Chapter 2 – Classes, Objects and Methods

Marks Allotted: 16
Lectures: 08

Semester Pattern:

Exam:	Winter 2008	Summer 2009
Marks:	24	28

Yearly Pattern (JPR-IF 1526):

Exam:	S'04	W'04	S'05	W'05	S'06	W'06	S'07	W'08
Marks:	40	28	48	48	40	44	36	40

Syllabus Contents:

2.1 Defining a class, Creating object, Accessing class members, Constructor, Methods Overloading, Static Member
2.2 Inheritance Extending a Class (Defining a subclass Constructor, Multilevel inheritance, Hierarchical inheritance, Overriding Methods, Final variable and Methods, Final Classes, Abstract method and Classes
2.3 Visibility Control
Public access, friend access, Protected access, Private access, Private Protected access
2.4 Array, Strings and Vectors
Arrays, One Dimensional array, Creating an array, Two Dimensional array, Strings, Vectors, Wrapper Classes

Introduction

Java is true object-oriented programming language and therefore everything in Java is a concept of the class. Therefore, class is at the core of Java. The entire Java language is built upon this logical construct. It defines the shape and nature of the object. The classes form basis for object-oriented programming in Java

Classes create objects and use methods to communicate between them. That is all about object-oriented programming. Classes provide convenient method for packing together a logically related data items and functions that work on them. In Java, the data items are called as fields and functions are called as 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.

Class definition

A class is a user-defined data type with the template that serves to define its properties. Most important thing here is to understand is that a class defines new data type. Once defined, this new data type can be used to create objects of that type. Thus, class is a template for an object and object is the instance of the class. Object can be called as variable of type class. When we define a class, we declare its exact form and nature. Here we specify the data that it contains and the code that operates on data. Very simple classes may contain only the code or only data, most real world classes contain both.

A class is declared by the keyword **class**. Class definition form is given below.

```
class class-name
{
      data-type instance-variable1;
      data-type instance-variable2;
      - - - - -;
      data-type instance-variableN;
      data-type method-name1(parameter-list)
      {
           //body of the method1
      }
      data-type method-name2(parameter-list)
      {
           //body of the method2
      }
      - - - - -;
      data-type method-nameN(parameter-list)
      {
           //body of the methodN
```

```
        }
}
```

The data or variables defined within a class are called as instance variables because each instance of the class (that is, each object) contains its own separate copy of variables. The code is contained within the methods. But remember any of the instance variables and methods is optional to declare. If we are familiar with C++ programming language we have to remember that the closing curly bracket does not contain any semicolon at end. Collectively, the code inside the class is called as members of the class. In the most classes, the instance variables acted upon and accessed by the methods defined for that class.

The class name, instance variables and method names are valid Java identifiers names. In general we have to follow the naming conventions for giving names to class and its instance variables. All methods have same general form as main ( ). However, most methods will not be specified as static or public.

Adding variables to a class

The data in a class is in the form of instance variables. We can declare the instance variables exactly the same way as we declare the local variables. For example:

```
class Country
{
        long population;
        float currencyRate;
        int noOfStates;
}
```

Here the class name is ‘Country’. It contains three different instance variables named population (data type: long), currancyRate(data type: float) and noOfStates(data type: int). As we have not created the object of the class, the storage space in the memory is not allocated for it. It will be allocated after creation of object.

Creating objects

Creating the objects of the class is also called as instantiating an object. As stated earlier, a class creates new data type. In the above case of example ‘Country’ is new data type.

Objects are created using the ‘new’ operator. This ‘new’ operator creates an object of the specified class and returns the reference of that object. See the example for creating the object of the above class.

```
Country japan;                  // declaration
japan = new Country();          // instantiation
```

The first statement declares a variable to hold the object reference. Second statement actually assigns the object reference to the variable. The variable name 'japan' is called as the object of class 'Country'.
Both statements can also be combined into one as shown below:

```
Country japan = new Country();
```

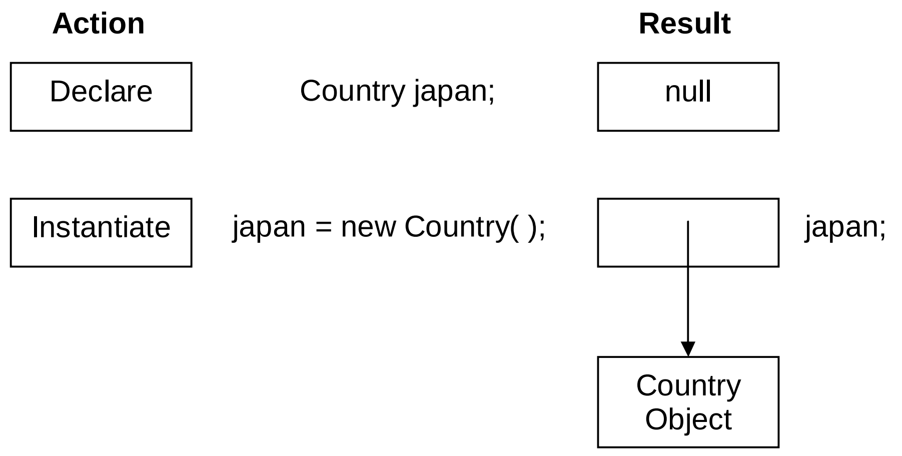
Here, Country ( ) specifies the default constructor of the class. We can create any number of objects of the class 'Country' as,

```
Country nepal = new Country();
Country bhutan = new Country();
Country myanmar = new Country();
```

After the objects are created, there is a separate copy of the instance variables is created for each of them. So, separate memory is allocated to them.
The new operator dynamically allocates the memory for the object. That is, it allocates the memory at run time. It has the general form:

```
class-var = new classname();
```

Here, the 'class-var' is a variable of the class type being created. The 'classname' is name of the class that is being instantiated. Figure will explain how the objects are actually created using 'new'.



declaring and creating object of class 'Country'

Accessing members using object

If we are familiar with C++, the same way is used in both of the languages to access data members of the class. As seen in the first chapter the dot operator or member selection operator is used to access the instance variables of the class. We can not access the instance variables outside of the class without referring to the object. Member access follows the following syntax:

```
objectname.variablename
```

Here, the ‘objectname’ is the name of the object and ‘variablename’ is name of the instance variable. So, the instance variables of the class ‘Country’ can be accessed as,

```
japan.population = 58612453;
bhutan.currancyRate = 0.236;
japan.noOfSates = 5;
```

The first statement tells compiler to assign the copy of population that is contained within object ‘japan’ with value of 58612453. Other statements also follow the same.

Observe the following program which inputs the values in the class and displays them.

```
//Accessing the member variables.
  import java.util.Scanner;
  class Country
  {
      long population;
      int noOfStates;
      float currancyRate;
  }
  class MyClass
  {
      public static void main(String args[])
      {
          Country nepal = new Country();
          Scanner in = new Scanner(System.in);

          System.out.print("Enter population of nepal: ");
          nepal.population = in.nextLong();

          System.out.print("Enter currency rate of nepal: ");
          nepal.currancyRate = in.nextFloat();

          System.out.print("Enter no of states of nepal: ");
          nepal.noOfStates = in.nextInt();

          System.out.println("\n* Entered Information *");
          System.out.print("Population of nepal is : ");
          System.out.println(nepal.population);
          System.out.print("Currency rate of nepal is : ");
          System.out.println(nepal.currancyRate);
          System.out.print("No of states of nepal is : ");
          System.out.println(nepal.noOfStates);
      }
  }
```

Accessing the member variables of class.

Output:

```
Enter population of nepal: 856223
Enter currency rate of nepal: 56.23
Enter no of states of nepal: 12

* Entered Information *
Population of nepal is : 856223
Currency rate of nepal is : 56.23
No of states of nepal is : 12
```

Observe another practical example that creates a class for the circles and calculates the specifications related to that.

```
//Circle specifications
  import java.util.Scanner;
  class Circle
  {
      int radius;
      float perimeter;
      float area;
  }
  class MyCircle
  {
      public static void main(String args[])
      {
         final float pi = 3.14f;
         Circle c = new Circle();
         Scanner in = new Scanner(System.in);

         System.out.print("Enter radius : ");
         c.radius = in.nextInt();
         c.perimeter = 2.0f * pi * (float) c.radius;
         c.area = pi * (float) (c.radius * c.radius);

         System.out.println("Perimeter : "+c.perimeter);
         System.out.println("Area : "+c.area);
      }
  }
```

Program to calculate the specifications of the circle

Output:

```
Enter radius : 23
Perimeter : 144.44
```

```
Area : 1661.06
```

Adding methods

As stated earlier, the class contains two parts i.e. instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility.

The objects created by a class having only variables can not communicate effectively with external world. We must add therefore methods that are necessary for manipulating the data contained in the class. Methods are declared in the body of the class immediately after the declaration of instance variables. We can call method as a self contained block of statements that are used to perform certain task. The general form of the declaration of the method is,

```
data-type method-name (parameter-list)
{
     Body of the method;
}
```

The method declaration contains four different parts:

1 Name of the method (method-name)
2 Data type of the values returned by method (data-type)
3 The list of parameters (parameter-list)
4 Body of the method

The 'data-type' specifies the type of data returned by method. This can be any valid data type including class type that we create. It the method does not return any value, its return type must be **void**. The 'method-name' must be a valid identifier's name. Its name must not be used anywhere in the current scope. The parameter-list is always enclosed in the pair of parenthesis. This list contains variable names and type of all the values that we want to give method as input. The variables in the list must be separated by commas. Parameters can also be called as arguments of the method. If method has no parameters, then the parameter list may be empty.

Body of the method actually defines the sequence operations to be done on the data supplied to method. The class 'Country' will look like after adding method as shown below:

```
class Country
{
     long population;
     int noOfStates;
     float currancyRate;

     void input(long x, int y)
     {
          population = x;
```

```
            noOfStates = y;
        }
    }
```

Here, the method 'input' is added in the class. Its return type is void and contains two different parameters i.e. x and y having data type long and int respectively. In order to access the member method of the class, we use same dot operator (i.e. member selection operator) in between object name and method name as,

```
objectname.methodname(parameters-list);
```

Let us rewrite the program by adding methods to it.

```
//Circle specifications using methods
  import java.util.Scanner;
  class Circle
  {
      int radius;
      float perimeter;
      float area;

      void input()
      {
         Scanner in = new Scanner(System.in);

         System.out.print("Enter radius : ");
         radius = in.nextInt();
      }
      void calculate()
      {
         final float pi = 3.14f;
         perimeter = 2.0f * pi * (float) radius;
         area = pi * (float) (radius * radius);

         System.out.println("Perimeter : "+ perimeter);
         System.out.println("Area : "+ area);
      }
  }
  class OurCircle
  {
      public static void main(String args[])
      {
         Circle c = new Circle();
         c.input();
         c.calculate();
      }
  }
```

Program by adding methods

Output:

```
Enter radius : 23
Perimeter : 144.44
Area : 1661.06
```

Here both the declared methods are **void**. That is, they are not returning any value. A method can return a value. The return statement must be written at the end of the method. General form of writing a return statement is,

```
return (value);
return (expression);
```

In the second form the expression is evaluated and its value is returned in the calling method. It is necessary for interclass communication.

Following method returns the float value.

```
float area(int radius)
{
     final float pi = 3.14f;
     float val;
     val = pi * (float) (radius * radius);
     return(val);
}
```

Observe another simple program that uses two different methods for finding the maximum and minimum number between two integers.

```
//Find min and max number using methods
  import java.util.Scanner;
  class Number
  {
     int max(int x,int y)
     {
        if(x>y)
           return(x);
        else
           return(y);
     }
     int min(int x,int y)
     {
        if(x<y)
           return(x);
        else
           return(y);
     }
  }
  class MaxMin
  {
      public static void main(String args[])
      {
         int a, b;
         int mx, mn;
         Number n = new Number();
         Scanner in = new Scanner(System.in);
         System.out.print("Enter first number : ");
         a = in.nextInt();

         System.out.print("Enter second number : ");
         b = in.nextInt();

         mx = n.max(a,b);
         System.out.print("\nMaximum : "+mx);
         mn = n.min(a,b);
         System.out.print("\nMinimum : "+mn);

      }
  }
```

Find maximum and minimum by adding methods

Output:

```
Enter first number : 56
Enter second number : 84

Maximum : 84
Minimum : 56
```

The methods ‘max’ and ‘min’ both contains two return statements. But in any case only one of them is executed at a time as per the condition. The variables x and y has local block scope within the method, where they are declared. So, multiple declarations in different blocks are possible. Variables ‘a’ and ‘b’ are called actual parameters and ‘x’ and ‘y’ are called as formal parameters of the method. In a method call, the actual parameter’ values are substituted with formal parameter’ values in sequence. Remember, the value given in the return statement and the data type specified for the method must be matched.

