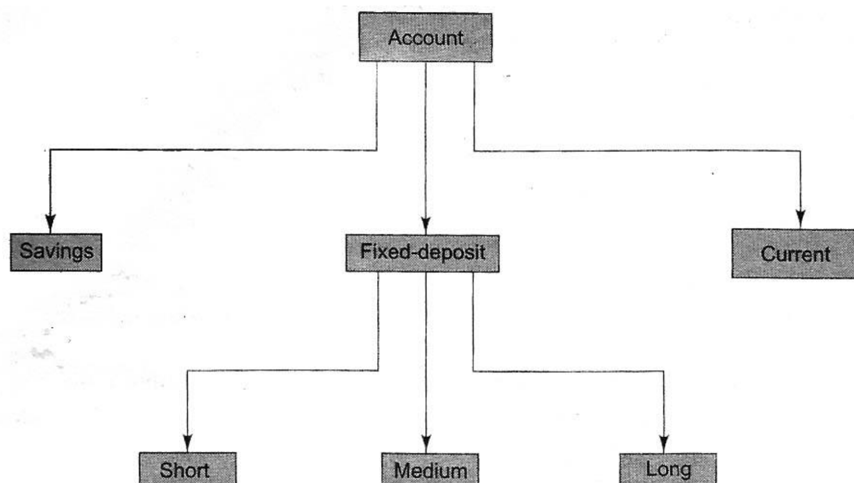

Fig. 6.6 Hierarchical classification of bank accounts

Overriding Methods

We have seen that a method defined in a super class is inherited by its subclass and is used by the objects created by the subclass. Method inheritance enables us to define and use method repeatedly in subclasses without having to define the methods again in subclass.

However, there may be occasions when we want an object to respond to the same method but have different behavior when that method is called. That means, we should override the method defined in the superclass. This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass. Then, when that methods is called, the method defined in the subclass is invoked and executed instead of the one in the superclass. This is known as overriding. Program 6.6 illustrates the concept of overriding. The method **display( )** is overridden.

Program 6.6 Illustration of method overriding

```
class Super
{
      int x;
      super(int x)
      {
            this.x = x;
      }
      void display( )                  // method defned
      {
            System.out.println("Super x = " + x);
      }
}
class Sub extends Super
{
      int y;
      sub (int x, int y)
      {
            Super(x);
            this.y = y;
      }
      void display( )                  // method defined again
      {
            System.out.println("Super x = " + x);
            System.out.println("Sub y = " + y);
      }
}
class OverrideTest
{
      public static void main(String args[ ])
      {
            Sub s1 = new Sub(100,200);
            S1.display( );
      }
}
```

Output of Program 6.6

```
Super x = 100
Sub   y = 200
```

Note that the method display **( )** defined in the subclass is invoked.

Final Variables and Methods

All methods and variables can be overridden by default in subclasses. If we wish to prevent the subclasses from overriding the members of the superclass, we can declare them as final using the keyword **final** as a modifier. Example:

```
final int SIZE = 100;
final void showstatus( ) {........}
```

Making a method final ensure that the functionality defined in this method will never be altered in any way. Similarly, the value of a final variable can never be changed. Final variables, behave like class variables and they do not take any space on individual objects of the class.

Final Classes

Sometimes we may like to prevent a class being further subclasses for security reasons. A class that cannot be subclassed is called a *final class*. This is achieved in Java using the keyword **final** as follows:

```
final class Aclass {..........}
final class Bclass extends someclass {..........}
```

Any attempt to inherit these classes will cause an error and the computer will not allow it.

Declaring a class **final** prevents any unwanted e3xtensions to the class. It also allows the compiler to perform some optimistation when a method of a final class is invoked.

Finalizer Methods

We have seen that a constructor method is used to initialize an object when it is declared. This process is known as *initialization*. Similarly, Java supports a concept called *finalization*, which is just opposite to initialization. We know that Java run-time is an automatic garbage collecting system. It automatically frees up the memory resources used by the objects. But objects may hold other non-object resources such as file descriptors or window system fonts. The garbage collector cannot free these resources. In order to free these resources we must use a *finalizer method*. This is similar to *destructors* in C++.

The finalizer method is simply **finalize( )** and can be added to any class. Java calls that method whenever it is about to reclaim the space for that object. The **finalize** method should explicitly define the tasks to be performed.

Abstract Methods and Classes

We have seen that by making a method ***final*** we ensure that the method is not redefined in a subclass. That is, the method can be sub classed. Java allows us to do something that is exactly opposite to this. That is, we can indicate that a method must always be redefined in a subclass, this making overriding compulsory. This is done using the modifier keyword **abstract** in the method definition. Example:

```
abstract class Shape
{
     ..........
     ..........
     abstract void draw( );
     ..........
     ..........
}
```

When class contains one or more abstract methods, it should be declared **abstract** as shown in the example above.

While using abstract classes, we must satisfy the following conditions:

- We cannot use abstract classes to instantiate objects directly. For example,

```
Shape s = new Shape( )
```

is illegal because **shape** is an abstract class.

- The abstract methods of an abstract class must be defined in its subclass.
- We cannot declare abstract constructors or abstract static methods.