Constructor

It can be boring to initialize all of the variables in a class each time an instance is created. Even when we add convenience functions like input( ), it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and it is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object instantly.

Let's examine the constructor included for the class 'Country'.

```
class Country
{
     long population;
     int noOfStates;
     float currancyRate;

     Country(long x, int y)
     {
          population = x;
          noOfStates = y;
     }
     void display()
     {
          System.out.println("Population:"+population);
          System.out.println("No of states:"+noOfStates);
     }
}
```

The constructor given here is having two parameters i.e. long x and int y. When we create the object of class, the constructor can be called like,

```
Country japan = new Country(4582452, 15);
```

The instance variables, 'population' and 'noOfStates' for object 'japan' will be initialized with the values given in the creation of object. So there is no need to initialize these variables for particular object. If we print the value of 'japan.population' it will print 4582452. The constructor which contains parameters is called as parameterized constructor else it is a default constructor. A default constructor is called upon the creation of object.

Constructor overloading

The concept of overloading refers to the use of same name for different purposes. In simple language, a class can have more than one constructor but with different parameters. This is known as constructor overloading. Program 2.5 illustrates this concept.

```
// Overloaded constructors
```

```
class Box
{
    int height;
    int depth;
    int length;

    Box()
    {
       height = depth = length = 10;
    }
    Box(int x,int y)
    {
       height = x;
       depth = y;
    }
    Box(int x, int y, int z)
    {
       height = x;
       depth = y;
       length = z;
    }
}
class BoxClass
{
    public static void main(String args[])
    {
        Box a = new Box();            //statement1
        System.out.println("depth of a : "+a.depth);

        Box b = new Box(12,23);       //statement2
        System.out.println("depth of b : "+b.depth);

        Box c = new Box(99,84,36);    //statement3
        System.out.println("depth of c : "+c.depth);
    }
}
```

Illustration of constructor overloading

Output:

```
depth of a : 10
depth of b : 23
depth of c : 84
```

In program 2.5, the class Box contains three different variables i.e. height, depth and length. It also has three different constructors. First of them is default constructor and another two are parameterized constructers. Second constructor contains two parameters and third contains three. When the object 'a' is created in statement1, default constructor is called in which all the instance variables are initialized to value 10. Thus, a.depth is outputted as 10. Second constructor only initializes two instance variables i.e. height and depth with the given values in sequence. Third constructor initializes all the three instance variables upon creation of object. Remember we can call any constructor for any object in the program. A class can contain any number of constructors but they must differ in type and number of parameters.

If we declare two constructors with same number and type of parameters the compiler will flash the error that 'constructor is already defined in the class'.

The 'this' keyword

Many times it is necessary to refer to its own object in a method or a constructor. To allow this Java defines the 'this' keyword. If we are familiar with C++, it is not hard to learn and understand the concept of 'this' keyword. The 'this' is used inside the method or constructor to refer its own object. That is, 'this' is always a reference to the object of the current class' type.

Observe the following code,

```
class Box
{
  int height;
  int depth;
  int length;

  Box(int height, int depth, int length)
  {
     this.height = height;
     this.depth = depth;
     this.length = length;
  }
}
```

The class 'Box' contains three instance variables i.e. height, depth and length. The constructor of the class also contains three different local variables with the same names of instance variables. Compiler will not show any error here. Then how differentiate among these variables? The 'this' keyword does this job. 'this' will refer to the variables of its own class. It is acting as the object of the current class.

In method also the 'this' keyword is used to differentiate between instance and local variables. This concept is also referred as 'Instance variable hiding'. This resolves the name-space collisions.

One of the most important application of the 'this' keyword is that it can be used to call one constructor from another constructor. Program 2.6 illustrates this concept.

```
//Uses of this keyword
  class Planet
```

```
{
    long distance;
    long satelite;
    int oneDay;
    Planet()              //constructor1
    {
       satelite = 0;
       oneDay = 24;
    }
    Planet(long x)        //constructor2
    {
       this();
       distance = x;
    }
    Planet(long x, long y) //constructor3
    {
       this(x);
       satelite = y;
    }
}
class UseOfThis
{
    public static void main(String args[])
    {
       Planet p = new Planet(452L, 96L);
       System.out.println("Distance: "+p.distance);
       System.out.println("Satelite: "+p.satelite);
       System.out.println("One day : "+p.oneDay);
    }
}
```

Uses of the ‘this’ keyword

Output:

```
Distance: 452
Satelite: 96
One day : 24
```

In program, the class ‘Planet’ contains three different constructors. The constructor3 makes a call to constructor2 by passing one parameter to ‘this’ keyword. Again constructor2 makes a call to the default constructor, constructor1. This chain of constructors has called each constructor in a single statement.

The ‘this’ can also be used to call the method of its own class by following the syntax:

```
this.methodname();
```

Methods overloading

In Java it is possible to define two or more methods within the same class that are having the same name, but their parameter declarations are different. In the case, the methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways of Java implementation of polymorphism. This concept of method overloading is just same as function overloading of C++. If we have never used a language that allows the overloading of methods, then the concept may seem strange at first. But, method overloading is one of Java's most exciting and useful features.

When an overloaded method is called, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Program illustrates the concept of method overloading.

```
//Method overloading
  class Maths
  {
      int cal;
      float val;
      int add(int x, int y)        //version1
      {
          cal = x + y;
          return(cal);
      }
      int add(int z)               //version2
      {
         cal = z + 10;
         return(cal);
      }
      float add(float x, float y)  //version3
      {
         val = x + y;
         return(val);
      }
  }
  class Overloading
  {
      public static void main(String args[])
      {
          Maths m = new Maths();
          System.out.println("52 + 41 = "+m.add(52,41));
          System.out.println("52 + 10 = "+m.add(52));
          System.out.println("1.2 + 9.2 = "+m.add(1.2f,9.2f));
      }
  }
```

Methods overloading

Output:

```
52 + 41 = 93
52 + 10 = 62
1.2 + 9.2 = 10.4
```

Program contains three different versions of the method ‘add’ in the class ‘Maths’. All of they contain different type and number of parameters. According to the number and type used in the method call using object, the appropriate version of the method is called. A class can contain any number of versions of a single method.

Observe and analyze one more application oriented program which uses the concept of methods overloading.

```
//Method overloading
  import java.util.Scanner;
  class Area
  {
      int area(int side)                //area of a square
      {
          return(side * side);
      }
      float area(float radius)          //area of circle
      {
          return(3.14f * radius * radius);
      }
      int area(int len, int wid)        //area of rectangle
      {
          return(len * wid);
      }
  }
  class Shape
  {
      public static void main(String args[])
      {
         Area s = new Area();
         int x,y;
         float z;
         Scanner in = new Scanner(System.in);
         System.out.print("Enter square side : ");
         x = in.nextInt();
         System.out.println("Area of square : "+s.area(x));

         System.out.print("Enter length & width : ");
         x = in.nextInt();
         y = in.nextInt();
```

```
            System.out.println("Area of rect: "+s.area(x,y));

            System.out.print("Enter radius : ");
            z = in.nextFloat();
            System.out.println("Area of circle : "+s.area(z));
        }
  }
```

Method overloading to calculate area of various shapes

Output:

```
Enter square side : 12
Area of square : 144
Enter length & width : 8 9
Area of rect: 72
Enter radius : 23.10
Area of circle : 1675.5355
```

Static Members

Methods and variables defined inside the class are called as instance methods and instance variables. That is, a separate copy of them is created upon creation of each new object. But in some cases it is necessary 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 created by preceding them with the keyword static.

For example:

```
static int cal;
static float min = 1;
static void display(int x)
```

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. We can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist. Instance variables declared as static are, essentially, 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 members.

Since, these members are associated with the class itself rather than individual objects, the static variables and static methods are often referred as class variables and class methods in order to distinguish them from instance variables and instance methods.

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. (The keyword super relates to inheritance and is described in the next topic).

Once we declared a method or variables as static, we don't need to create the object of the class in order to access it. Just use the dot operator in between the name of the class and static member. Java class library has several set of static variables and methods. They will be discussed in the upcoming chapters. The concept of static members has added the power and flexibility to the Java language.

Following example demonstrates the use of static members.

```
//Demonstration of static members
  class MyWork
  {
     static int x = 10;
     static int count = 1;
     static void display()
     {
        System.out.println("Static has initialized...");
     }
     static void increment()
     {
        System.out.println("Function call : "+count);
        count++;
     }
  }
  class StaticMember
  {
     public static void main(String args[])
     {
        MyWork.display();                    //statement1
        System.out.print("Value of x: ");
        System.out.println(MyWork.x);        //statement2
        MyWork.increment();                  //statement3
        MyWork.increment();                  //statement4
        MyWork.increment();                  //statement5
     }
 }
```

Program using static variables and methods

Output:

```
Static has initialized...
Value of x: 10
Function call : 1
Function call : 2
Function call : 3
```

In program, two static variables and two static methods are defined. We have called these methods outside of the class by using class name directly (observe statement1). In statement2, the static variable is also accessed by referring only the name of class. The value of second static variable 'count' is incremented in another static method 'increment'. This method is also called three times and for all three times we got the different output. Because the static members are initialized once so there value remains as it until program does not come to end.

Argument passing to methods

In general, there are two ways that a programming language can pass an argument to a function. The first way is call-by-value. This method copies the value of an argument into the formal parameter of the function. Therefore, changes made to the parameter of the function have no effect on the argument. The second way an argument can be passed is call-by-reference. In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the function, 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 function. Java uses both approaches, depending upon what is passed.

In Java, when you pass a simple type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

For example:

```
// Call by value.
  class Test
  {
     void meth(int i, int j)
     {
        i++;
        j++;
     }
  }
  class CallByValue
  {
      public static void main(String args[])
      {
          Test ob = new Test();
          int a = 22, b = 93;
          System.out.print("a and b before call: ");
          System.out.println(a + " " + b);

          ob.meth(a, b);

          System.out.print("a and b after call: ");
          System.out.println(a + " " + b);
      }
  }
```

Example of Call by Value

Output:

```
a and b before call: 22 93
a and b after call: 22 93
```

When we pass an object to a method, it is called as passed by reference or the method is called by reference. Keep in mind that when we create a variable of a class type, we are only creating a reference to an object. Thus, when we 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. For example, consider the following program:

```
// Objects are passed by reference.
  class Test
  {
      int a, b;

      void meth(Test o)
      {
         o.a++;
         o.b++;
      }
  }
  class CallByRef
  {
      public static void main(String args[])
      {
          Test ob = new Test();

          ob.a = 22;
          ob.b = 93;
          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);
      }
  }
```

Example of Call by Reference

Output:

```
ob.a and ob.b before call: 22 93
ob.a and ob.b after call: 23 94
```

By observing the output, we can say that the changes are made onto the arguments. We have passed the object as the parameter, changed the values of the variables that has affected the actual parameters also. Though like C++, Java does not support the pointers, it supports the concept of call by reference.

Remember, when a simple data type is passed to a method, it is done by use of call-by-value. Objects are always passed by use of call-by-reference.

Methods returning objects

A method is able to return any type of the data. It can return the objects also. But in the method call the returning value of the method must be assigned to the object of that type.

```
//A method returning object
  class Square
  {
       int n;
       Square(int x)
       {
          n = x;
       }
       Square change()
       {
          Square temp = new Square(n);
          temp.n = temp.n * temp.n;
          return(temp);
       }
   }
   class ReturnObject
   {
       public static void main(String args[])
       {
           Square s = new Square(8);
           Square t;
           t = s.change();
           System.out.println("Square of 8 is: "+t.n);
       }
   }
```

Method which returns object

Output:

```
Square of 8 is: 64
```

In program, a method named ‘change( )’ is defined which return the object of its own class. When this method is called in the main method its return values is assigned to the object of the same class type.

After this assignment, the returned object's members are assigned as it is to the object named, 't'.

Nested and inner classes

It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B is known to A, but not outside of A. A nested class has access to all the members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

There are two types of nested classes: static and non-static. A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are rarely used. The most important type of nested class is the inner class.

An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class.

Observe the example,

```
// Demonstration of an inner class.
  class Outer            //Outer class
  {
      int out_x;
      Outer(int x)
      {
         out_x = x;
      }
      void test()
      {
          Inner inner = new Inner();
          inner.display();
      }

      class Inner        //Inner class
      {
          void display()
          {
               System.out.print("Value: out_x =");
               System.out.println(out_x);
          }
      }
  }
  class InnerClass
  {
       public static void main(String args[])
```

```
        {
            Outer outer = new Outer(15);
            outer.test();
        }
}
```

Inner and Outer class demo

Output:

```
Value: out_x =15
```

In program, two classes are defined named, ‘Inner’ and ‘Outer’. ‘Inner’ class is defined inside the ‘Outer’ class. So we can create the object of the ‘Inner’ class only inside the ‘Outer’ class. That is, ‘Inner’ class is bounded by the scope of the ‘Outer’ class. If we create the object of ‘Inner’ class in ‘InnerClass’ then the compiler will flash the error. Because, ‘Inner’ class can not be accessed in ‘InnerClass’.

We have created object of ‘Outer’ class. Then by using it we called the method ‘test ( )’ of it. Inside the ‘test ( )’ method, the object of ‘Inner’ class is created. That object is used to call the method ‘display ( )’ of the ‘Inner’ class. Inner class can access all the members of the outer class.

Remember, any number of classes can be nested inside them. As class creates a new data type, it can also be created inside the method. Program 2.15 shows demonstration of creation of a class inside the method.

```
// Demonstrate an inner class inside method.
class MethodClass
{
        public static void main(String args[])
        {
            class MyClass     //A class inside main() method
            {
                int x;
                MyClass(int x) //Constructor
                {
                    this.x = x;
                }
                void display() //Method
                {
                    System.out.print("Your value : ");
                    System.out.println(x);
                }
            }

            MyClass m = new MyClass(63);  //Object
            m.display();                  //Method call
        }
}
```

Defining a class inside a method

Output:

```
Your value : 63
```

In program 2.16, the class ‘MyClass’ is defined inside the main ( ) method. So we can create its object inside the main only. It is bounded by the scope of main ( ).

Inheritance

Reusability is one of the most useful advantages of the Object Oriented Programming. Reusability means using something again and again without creating a new one. Java programming language supports this concept. The reusability can be obtained by creating new classes and reusing the properties of the existing one. The mechanism of deriving a new class from existing one is called as inheritance. The old class or existing class is known as super class and new class is known as subclass. Thus, a subclass is specialized version of super class. We can call super class as base class or parent class and subclass as derived class or child class.

The subclass derives all of the instance variables and methods defined by super class and add its own unique elements. So the elements defined in super class are reused in the subclass.

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. There are four different types of the inheritance which are allowed by Java.

1. Single Inheritance (only one super and subclass)
2. Multiple Inheritance (several super classes for a subclass)
3. Hierarchical Inheritance (one super class with many subclasses)
4. Multilevel Inheritance (subclass is super class of another class)
5. Hybrid Inheritance (combination of above three types)

These forms are shown in fig.2.1.

Java does not directly implement the multiple inheritance. This is implemented using the concept of interface. This is discussed in the next chapter.

Creating the inheritance

The class can derived from another class by following the syntax:

```
class subclassname extends superclassname
{
     //Body of the class;
}
```

The keyword ‘extends’ specifies that the properties of superclassname are extended to subclassname. After this the subclass will contain all the methods of super class and it will add the members of its own. Remember, we can not extend a subclass from more than one super class.

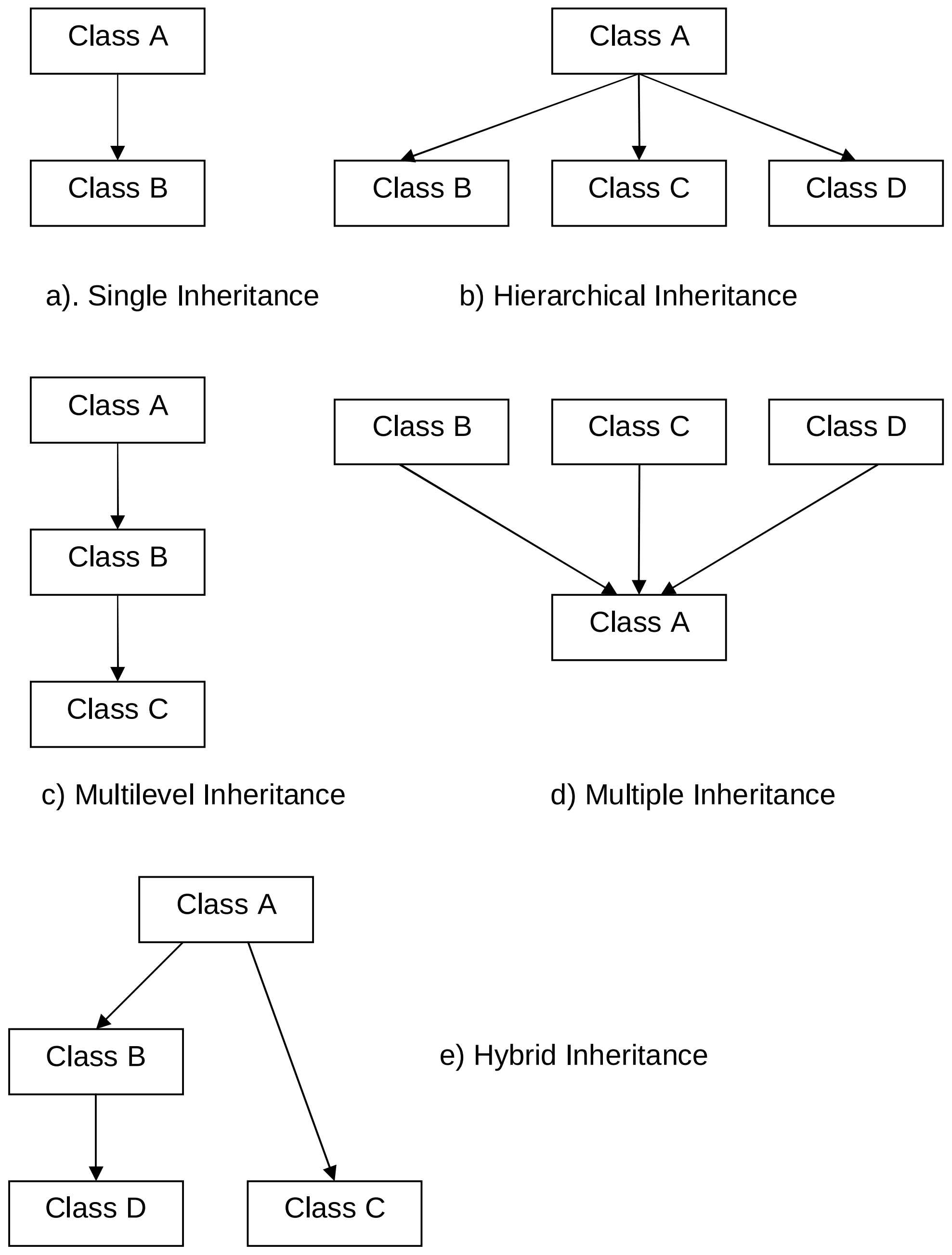


Fig. 2.2 Types of Inheritance

Program illustrates the concept of inheritance.

```
// A simple example of inheritance.

  class Super
  {
     int i, j;
     void show()
     {
        System.out.print("i and j: ");
        System.out.println( + i + " " + j);
     }
  }

  class Sub extends Super
  {
     int k;

     void display()
     {
        System.out.println("k: " + k);
     }
     void sum()
     {
        System.out.println("i+j+k: " + (i+j+k));
     }
  }

  class SimpleInheritance
  {
      public static void main(String args[])
      {
           Super a = new Super();
           Sub b = new Sub();
           a.i = 5;
           a.j = 12;
           System.out.println("Contents of super: ");
           a.show();
           System.out.println();

           b.i = 11;
           b.j = 13;
           b.k = 17;
           System.out.println("Contents of sub: ");
           b.show();
           b.display();
           System.out.println();

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

Program Example of simple inheritance

Output:

```
Contents of super:
i and j: 5 12

Contents of sub:
i and j: 11 13
k: 17

Sum of i, j and k in sub:
i+j+k: 41
```

Program shows an example of simple single inheritance in which the ‘Sub’ class is inherited from ‘Super’ class. So the members of ‘Super’ class that is, ‘i', ‘j’ and method ‘show ( )’ are available in ‘Sub’ class. By creating the object of ‘Sub’ class we can access all these members.

But the methods ‘display ( )’, ‘sum ( )’ and variable ‘k’ can not accessed in ‘Super’ class. So ‘Super’ class object can not use these members.

```
// An example of Single inheritance
   import java.util.Scanner;
   class First
   {
       int val;
       void init()
       {
           System.out.print("Enter a number: ");
           Scanner in = new Scanner(System.in);
           val = in.nextInt();
       }
       int square()
       {
           return(val*val);
       }
   }
   class Second extends First
   {
       int mem;
       int cube()
       {
          mem = square() * val;
          return mem;
       }
   }
   class SingleInheritance
   {
       public static void main(String args[])
       {
          Second s = new Second();
          s.init();
          System.out.println("Cube : "+s.cube());
       }
   }
```

Example of single inheritance

Output:

```
Enter a number: 5
Cube : 125
```

Program 2.18 gives more practical example of the inheritance. Here, ‘Second’ class is inherited from ‘First’ class. In ‘Second’ class, the member method ‘Square( )’ has been called in an expression. After creating the object of the ‘Second’ class we called the method ‘init ( )’ of the ‘First’ class. So remember in the derived class also we can call the method of super class and by creating object of subclass also it is possible to call method from super class.

Super class variable can refer to the subclass object

A reference variable of super class can be assigned a reference to any subclass derived from that super class. In many situations, we find this concept very useful. Program 2.17 has been rewritten by applying this concept in program 2.19.

```
// Super class variable to subclass object
  class Super
  {
     int i, j;
     void show()
     {
        System.out.print("i and j: ");
        System.out.println( + i + " " + j);
     }
  }

  class Sub extends Super
  {
     int k;

     void display()
     {
        System.out.println("k: " + k);
     }
     void sum()
     {
        System.out.println("i+j+k: " + (i+j+k));
     }
  }
  class Reference
  {
      public static void main(String args[])
      {
           Super a;           //Statement1
           Sub b = new Sub();

           b.i = 11;
           b.j = 13;
           b.k = 17;
           System.out.println("Contents of sub: ");
           b.show();
           b.display();
           System.out.println();

           a = b;             //Statement2

           System.out.println("Contents of super: ");
           a.show();
           System.out.println();

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

Super class variable can refer to the subclass object

Output:

```
Contents of sub:
i and j: 11 13
k: 17

Contents of super:
i and j: 11 13

Sum of i, j and k in sub:
i+j+k: 41
```

In program, observe statement1. A reference of the ‘Super’ class has been created. In the statement2, this reference has been assigned to the object of subclass. Here the data that the sub class object is having will get copied into super class object; except its own unique data. This is also known as object slicing. After this assignment, both objects’ common values will be same. This concept has several practical applications. We will see these in upcoming topics.