Methods with Varargs

Varargs represents variable length arguments in methods, which is one of the features introduced by J2SE 5.0. It makes that Java code simple and flexible. Varargs takes the following form:

```
<access specifier> <static> void method-name(Object...arguments)
{
}
```

In the above syntax, the method contains an argument called *varargs* in which Object is the type of an argument, ellipsis (…) is the key to varargs and *arguments* in the name of the variable.

Thus, varargs allows us to declare a method with the unspecified number of parameters for a given argument. The varargs must be the final argument in the argument list of a method. Varargs is identified by the type of an argument followed by the ellipsis (…) and the name of a variable.

For example, consider the following declaration of the method, **sample,** which contains the same type of arguments. String is used for more than one arguments.

```
public void sample (String username, String password, String mailid);
```

The above code is an example for simple method declaration. The above method declaration can be replaced by varargs, as shown below:

```
public void sample(String ... var_name);
```

Where **String … var_name** specifies that we can pass any number of String arguments to the sample method. The following declarations invoke the constructor method of the class, which contains the method, sample:

```
public void method_name(String user, String pword);
public void method_name(String user, String pword, String mailid);
or
public void method_name(String user, String pword, String mailid), Stringdesc);
```

It is possible to use the variable length argument as a final argument to other type of constructors. Hence, in the above declaration, the third line can be rewritten as:

```
public void method_name(String user, String ... var_arg);
```

Program 6.7 Illustrates the use of varargs to print the String value passed as an argument to a method

```
class Exampleprg
{
   Exampleprg(String... person)
   {
   for(String name: person)
   {
      System.out.println("Hello " + name);
   }
}
   public static void main(String args[]);
   {
      Exampleprg("John", "David", "Suhel");
   }
}
```

Program 6.7 produces the following output:

```
Hello John
Hello David
Hello Suhel
```

At complete time, String… var_arg is converted to String [ ] var_arg. We can also pass an array of strings to the method, as follows:

```
class Exampleprg
{
   String str1, str2;
   Exampleprg(String[] vargs)
   {
       for(int i=0; i<vargs.length; i++)
       {
             Str1=vargs[i];
             System.out.println("Hello "+str1+".");
       }
   }
   public static void main(String[]args)
   {
       Exampleprg ex=new Exampleprg(args);
   }
}
```

Compile and run the above program as shown below:

```
Javac Exampleprg John David Suhel
```

This code yields the same output as the above program.

Note: Varargs does not generate any compile time errors even if an empty argument is passed as a parameter to a method.

Visibility Control

We stated earlier that it is possible to inherit all the members of a class by a subclass using the keyword **extends**. We have also seen that the variables and methods of a class are visible everywhere in the program. However, it may be necessary in some situations to restrict the access to certain variables and methods from outside the class. For example, we may not like the objects of a class directly alter the value of a variable or access a method. We can achieve this in Java by applying *visibility modifiers* to the instance variables and methods. The visibility modifiers are also known as *access modifiers*. Java provides three types of visibility modifiers: **public, private** and **protected.** They provide different levels of protection as described below.

public Access

Any variable or method is visible to the entire class in which it is defined. What if we want to make it visible to all the classes outside this class? This is possible by simply declaring the variable or method as **public**. Example:

```
public int number;
public void sum( ) {...........}
```

A variable or method declared as **public** has the widest possible visibility and accessible everywhere. In fact, this is what we would like to prevent in many programs. This takes us to the next levels of protection.

friendly Access

In many of our previous examples, we have not used **public** modifier, yet they were still accessible in other classes in the program. When no access modifier is specified, the member defaults to a limited version of public accessibility known as "friendly" level of access.

The difference between the "public" access and the "friendly" access is that the **public** modifier makes fields visible in all classes, regardless of their packages while the friendly access makes fields visible only in the same package, but not in other packages. (A package is a group of related classes stored separately. They are explored in detail in Chapter 9). A package in Java is similar to a source file in C.

Protected Access

The visibility level of a "protected" field lies in between the public access and friendly access. That is, the **protected** modifier makes the fields visible not only to all classes and subclasses in the same package but also the subclasses in other packages. Note that non-subclasses in other packages cannot access the "protected" members.

private Access

private fields enjoy the highest degree of protection. They are accessible only with their own class. They cannot be inherited by subclasses and therefore not accessible in subclasses. A method declared as **private** behaves like a method declared as **final**. It prevents the method from being subclassed. Also note that we cannot override a non-private method in a subclass and then make it private.

Private protected Access

A field can be declared with two keywords **private** and **protected** together like:

```
private protected int condeNumber;
```

This gives a visibility level in between the "protected" access and "private" access. This modifier makes the fields visible in all subclasses regardless of what package they are in. Remember, these fields are not accessible by other classes in the same package. Table 8.1 summarises the visibility provided by various access modifiers.

Table 6.1 Visibility of Field in a Class

Access location ↓ / Access modifier →	*public*	*protected*	*friendly (default)*	*private protected*	*private*
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

Rules of Thumb

The details discussed so far about field visibility may be quite confusing and seem complicated. Given below are some simple rules for applying appropriate access modifies.

1. Use **public** if the field is to be visible everywhere.
2. Use **protected** if the field is to be visible everywhere in the current package and also subclasses in other packages.
3. Use "default" if the field is to be visible everywhere in the current package only.
4. Use **private protected** if the field is to be visible only in subclasses, regardless of packages.
5. Use **private** if the field in ***not*** to be visible anywhere except in its own class.