The 'super' keyword

The keyword 'super' has specially been added for inheritance. Whenever the subclass needs to refer to its immediate super class, 'super' keyword is used. There are two different applications of the 'super'. That are,

- Accessing members from super class that has been hidden by members of the subclass
- Calling super class constructor

Accessing members from super class

In order to access the members from super class following form of the 'super' is used.

```
super.variablename;
```

It is necessary when the name given to the variables in subclass and super class are same. In such cases by creating the object of sub class we can only access its own variable. It can't access the variable from super class. The keyword 'super' is essential in such situations.

//Use of super to access super class members

```
class Primary
{
    int cal;                //declaration1
    void show()
    {
      System.out.println("Super class cal : "+cal);
    }
}
class Secondary extends Primary
{
    int cal;                //declaration2
    Secondary(int x,int y)  //statement1
    {
       cal = x;             //statement2
       super.cal = y;       //statement3
    }
    void display()
    {
      System.out.println("Sub class cal : "+cal);
    }
}
class SuperUse1
{
    public static void main(String args[])
    {
        Secondary s = new Secondary(15,22);
        s.show();
        s.display();
    }
}
```

First use of the 'super' keyword

Output:

```
Super class cal : 22
Sub class cal : 15
```

Program 2.20 illustrates the first use of 'super' keyword. The class 'Secondary' is inherited from class 'Primary'. Both of the classes declared with same variable name with same data type i.e. int cal.

Statement1 is the constructor of the sub class 'Secondary'. When this constructor is called, statement2 and statement3 are executed. Statement2 will assign value of first parameter of constructor to variable 'cal' of class 'Secondary' and statement3 will assign value of second parameter of constructor to variable 'cal' of class 'Primary'. Because of the keyword 'super' is used along variable name, it will refer to the instance variable of super class always. The method 'show ( )' will print value of 'Primary' class 'cal' and 'display ( )' will print value of 'Secondary' class 'cal'.

Calling super class constructor

A subclass can call the constructor defined by super class by using the following form of the 'super' keyword.

```
super(argumentlist);
```

The argument list specifies any arguments needed by constructor in the super class. Remember 'super ( )' must always be the first statement executed inside a subclass' constructor.

```
//Use of super to access super class constructor
  class Primary
  {
      int cal;                 //declaration1
      Primary(int a)
      {
         cal = a;
      }
      void show()
      {
         System.out.println("Super class cal : "+cal);
      }
  }
  class Secondary extends Primary
  {
      int cal;                    //declaration2
      Secondary(int x,int y)    //statement1
      {
         super(y);              //statement2
         cal = x;               //statement3
      }
      void display()
      {
        System.out.println("Sub class cal : "+cal);
      }
  }
  class SuperUse2
  {
      public static void main(String args[])
      {
          Secondary s = new Secondary(15,22);
          s.show();
          s.display();
      }
  }
```

Use of 'super' to access the super class constructor

Output:

```
Super class cal : 22
Sub class cal : 15
```

The concept of program 2.20 and 2.21 is same only some statements are changed. The super class 'Primary' has been added with a constructor of one parameter. This constructor has been called in sub class' constructor in statement2 by passing one parameter to super ( ). If the super class contains more number of constructors all that can be called using super keyword in sub class' constructor. Statement that contains 'super' must be a first statement in the sub class' constructor.

```
call to super must be first statement in constructor
```

If we write the statement containing 'super' elsewhere in the program, the same error will be flashed by compiler. By making effective use of 'super' we can call all the constructors in the multilevel inheritance hierarchy.

Creating multilevel hierarchy

Till this point we have seen only the concept of single inheritance consisting of only single subclass and single super class. However we can create any layers in the in the inheritance as we want. This concept is shown in figure 2.1 (c). General form of this concept is given below.

```
     class A
     {
          …….
          …….
          …….
     }
     class B extends A
     {
          …….
          …….
          …….
     }
     class C extends B
     {
          …….
          …….
          …….
     }
```

This hierarchy can be created at any level of inheritance. Program illustrates the concept of multilevel inheritance.

```
// Creating multilevel hierarchy
  class White
  {
      int num;
      White(int x)            //statement1
      {
         num = x;
      }
  }
  class Red extends White
  {
      Red(int y)
      {
         super(y);            //statement2
      }
      void print()
      {
         System.out.println("Value of num: "+num);
      }
  }
  class Magenta extends Red
  {
      int maggy;
      Magenta(int z)
      {
         super(z);            //statement3
         maggy = z;
      }
  }
  class MultiLevel
  {
      public static void main(String args[])
      {
         Magenta m = new Magenta(96);   //statement4
         m.print();
         System.out.println("Value of maggy : "+m.maggy);
      }
  }
```

Creation of multilevel hierarchy

Output:

```
Value of num: 96
Value of maggy : 96
```

In program 2.22, three different classes i.e. White, Red and Magenta are created. ‘Magenta’ is inherited from ‘Red’ and ‘Red’ is inherited from ‘White’. So ‘Magenta’ will contain all the members from both ‘Red’ and ‘White’ and including its own members. ‘Red’ will contain its own members and members from ‘White’. When statement4 is executed, statement3, statement2 and statement1 are executed in sequence. Because, ‘super ( )’ has linked all of them. Remember, super ( ) can only call immediate super class’ constructor. That is, ‘super’ from ‘Magenta’ can call only the constructor of ‘Red’ and super’ from ‘Red’ can call only the constructor of ‘White’. The object ‘m’ has access to both the variables used in the hierarchy i.e. ‘num’ and ‘maggy’. It can call the ‘print ( )’ method also.

Calling sequence of constructors

When the class hierarchy in multilevel inheritance is created, there is a sequence in which the default constructors are called. Constructors are always called in the order of derivation from super class to subclass. This order does not change though the ‘super ( )’ is used or not used. If super ( ) is not used, then the default constructor of each super class will be executed.

Program illustrates the sequence of calling the constructors.

```
//Calling sequence of constructors
  class GrandFather
  {
      GrandFather()
      {
        System.out.println("This is GrandFather");
      }
  }
  class Father extends GrandFather
  {
      Father()
      {
        System.out.println("This is Father");
      }
  }
  class Child extends Father
  {
      Child()
      {
        System.out.println("This is Child");
      }
  }
  class CallConstructor
  {
       public static void main(String args[])
       {
          Child roger = new Child();
       }
  }
```

Calling sequence of constructors

Output:

```
This is GrandFather
This is Father
This is Child
```

As we can see, the sequence of the call of default constructors is in the order of derivation. Why it is so? Because a super class has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

Method overriding

If we are familiar with C++, the concept of method overriding is Java's counterpart of function overriding. In a class hierarchy, when a method in a subclass has the same name, same arguments and same return type as a method in its super class, then the method in the subclass is said to override the method in the super class. 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 super class will be hidden.

```
//Method Overriding
  class Maths
  {
      int var1, var2, var3;
      Maths(int x, int y)
      {
          var1 = x;
```

```
        var2 = y;
    }
    void calculate()             //statement1
    {
        var3 = var1 + var2;
        System.out.println("Addition : "+var3);
    }
}
class Arithmetic extends Maths
{
    Arithmetic(int x,int y)
    {
       super(x,y);
    }
    void calculate()             //statement2
    {
        var3 = var1 - var2;
        System.out.println("Subtraction : "+var3);
    }
}
class OverRiding
{
    public static void main(String args[])
    {
        Arithmetic a = new Arithmetic(30,18);
        a.calculate();           //statement3
    }
}
```

Method Overriding

Output:

```
Subtraction : 12
```

Program 2.24 illustrates the concept of method overriding. 'Maths' is the super class of 'Arithmetic'. Both the classes contain method,

```
void calculate( )
```

Though, names return types and parameters of both the methods are same their definition is different. The subclass 'Arithmetic' will contain two copies of this method. When we call the method by creating the object of subclass, its own method will be called. So preference will be given to own method. In order to call the method from super class, we have to create the object of super class. Here the method 'calculate ( )' in subclass is said to be overridden by the method 'calculate ( )' in super class.

If we include following statement at the start of method 'calculate ( )' in subclass,

```
     super.calculate( );
```

Both of the methods will be executed when only one method is called. This gives one more advantage of 'super'. That is, 'super' can be used to call the method from super class when the methods are overridden.

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. Dynamic method dispatch is important because this is the only way in which Java implements run-time polymorphism.

A super class reference variable can refer to a subclass object. Java uses this concept to resolve calls to overridden methods at run time. When an overridden method is called through a super class 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 super class contains a method that is overridden by a subclass, then when different types of objects are referred to through super class reference variable, different versions of the method are executed. This concept is C++ counterpart of Virtual function in Java.

Program 2.25 illustrates the concept of dynamic method dispatch.

```
//Dynamic method dispatch
  class Principal
  {
     void message()
     {
        System.out.println("This is Principal");
     }
  }
  class Professor extends Principal
  {
     void message()
     {
        System.out.println("This is Professor");
     }
  }
  class Lecturer extends Professor
  {
     void message()
     {
        System.out.println("This is Lecturer");
     }
  }
  class Dynamic
  {
      public static void main(String args[])
      {
         Principal x = new Principal();
         Professor y = new Professor();
         Lecturer z = new Lecturer();
         Principal ref;  //reference variable of super class

         ref = x;              //statement1
         ref.message();

         ref = y;              //statement2
         ref.message();

         ref = z;              //statement3
         ref.message();
      }
  }
```

Dynamic method dispatch

Output:

```
This is Principal
This is Professor
This is Lecturer
```

Program 2.25 contains a multilevel hierarchy. 'Lecturer' is derived from 'Professor' and 'Professor' is derived from 'Principal'. All the three classes contain method,

```
     void message()
```

We have created a reference variable 'ref' of super class 'Principal'. Object of 'Principal' has been assigned to 'ref' in statement1. Then 'ref' can be used to refer the object of type 'Principal'. When the 'ref' is assigned with object of subclass 'Professor', it can be used to refer the 'Professor's object. This is done in statement2. Statement3 also performs the same function of assigning the subclass object to the super class reference. Here we are using the single reference variable 'ref' to refer all the class objects in the inheritance hierarchy. This is used to call all the versions of particular method at run time. Therefore, it is called as run time polymorphism. This concept is also applicable to tree structure inheritance or hierarchical inheritance.

Polymorphism allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

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 super class 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 object-oriented design brings code reuse and robustness. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Observe another practical example of dynamic method dispatch.

```
//Practical application of dynamic method
  class Square
  {
     int side,val;
     Square() {}
     Square(int x)
     {
        side = x;
     }
     void area()
     {
        val = side * side;
        System.out.println("Square side : "+side);
        System.out.println("Square area : "+val);
     }
  }
  class Rectangle extends Square
  {
     int len, wid, val;
     Rectangle(int x,int y)
     {
        len = x;
        wid = y;
     }
     void area()
     {
        val = len * wid;
        System.out.println("Rectangle length: "+len);
        System.out.println("Rectangle width : "+wid);
        System.out.println("Rectangle area : "+val);
     }
  }
  class Circle extends Square
  {
     float rad, val;
     Circle(float x)
     {
        rad = x;
     }
     void area()
     {
        val = 3.14f * rad * rad;
        System.out.println("Circle radius : "+rad);
        System.out.println("Circle area : "+val);
     }
  }
  class DynamicMethod
  {
      public static void main(String args[])
      {
          Square s = new Square(18);
          Rectangle r = new Rectangle(5,26);
          Circle c = new Circle(14.20f);

          Square ref;

          ref = s;
          ref.area();

          ref = r;
          ref.area();

          ref = c;
          ref.area();
      }
  }
```

Example of dynamic method dispatch

Output:

```
Square side : 18
Square area : 324
Rectangle length: 5
Rectangle width : 26
Rectangle area : 130
Circle radius : 14.2
Circle area : 633.1496
```

Abstract classes and methods

In some situations, it may be required to define a super class that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we will want to create a super class that only defines a generalized form that will be shared by all of its subclasses by leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. This can be done by creating a class as an abstract.

We can indicate that a method must always be redefined in a subclass by making the overriding compulsory. This can be achieved by preceding the keyword ‘abstract’ in the method definition with following form:

```
abstract datatype methodname(parameters);
```

There is no method body present. Any class that contains one or more abstract methods must also be declared as abstract. To declare a class abstract, we have to write ‘abstract’ in front of keyword ‘class’ in the definition. For example:

```
abstract class White
{
     ……
     abstract void paint();
     ……
     ……
}
```

We can not create the objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, we cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the super class, or be itself declared abstract.

Program shows the example of abstract class and methods.

```
//Abstract class and method
  abstract class Maths
  {
      int var1, var2, var3;
      abstract void calculate();
      void addition()
      {
          var3 = var1 + var2;
          System.out.println("Addition : "+var3);
      }
  }
  class Arithmetic extends Maths
  {
      Arithmetic(int x,int y)
      {
          var1 = x;
          var2 = y;
```

```
        }
        void calculate()
        {
            var3 = var1 - var2;
            System.out.println("Subtraction : "+var3);
        }
    }
    class AbstractClass
    {
        public static void main(String args[])
        {
            Arithmetic a = new Arithmetic(30,18);
            a.calculate();
            a.addition();
        }
    }
```

Using abstract class and method

Output:

```
Subtraction : 12
Addition : 48
```

As class ‘Maths’ is declared as abstract we can not create object of this class. ‘Maths’ class has is own concrete method named ‘addition’. This is acceptable. The overridden method ‘calculate’ is declared abstract in the super class with empty implementation. The concept of abstract classes and methods can be effectively implemented in large multilevel and hierarchical inheritance.

Application of ‘final’ in inheritance

We have already one use of the keyword ‘final’ i.e. to make a constant variable. It has been associated with inheritance two different ways. It is used to prevent the method overriding and to prevent the inheritance.

Preventing method overriding

While the method overriding is Java’s one of the most useful features, many times it is required to prevent the overriding. To disallow a method being overridden, we have to specify ‘final’ before the start of its declaration. Following program segment illustrates the use of ‘final’.

```
class First
{
     final void display()
     {
          System.out.println(“This is first class”);
     }
}
class Second extends First
{
     void display()      //statement1
     {
          System.out.println(“This is second class”);
     }
}
```

Here the method ‘display ( )’ is declared as final. So after compiling the program, the compiler will flash following error on statement1.

```
display() in Second cannot override display() in First;
overridden method is final
```

Normally, Java resolves calls to methods dynamically, at run time. This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

Preventing inheritance

Many times we require preventing a class from being inherited. For this purpose, we have to write keyword ‘final’ before class declaration. Declaring a class as ‘final’ automatically makes all its methods final too. Remember, it is illegal to declare a class both abstract and final.

Following code snippet illustrates this concept.

```
final class First
{
     // ….
}
class Second extends First         //statement1
{
     // ….
}
```

Here the class ‘First’ is declared as ‘final’ and the class ‘Second’ is inherited from ‘First’. If we compile this code, compiler will flash following error on statement1.

```
cannot inherit from final First
```

In Java’s class library, many methods and classes are declared as ‘final’ as well as ‘abstract’ separately. While using these in our program, we must be aware of that.

The Object class

There is one special class, ‘Object’, defined by Java. All other classes are subclasses of ‘Object’. That is, ‘Object’ is a super class of all other classes. This means that a reference variable of type ‘Object’ can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type ‘Object’ can also refer to any array.

The visibility controls

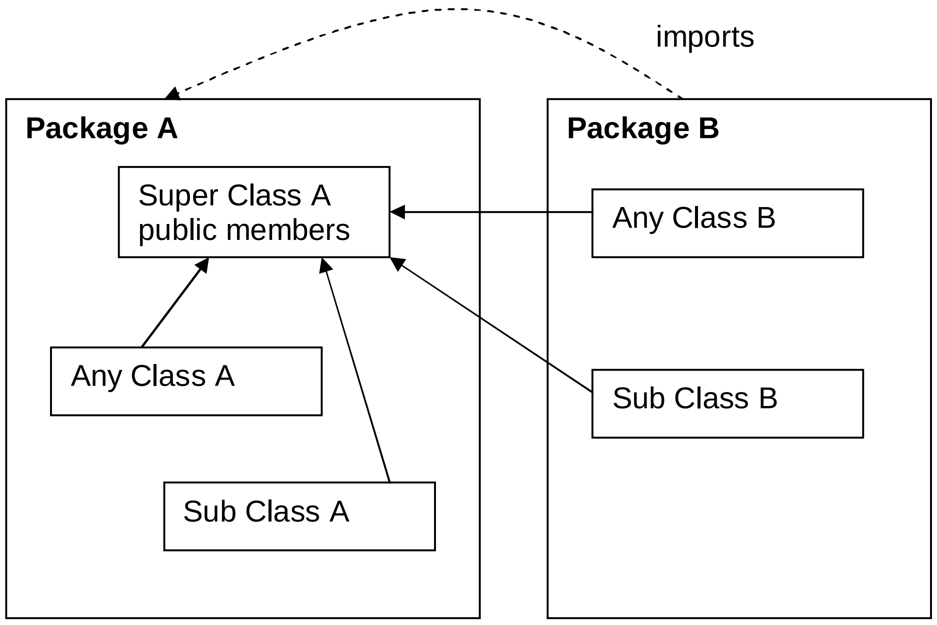
Many times it becomes necessary in some situations to restrict the access to certain variables and methods from outside the class. For example, we may not like the object of the class directly alter the value of variable or access the method. For this purpose Java apply the visibility modifiers to instance variables and methods. The visibility modifiers are also known as access modifiers. Java provides three different types of visibility modifiers i.e. public, private and protected. Java also defines the default access level. If you are familiar with C++, the concept of visibility controls for both of the languages is very same.

First of all, let’s introduce the concept of packages. A package in Java is an encapsulation mechanism that can be used to group related classes, interfaces and sub-packages. Package contains related classes. In order to use these classes and their methods in our program, we write import statement at the start of our program. A class defined in one package can be inherited from another package also. Java package library has implemented this concept as well as programmer can also implement this.

public access

If we declare any variable and method as ‘public’, it will be accessible to all the entire class and visible to all the classes outside the class. It can be accessed from outside package also. For example,

```
    public int val;
    public void display();
```



public accessibility

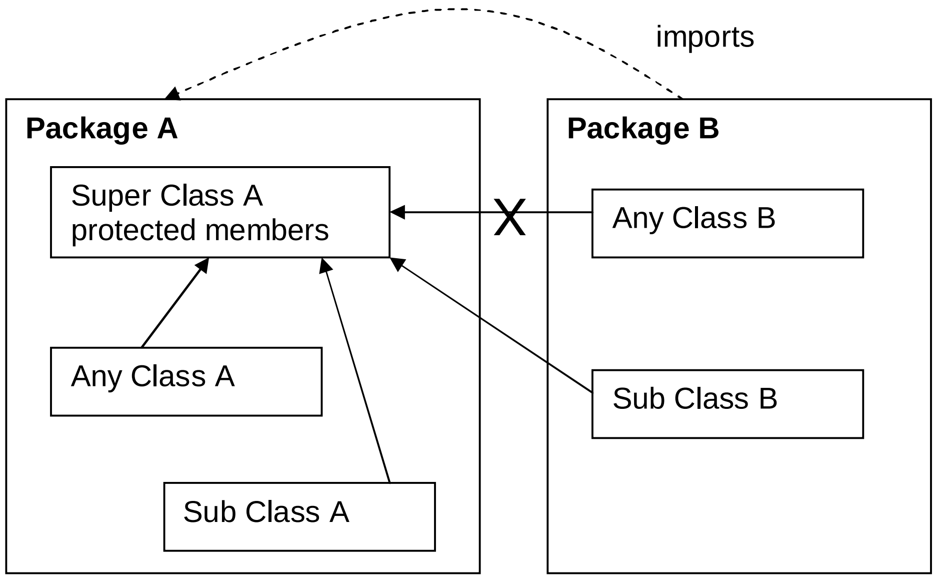
Figure shows the concept of public. The public members of super class A can be accessed from any class and sub class from the same package and any class and sub class from another package. So variable or method declared as public has widest possible visibility and accessible anywhere.

protected access

A protected member is accessible in all classes in the package containing its class and by and by all subclasses of its class in any package where this class is visible. In other words, non-subclasses in other packages cannot access protected members from other packages. Means, this visibility control is strongly related to inheritance.
For example,

```
protected int val;
protected void display();
```

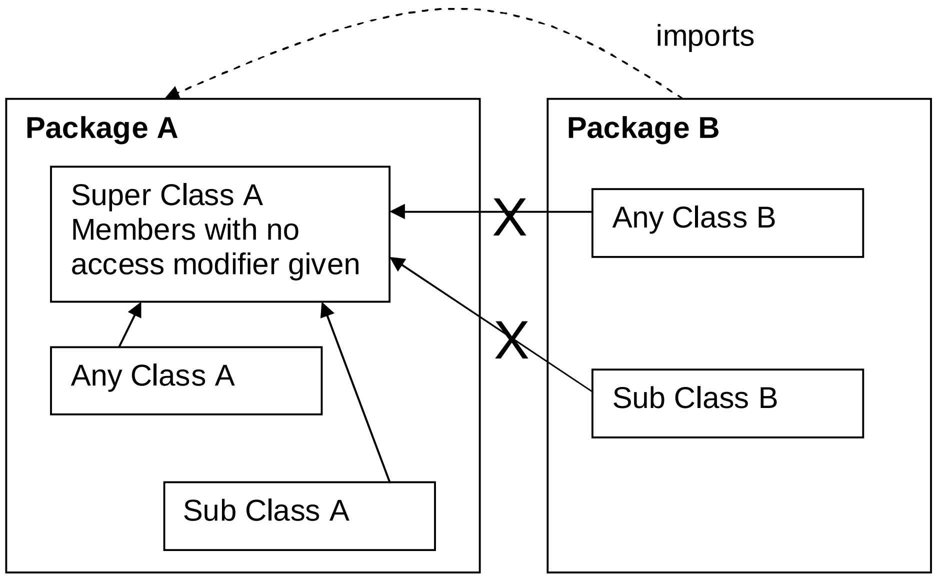
Fig. 2.4 illustrates the concept of protected accessibility. The protected members from super class can be accessed from sub class and any other class from the same package as well as sub class from another package. But it can not be accessed from any class from another package.



protected accessibility

default access / package access / friendly access

The default access can also be called as package access or friendly access. When no member accessibility modifier is specified, the member is only accessible inside its own package only. Even if its class is visible in another package, the member is not accessible there. Default member accessibility is more restrictive than the protected member accessibility.



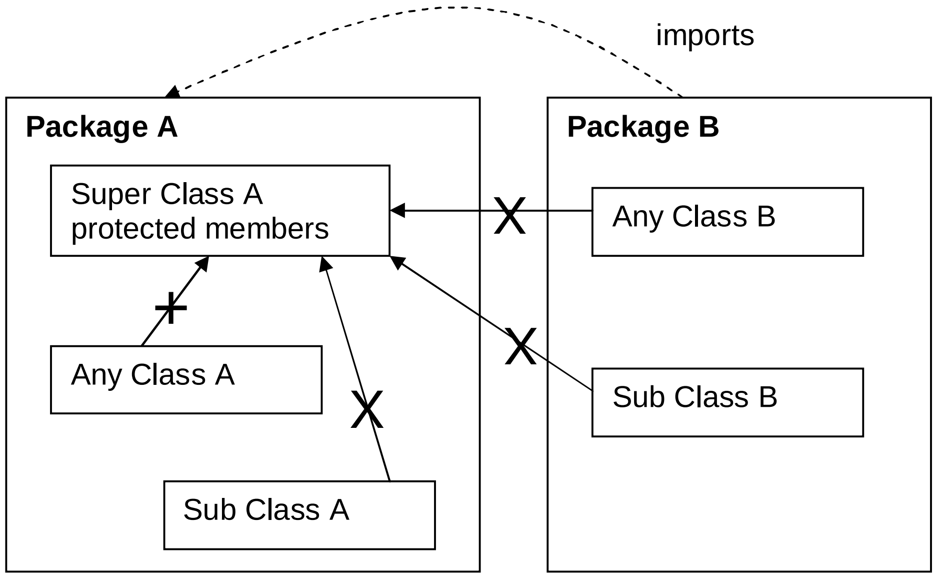
default accessibility

Figure shows the default accessibility. When we don't give any access modifier to variables and methods, these can not be accessed outside of the class. Only its own package member can access it.

private access

This is most restrictive than all other visibility controls. Private members are not accessible from any other class. They can not be accessed even from any subclass or any class from the same package. In order to declare a member as private, we just need to write 'private' in front of the member as,

```
private int val;
private void display();
```



private accessibility

As shown in the figure 2.6, the private members have access only to the members of its own class.

Arrays

An array is contiguous or related data items that share the common name. It offers the convenient means of grouping information. This means that all elements in the array have the same data type. A position in the array is indicated by a non-negative integer value called as index. An element at the given position is accessed by using this index. The size of array is fixed and can not increase to accommodate more elements.

In Java, arrays are objects. Arrays can be of primitive data types or reference types. In the former case, all elements in the array are of a specific primitive data type. In the latter case, all elements are references of a specific reference type. References in the array can then denote objects of this reference type or its subtypes. Each array object has a final field called length, which specifies the array size, that is, the number of elements the array can accommodate. The first element is always at index 0 and the last element at index n-1, where n is the value of the length field in the array.

Simple arrays are one-dimensional arrays, that is, a simple sequence of values. Since arrays can store object references, the objects referenced can also be array objects. This allows implementation of array of arrays. Remember, arrays work differently that they work in C/C++.

One dimensional arrays

One dimensional array is essentially, a list of same-typed variables. A list of items can be given one variable name using only one subscript. The general form of creating one dimensional array is,

```
datatype variablename[] = new datatype[size];
```

Here datatype declares the base type of the array. It determines the data type of each element that comprises the array. Thus datatype determines the type of the data that the array can hold. ‘variablename’ is the combined name given to the array. It can be used to refer the individual elements in the array. The set of first empty square brackets specifies that the variable of type array is created but actual memory is not allocated to it. In order to allocate the physical memory to the array the new operator is used. The ‘size’ determines number of elements in the array. Remember, unlike C/C++ it is possible to create an array with zero element in Java. The above declaration can be done in following way also.

```
datatype variablename[];
variablename = new datatype[size];
```

It is also possible to give the square brackets before the variable name as,

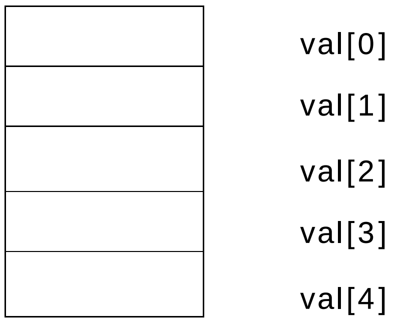
```
datatype []variablename = new datatype[size];
```

Let’s look at the example,

```
int val[] = new int[5];
```

This declaration will create five different variables in a contiguous memory locations referred by the common name ‘val’. The first element among them is val[0] to val[4]. Remember the index number of the array always start with 0.

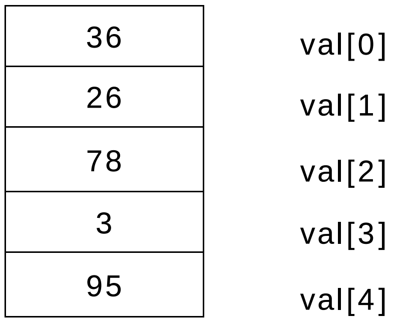
After this declaration the compiler will reserve five storage memory locations of 4 bytes each for storing integer variables as,



Individually, these memory locations are referred as a single variable name as,

```
val[0] = 36;
val[1] = 26;
val[2] = 78;
val[3] = 3;
val[4] = 95;
```

This would cause the array ‘val’ to store the values as shown below.



These variables may be used in programs just same as other Java variables. That is, they can be used in any Java expression. It can be used in Java’s input as well as output expressions. For example, following are the valid Java statements:

```
x = val[0] * val[1];
cal = val[4] + 15;
val[2] = cal + 92;
val[3] = val[1] + val[2];
num = val[z] * 5;
```

The index of the array can be integer constants, integer variables like z in above statement. If we refer the whole array at a time as,

```
x = val * 5;
```

Compiler will give error for this statement. Because the whole array can not be referred at a time for different operations like relational, arithmetic, boolean or logical.

Array initialization

When an array is created, each element of the array is set to the default initial value for its type i.e. zero for all numeric types, '\u0000' for char type, false for boolean and null for reference types. When we declare an array of a reference type, we are actually declaring array of variables of that type.

Putting the values inside the array is called as array initialization. When we initialize the array inside the program known as compile time initialization and when it is initialized by taking the values as input from keyboard it is called as run time initialization. For array also, before using it in the any expression in the program it must be initialized first. At compile time the individual element can be initialized as,

```
arrayname[index] = value;
```

or

```
arrayname[index] = expression;
```

For example:

```
val[0] = 36;
val[1] = 26;
val[2] = 78;
val[3] = 3;
val[4] = 95;
val[1] = (a + b) * 2;
val[0] = 95 / 19;
```

As Java is a robust language, it protects the array from overrun and under run. If we try to access the values beyond the array boundaries the compiler will generate the error. For this purpose an exception is defined i.e. ArrayIndexOutOfBoundsException.

We can also initialize array automatically in the same way as the ordinary variables when they are declared, as shown below:

```
datatype arrayname[] = {list of values separated by commas};
```

Note, that here size of the array is not given. The compiler will automatically allocate enough space for all the elements specified in the list. In case of C/C++, if number of elements is less that size then also compiler will keep unused locations will zero values. This is just the wastage of memory space. It is avoided in Java.

For Example:

```
int num[] = { 45, 12, 56, 10, 20, 86, 19, 46, 30 };
```

After this statement the array of size 9 will be created and num[0] to num[8] will be allocated with the given values.

It is also possible to assign an array object to another as,

```
int a[] = {5, 8, 6, 3, 4, 7};
int b[];
b = a;
```

This statement will assign the array values of ‘a’ to ‘b’. Both arrays will have same values. When we give the size while valued initialization of array as,

```
int a[6] = {5, 8, 6, 3, 4, 7};
```

Compiler will generate error. Loops may be used to initialize the large arrays as,

```
for(int i=0;i<100;i++)
{
     if(i<50)
       a[i] = 0;
     else
       a[i] = 1;
}
```

This will assign value 0 to all the elements from a[0] to a[49] and value 1 to a[50] to a[99].

Array length

All arrays store the allocated size in an attribute named length. We can treat the array as object in order to access number of elements of it. Such as,

For example:

```
int arr[] = {8, 6, 2, 4, 9, 3, 1};
int size = arr.length;
```

After the creation of the array ‘arr’, compiler will automatically decide the size of the array. This size or length or number of elements of the array can be obtained using attribute ‘length’ as shown above. The variable ‘size’ will contain value 7.

//Find maximum number from array

```
  class FindMax
  {
      public static void main(String args[])
      {
          int x[] = {8,4,6,2,0,3,11,5,1}; //array defined
          int max = x[0];  //consider first no. as max

          System.out.print("Array is : ");
          for(int i=0;i<x.length;i++)  //print array
              System.out.print(x[i]+", ");

          for(int i=1;i<x.length;i++)
          {
              if(x[i] > max) //check each no. with max
                 max = x[i];
          }
          System.out.println("\nMaximum is "+max);
      }
  }
```

Finding maximum number from the array

Output:

```
Array is : 8, 4, 6, 2, 0, 3, 11, 5, 1,
Maximum is 11
```

Program 3.1 demonstrates the use of array and its attribute ‘length’.

Anonymous arrays

When the array does not contain any name it is called as anonymous array. It takes the following form:

```
new datatype[]  {list of values separated by comma};
```

Generally the anonymous arrays are used to pass the values to a method as array with values. Program 3.2 demonstrates the use of anonymous arrays.

```
//Program that used anonymous array
  class Anonymous
  {
      void printArray(int arr[])
      {
          for(int i=0;i<arr.length;i++)
            System.out.print(arr[i]+", ");
      }
      public static void main(String args[])
      {
          Anonymous a = new Anonymous();
          System.out.println("Array is: ");
          a.printArray(new int[] {5,7,9,3,4}); //statement1
      }
  }
```

Program uses the anonymous array

Output:

```
Array is:
5, 7, 9, 3, 4,
```

Program 3.2 shows how to use the concept of anonymous arrays. An array with no name is created in the statement1. It is used to pass the array parameter to a method. All its values are substituted with actual array parameter 'arr' of the method 'printArray( )'. Remember, we can not directly pass the values as, {5,7,9,3,4}. We have to create the array at function call.

Two dimensional arrays

It is also called as array of arrays. Many times it is required to manipulate the data in table format or in matrix format which contains rows and columns. In these cases, we have to give two dimensions to the array. That is, a table of 5 rows and 4 columns can be referred as,

```
table[5][4]
```

The first dimension 5 is number of rows and second dimension 4 is number of columns. In order to create such two dimensional arrays in Java following syntax should be followed.

```
datatype arrayname[][] = new datatype[row][column];
or
datatype [][]arrayname = new datatype[row][column];
```

For example:

```
int table[][] = new int[5][4];
```

This will create total 20 storage locations for two dimensional arrays as,

	Columns			
	[0][0]	[0][1]	[0][2]	[0][3]
	[1][0]	[1][1]	[1][2]	[1][3]
Rows	[2][0]	[2][1]	[2][2]	[2][3]

[3][0]	[3][1]	[3][2]	[3][3]
[4][0]	[4][1]	[4][2]	[4][3]

We can store the values in each of these memory locations by referring there respective row and column number as,

```
table[2][3] = 10;
table[1][1] = -52;
```

Like one dimensional arrays, two dimensional arrays can also be initialized at compile time as,

```
int table[2][2] = {8, 5, 9, 6};
```

It will initialize the array as shown below,

8	5
9	6

Following declaration and initialization is more understandable than previous one.

```
int table[][] = {{8, 5}, {9, 6}};
or
int table[][] = {
                    {8, 5},
                    {9, 6}
                  };
```

The large two dimensional arrays can also be initialized using loops such as,

```
int arr[][] = new int[5][5];
int z = 0;
for(int x=0;x<5;x++)
{
   for(int y=0;y<5;y++)
   {
     arr[x][y] = z;
     z++;
   }
}
```

This will initialize the array ‘arr’ as,

0	1	2	3	4
5	6	7	8	9

10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Same can be printed using nesting of two loops as,

```
for(int x=0;x<5;x++)
   for(int y=0;y<5;y++)
       System.out.println(“ ”+arr[x][y]);
```

Basically, the two dimensional array are used for manipulating different matrix operations. Program 3.3 shows the application of two dimensional arrays.

```
//addition of matrices
  class TwoD
  {
      public static void main(String args[])
      {
         int x[][] = { {8,5,6},
                       {1,2,1},
                       {0,8,7}
                     };

         int y[][] = { {4,3,2},
                       {3,6,4},
                       {0,0,0}
                     };
         System.out.println("First matrix: ");
         for(int i=0;i<3;i++)
         {
            for(int j=0;j<3;j++)
              System.out.print(" "+x[i][j]);
            System.out.print("\n");
         }
         System.out.println("Second matrix: ");
         for(int i=0;i<3;i++)
         {
            for(int j=0;j<3;j++)
              System.out.print(" "+y[i][j]);
            System.out.print("\n");
         }
         System.out.println("Addition: ");
         for(int i=0;i<3;i++)
         {
            for(int j=0;j<3;j++)
              System.out.print(" "+(x[i][j]+y[i][j]));
            System.out.print("\n");
         }
```

```
        }
    }
```

Addition of matrices using 2D arrays

Output:

```
First matrix:
 8 5 6
 1 2 1
 0 8 7
Second matrix:
 4 3 2
 3 6 4
 0 0 0
Addition:
 12 8 8
 4 8 5
 0 8 7
```

Variable size arrays

Java treats the multidimensional arrays as "arrays of arrays". So it is possible to declare the two-dimensional arrays as follows:

```
int table[][] = new int[3][];
table[0] = new int[5];
table[1] = new int[2];
table[2] = new int[3];
```

These statements will create the two dimensional arrays with different lengths as shown in fig. 3.2.

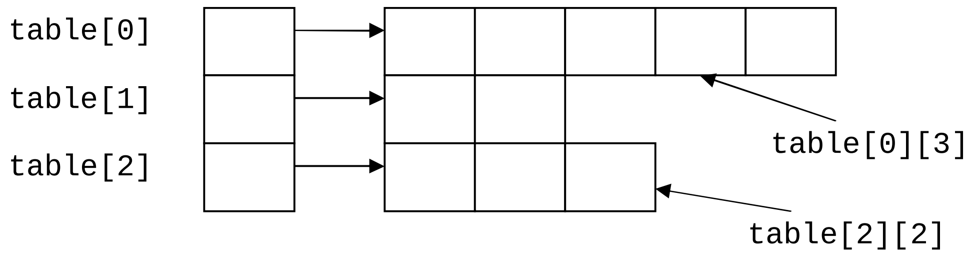


Fig. 3.2 Variable size arrays

Arrays class

Java library has defined a class named Arrays. It provides various methods that are useful for working with the arrays. All these methods are static. In order to use these methods in the program we have to import the class Arrays from java.util package.

Sorting the arrays

The sort method of Arrays class sorts the elements of the array in ascending order. This method has two versions. First version, shown here, sorts the entire array.

```
static void sort(byte array[])
static void sort(short array[])
static void sort(int array[])
static void sort(float array[])
static void sort(long array[])
static void sort(double array[])
static void sort(char array[])
static void sort(Object array[])
```

Here, 'array[ ]' is the array to be sorted. For example:

```
int a[] = {5,9,6,4,7};
Arrays.sort(a);
```

After execution of these statements the array 'a' will get sorted in ascending order. The second version of sort( ) enables you to specify a range within an array that you want to sort. Its forms are shown here:

```
static void sort(byte array[], int start, int end)
static void sort(short array[], int start, int end)
static void sort(int array[], int start, int end)
static void sort(float array[], int start, int end)
static void sort(long array[], int start, int end)
static void sort(double array[], int start, int end)
static void sort(char array[], int start, int end)
static void sort(Object array[], int start, int end)
```

Here, the range beginning at start and running through end–1 within array will be sorted. All of these methods can throw an IllegalArgumentException if start is greater than end, or an ArrayIndexOutOfBoundsException if start or end is out of bounds. For example:

```
int a[] = {5,9,6,4,7,0,3,2};
Arrays.sort(a,1,5);
```

Searching an element in array

The method binarySearch( ) uses the binary search to find the index of specific value inside the array. This method must be used only for sorted arrays. Following are some forms of this method:

```
static int binarySearch(byte array[], byte value)
static int binarySearch(short array[], short value)
static int binarySearch(int array[], int value)
static int binarySearch(float array[], float value)
static int binarySearch(long array[], long value)
static int binarySearch(double array[], double value)
static int binarySearch(char array[], char value)
static int binarySearch(Object array[], Object value)
```

Here the ‘value’ is the value to be searched in the ‘array[ ]’. All these methods return the index position of the searched element. In all cases, if value exists in array, the index of the element is returned. Otherwise, a negative value is returned. For example:

```
int arr[] = {0,2,3,5,8,9};
int val = Arrays.binarySearch(arr, 8);
```

After the execution of these statements, the value of variable ‘val’ will be the index of 8 that is 4.

Checking the arrays for equality

For checking whether two arrays are equal or not we can use the equals( ) method of Arrays class. It returns true if arrays are equal otherwise false. It has following forms:

```
static boolean equals(boolean arr1[], boolean arr2[])
static boolean equals(byte arr1[], byte arr2[])
static boolean equals(short arr1[], short arr2[])
static boolean equals(int arr1[], int arr2[])
static boolean equals(long arr1[], long arr2[])
static boolean equals(char arr1[], char arr2[])
static boolean equals(float arr1[], float arr2[])
static boolean equals(double arr1[], double arr2[])
static boolean equals(Object arr1[], Object arr2[])
```

Here, arr1 and arr2 are the two arrays that are used to check for equality. For example:

```
int x[] = {8,4,6,2,5};
int y[] = {8,4,6,2,5};
if(Arrays.equals(x,y))   //statement1
   System.out.println(“Arrays are equal”);
```

Here statement1 will check the arrays x and y for equality and returns true so we will get the output “Arrays are equal”.

Filling the arrays

The fill( ) method of Arrays class is used to fill the entire array of some range of the array with specific value. It is one of the useful methods for array initialization. This method has two versions. The first version has following forms which fills the entire array.

```
static void fill(boolean array[], boolean value)
static void fill(byte array[], byte value)
static void fill(short array[], short value)
static void fill(int array[], int value)
static void fill(float array[], float value)
static void fill(long array[], long value)
static void fill(double array[], double value)
static void fill(char array[], char value)
static void fill(Object array[], Object value)
```

Here, all the elements of the 'array[ ]' will be filled with the 'value'. Second version of the method fill( ) assigns value to the subset of the array. Its form is as follows:

```
static void fill(byte array[], int start, int end, byte value)
```

This method also has different forms for all the data types including Object as above methods were. Here, 'array[ ]' is the array to be filled with 'value' from index position 'start' to 'end-1'. These methods may all throw an IllegalArgumentException if start is greater than end or an ArrayIndexOutOfBoundsException if start or end is out of bounds.

Program 3.4 demonstrates the use of all these above methods of Arrays class.

```
// Demonstrate Arrays and its methods
  import java.util.Arrays;
  class ArraysDemo
  {
      public static void main(String args[])
      {
        int array[] = new int[10];
        for(int i = 0; i < 10; i++)
           array[i] = 3 * i;

        // display, sort, display
        System.out.print("Original contents: ");
        display(array);
        Arrays.sort(array);
        System.out.print("Sorted: ");
        display(array);

        // fill and display
        Arrays.fill(array, 2, 6, 5);
        System.out.print("After fill(): ");
        display(array);

        // sort and display
        Arrays.sort(array);
        System.out.print("After sorting again: ");
        display(array);

        // binary search for 21
        System.out.print("The value 21 is at location ");
        int index = Arrays.binarySearch(array, 21);
        System.out.println(index);
      }
      static void display(int array[])
      {
        for(int i = 0; i < array.length; i++)
           System.out.print(array[i] + " ");
        System.out.println("");
      }
  }
```

Demonstrate Arrays and its methods

Output:

```
Original contents: 0 3 6 9 12 15 18 21 24 27
Sorted: 0 3 6 9 12 15 18 21 24 27
After fill(): 0 3 5 5 5 5 18 21 24 27
After sorting again: 0 3 5 5 5 5 18 21 24 27
The value 21 is at location 7
```

String class

As is the case in most other programming languages, in Java a string is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type String. Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed.

Somewhat unexpectedly, when we create a String object, we are creating a string that cannot be changed. That is, once a String object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, this is not the case. We can still perform all types of string operations. The difference is that each time we need an altered version of an existing string; a new String object is created that contains the modifications. The original string is left unchanged.

For those cases in which a modifiable string is required, there is a companion class to String called StringBuffer, whose objects contain strings that can be modified after they are created. Both the String and StringBuffer classes are defined in java.lang package. Thus, they are available to all programs automatically. Both are declared final, which means that neither of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations.

The strings within objects of type String are unchangeable means that the contents of the String instance cannot be changed after it has been created. However, a variable declared as a String reference can be changed to point at some other String object at any time.

Creating the strings

The String class has several constructors defined. To create an empty string the default constructor is used. For example:

```
String s = new String();
```

It will create the instance of the string with no characters in it. If we want to initialize the string with values we can use following constructor.

```
String(char chars[])
```

For example:

```
char chars[] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’};
String s = new String(chars);
```

Direct initialization is also possible as,

```
String s = “Hello”;
```

We can specify a sub-range of a character array as an initializer using the following constructor:

```
String(char chars[ ], int startIndex, int numChars)
```

Here, startIndex specifies the index at which the sub-range begins, and numChars specifies the number of characters to use. Here is an example:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

This initializes s with the characters “cde”. Following three constructors can also be used to create the strings

```
String(String strObj)
String(byte asciiChars[])
String(byte asciiChars[], int startIndex, int numChars)
```

Here, ’strObj’ is another String object and ‘asciiChars’ is the array of bytes to create the string from the given ASCII characters.

String operations and methods

Following are the methods of the String class.

length( )

This method is used to find total number of characters from the string.
Syntax:

```
int length()
```

Example:

```
String s = “Pramod”;
int len = s.length();
```

This will printf the value 6 which is the length of string ‘s’.

concat( )

This is used to join two strings.
Syntax:

```
String concat(String str)
```

Example:

```
String s = “First”;
String t = “Second”;
s.concat(t);
```

The contents of the string ‘s’ will be “FirstSecond” after the execution of these statements. The operator + can also be used to join two strings together. We have used this operator several times in program for the same purpose. For example:

```
String s = “How ” + “are ” + “you ?”;
String x = “Number = ” + 2;
```

charAt( )

This method extracts a single character from the string.
Syntax:

```
char charAt(int index)
```

It extracts the character of invoking String from position specified by ‘index’.
Example:

```
String s = “Indian”;
char ch = s.charAt(2); //ch will be ‘d’ after this
```

getChars( )

It extracts the sequence of characters from the string.
Syntax:

```
void getChars(int s, int e, char target[], int tstart)
```

This will extract the characters from position s to e from invoking string and store it in character array target from position ‘tstart’.
Example:

```
String s = “I Love Java”;
char ch[] = new char[4];
s.getChars(2, 5, ch, 0); //ch will contain “love”
```

getBytes( )

It converts the string characters into ASCII equivalents and stores it into the byte array. Syntax:

```
byte[] = getBytes( );
```

Example:

```
String s = “Programming”;
byte b[] = new byte[10];
b = s.getBytes();
```

toCharArray( )

It converts all the characters of the string into the character array.
Syntax:

```
char[] toCharArray( )
```

Example:

```
     String s = “Programming”;
     char ch[] = new char[10];
     ch = ch.toCharArray();
```

equals( ) and equalsIgnoreCase( )

These methods are used to check two strings for equality and returns the boolean value true if they are equal else false. The equals( ) is case sensitive and equalsIgnoreCase( ) is case insensitive method.
Syntax:

```
     boolean equals(String str)
     boolean equalsIgnoreCase(String str)
```

Example:

```
     String a = “Hello”;
     String b = “HELLO”;
     if(a.equals(b))               //false
         System.out.println(“Equals in case”);
     if(a.equalsIgnoreCase(b))     //true
         System.out.println(“Equals in characters”);
```

Remember, the equals( ) method compares the characters inside a String object. The == operator compares two object references to see whether they refer to the same instance. The following program shows how two different String objects can contain the same characters, but references to these objects will not compare as equal:

```
  // equals() and ==
  class Equals
  {
       public static void main(String args[])
       {
            String s1 = "Nation";
            String s2 = new String(s1);
            System.out.println(s1+" equals "+s2+" : "+
                s1.equals(s2));
            System.out.println(s1+" == "+s2+" : "+(s1==s2));
       }
  }
```

Difference between equals and ==

Output:

```
Nation equals Nation : true
Nation == Nation : false
```

The variable s1 refers to the String instance created by “Nation”. The object referred to by s2 is created with s1 as an initializer. Thus, the contents of the two String objects are identical, but they are distinct objects. This means that s1 and s2 do not refer to the same objects and are, therefore, not ==, as is shown here by the output of the preceding example.

startsWith( ) and endsWith( )

The method ‘startsWith( )’ determines whether a given string starts with specified string or not. Conversely, the ‘endsWith( )’ determines whether a given string ends with specified string or not.
Syntax:

```
boolean startsWith(String str)
boolean endsWith(String str)
boolean startsWith(String str, int startIndex)
```

Here, str is the string to be checked and startIndex is the index from which the comparison is to be made.
Example:

```
boolean a = “camera”.startsWith(“cam”);      //true
boolean b = “camera”.endsWith(“mere”);  //false
```

compareTo( ) and compareToIgnoreCase( )

For sorting applications, we need to know which string is less than, equal to, or greater than the next. In such cases the compareTo( ) or compareToIgnoreCase() method can be used. It is used to check whether one string is greater than, less than or equal to another string or not. First compareTo( ) is case-sensitive and second form compareToIgnoreCase( ) is case insensitive. Syntax:

```
int compareTo(String str)
int compareToIgnoreCase(String str)
```

Here, str is the string to be compared with invoking object’s string. It will return an integer value. When,

value<0: invoking string is less that str
value>0: invoking string is greater that str
value=0: both the strings are equal

Example:

```
String str1 = “catch”;
String str2 = “match”;
if(str1.compareTo(str2)<0)    //true
   System.out.println(“str2 is greater”);
```

```
     else
        System.out.println("str1 is greater");
```

indexOf( )

If we want to search a particular character or substring in the string then this method can be used. This method returns the index of the particular character or sting which is searched. It can be used in different ways.

```
     int indexOf(int ch)
```

This method searches first occurrence of the character from the string.

```
     int indexOf(String str)
```

This method searches first occurrence of the substring from another string.

```
     int indexOf(int ch, int startIndex)
     int indexOf(String str, int startIndex)
```

Here the 'startIndex' specifies the index at which point of search begins.
Example:

```
     String s = "Maharashtra";
     String t = "Tamilnadu";
     int in = s.indexOf('a');       //in will be 1
     int mn = t.indexOf('a', 2);    //mn will be 6
     int x = s.indexOf("tra");      //x will be 8
```

lastIndexOf( )

This method searches the last occurrence of the particular character or string in the invoking object's string and returns index of that. Like indexOf( ) this method is also having four different forms.

```
     int lastIndexOf(int ch)
     int lastIndexOf(String str)
     int lastIndexOf(int ch, int startIndex)
     int lastIndexOf(String str, int startIndex)
```

Here 'ch' is the character and 'str' is the string to be searched inside the invoking string from last. In case of last two forms of the method, the search begins from 'startIndex' to zero.
Example:

```
String s = “Maharashtra”;
String t = “Tamilnadu”;
int in = s.lastIndexOf(‘a’);       //in will be 10
int mn = t.lastIndexOf(‘a’, 2);    //mn will be 1
int x = s.lastIndexOf(“ra”);       //x will be 9
```

substring( )

It is used to extract a substring from the string. It has two different forms:

```
String substring(int start)
String substring(int start, int end)
```

The first form extracts substring from ‘start’ to end of the string. Second form extracts the substring from position ‘start’ to the position ‘end’ of the invoking string.
Example:

```
String str = “Are you a Marathi guy”;
String s = s.substring(10);   //s will be “Marathi guy”
String t = s.substring(10,16); //t will be “Marathi”
```

replace( )

This method is used to replace all the occurrences of a character with another character of the string. It has following form:
Syntax:

```
String replace(char original, char replacement)
```

Here ‘original’ is character to be replaced and ‘replacement’ is new character exchanged instead of that.
Example:

```
String s = “mama”.replace(‘m’,’k’);
```

It will initialize the string ‘s’ with value “kaka”.

trim( )

It is used to remove the leading and trailing white spaces from the string. Remember, it does not remove the spaces in between the characters of the string.
Syntax:

```
String trim()
```

Example:

```
     String s = “   What is this?      “;
     String k = s.trim();
```

After this, the string ‘k’ will contain “What is this?”

toLowerCase( ) and toUpperCase( )

These methods will convert the strings into lower case and upper case respectively. Non-alphabetical characters such as digits and symbols are unaffected.
Syntax:

```
     String toLowerCase( )
     String toUpperCase( )
```

Example:

```
     String s = “Pramod and Kunda”;
     String t = s.toUpperCase();
     String u = s.toLowerCase();
```

After execution of these statements, the value of t will be “PRAMOD AND KUNDA” and u will be “pramod and kunda”.

String arrays

We can also create and use arrays that contain strings. The statement

```
     String sArray[] = new String[10];
```

This will create the sArray of size 10 to store ten different string elements. It can be treated just same as other arrays. Program 3.6 demonstrates the use of string arrays using its methods.

```
  //Sorting the strings
  class SortString
  {
      public static void main(String args[])
      {
          String s[] = {"Nitin","Arjun","Pramod",
                        "Vikas","Vinay","Sandeep"};
          String temp;
          for(int i=0;i<s.length;i++)
            for(int j=0;j<s.length;j++)
              if(s[i].compareTo(s[j])<0)
              {
                   temp = s[i];
                   s[i] = s[j];
                   s[j] = temp;
              }
         System.out.println("Sorted strings : ");
         for(int i=0;i<s.length;i++)
            System.out.println(s[i]);
     }
 }
```

Sorting the strings

Output:

```
Sorted strings :
Arjun
Nitin
Pramod
Sandeep
Vikas
Vinay
```

StringBuffer class

String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writeable character sequences. It is possible to insert the characters anywhere inside the StringBuffer. StringBuffer will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

Creating StringBuffer

StringBuffer class defines three different constructors:

```
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
```

The first form i.e. default constructor reserves room for 16 characters without reallocation. The second form accepts an integer argument that explicitly sets the size of the buffer with given value. The third form accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation. StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, StringBuffer reduces the number of reallocations that take place.

StringBuffer operations and methods

Several methods of classes String and StringBuffer are same but as per the functionality, StringBuffer has been added with some special methods.

length( ) and capacity( )

The 'length( )' finds total number of characters that a StringBuffer is having in it. The total allocated capacity (no. of characters) can be found using the capacity( ) method.
Syntax:

```
int length()
int capacity()
```

Example:

```
StringBuffer sb = new StringBuffer("Oriya");
int len = sb.length());        //len will be 5
int cap = sb.capacity());      //cap will be 21
```

Here the variable len will contain total number of characters i.e. 5 and cap will contain capacity 21 i.e. actual length + additional room for 16 characters.

ensureCapacity( )

If we want to pre-allocate room for a certain number of characters after a StringBuffer has been created, we can use ensureCapacity( ) to set the size of the buffer. This is useful if we know in advance that we will be appending a large number of small strings to a StringBuffer. ensureCapacity( ) has this general form:

```
void ensureCapacity(int capacity)
```

Here, capacity specifies the size of the buffer.
Example:

```
StringBuffer sb = new StringBuffer(10);
sb.ensureCapacity(20);
```

setLength( )

It is used to set the length of the buffer within a StringBuffer object. Its general form is:

```
void setLength(int len)
```

Here, len specifies the length of the buffer. This value must be nonnegative. When we increase the size of the buffer, null characters are added to the end of the existing buffer. If we call setLength( ) with a value less than the current value returned by length( ), then the characters stored beyond the new length will be lost.

setCharAt( )

If we want to set the character at certain index in the StringBuffer with specified value, this method can be used. It has following form:

```
void setCharAt(int index, char ch)
```

Here, ‘index’ is the position within StringBuffer whose value is to be set with value ‘ch’.
Example:

```
StringBuffer sb = new StringBuffer(“Kolkata”);
sb.setCharAt(0,‘C’);
```

After execution of these statements, ‘sb’ will be “Colkata”.

append( )

The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has overloaded versions for all the built-in types and for Object. Following are a few of its forms:

```
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)
```

Example:

```
StringBuffer sb = new StringBuffer("cricket");
int x = 52;
float f = 13.4f;
StringBuffer a = sb.append(x); //a will be cricket52
StringBuffer b = sb.append(f); //b will be cricket5213.4
```

insert( )

This method is used to insert one string, character or object into another string. Forms of this method are:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

Here the ‘index’ specifies the index position at which point the string will be inserted into the invoking StringBuffer object.
Example:

```
StringBuffer sb = new StringBuffer("I Java!");
sb.insert(2, "love ");
After this, the contents of ‘sb’ will be “I love Java!”
```

reverse( )

We can reverse the characters inside the StringBuffer using reverse( ) method.
Syntax:

```
StringBuffer reverse()
```

This method returns the reversed object upon which it is called.
Example:

```
StringBuffer s = new StringBuffer(“Yellow”);
s.reverse(); //s will be “wolleY”
```

delete( ) and deleteCharAt( )

These methods are used to delete a single character or a sequence of characters from the StringBuffer.
Syntax:

```
StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)
```

The delete( ) method deletes a sequence of characters from the invoking object. Here, ‘startIndex’ specifies the index of the first character to remove, and ‘endIndex’ specifies an index one past the last character to remove. Thus, the substring deleted runs from ‘startIndex’ to ‘endIndex’–1. The resulting StringBuffer object is returned. The deleteCharAt( ) method deletes the character at the index specified by ‘loc’. It returns the resulting StringBuffer object.
Example:

```
StringBuffer sb = new StringBuffer("Chandramukhi");
sb.delete(4, 7);    //sb will be “Chanmukhi”
sb.deleteCharAt(0); //sb will be “hanmukhi”
```

The methods such as getChars, replace, substring, indexOf and lastIndexOf are having the same syntax and use as that of String class except here the manipulation is with StringBuffer only.

Vector

The package java.util contains a library of Java's utility classes. One of them is Vector class. It implements a dynamic array which can hold the array of any type and any number. These objects do not have to be homogenous. Capacity of the Vector can be increased automatically.

Creation of Vector

Vector class defines three different constructors:

```
Vector()
Vector(int size)
Vector(int size, int incr)
```

The first form creates a default vector, which has an initial size of 10. The second form creates a vector whose initial capacity is specified by 'size' and the third form creates a vector whose initial capacity is specified by 'size' and whose increment is specified by 'incr'. The increment specifies the number of elements to allocate each time when new elements are added in the vector.

All vectors start with an initial capacity. After this initial capacity is reached, the next time when we attempt to store an object in the vector, the vector automatically allocates space for that object plus extra room for additional objects. By allocating more than just the required memory, the vector reduces the number of allocations that must take place. The amount of extra space allocated during each reallocation is determined by the increment that we specify when we create the vector. If we don't specify an increment, the vector's size is doubled by each allocation cycle.

The Vector class defines three protected data members:

```
int capacityIncrement;
int elementCount;
Object[] elementData;
```

The increment value of vector is stored in 'capacityIncrement'. The number of elements currently in the vector is stored in 'elementCount'. The array that holds the vector's elements is stored in 'elementData'.

Vector operations and methods

Following are the methods defined by Vector class for its operation.

void addElement(Object element):

The object specified by element is added to the vector.

int capacity()

It returns the capacity of the vector.

Object clone()

It returns a duplicate copy of the invoking vector.

boolean contains(Object element)

It returns true if ‘element’ is contained by the vector, and returns false if it is not.

void copyInto(Object array[])

The elements contained in the invoking vector are copied into the array specified by ‘array’.

Object elementAt(int index)

It returns the element at the location specified by ‘index’.

void ensureCapacity(int size)

This method sets the minimum capacity of the vector to ‘size’.

Object firstElement( )

It returns the first element in the vector.

int indexOf(Object element)

It returns the index of the first occurrence of ‘element’. If the object is not found in the vector, –1 is returned.

int indexOf(Object element, int start)

It returns the index of the first occurrence of ‘element’ at or after ‘start’. If the object is not in that portion of the vector, –1 is returned.

void insertElementAt(Object element, int index)

It adds ‘element’ to the vector at the location specified by ‘index’.

boolean isEmpty()

This method returns true if the vector is empty and returns false if it contains one or more elements.

Object lastElement()

It returns the last element in the vector.

int lastIndexOf(Object element)

It returns the index of the last occurrence of ‘element’. If the object is not in the vector, –1 is returned.

int lastIndexOf(Object element, int start)

It returns the index of the last occurrence of ‘element’ before ‘start’. If the object is not in that portion of the vector, –1 is returned.

void removeAllElements()

This method empties the vector. After is execution, the size of the vector is zero.

boolean removeElement(Object element)

It removes ‘element’ from the vector. If more than one instance of the specified object exists in the vector, then it is the first one that is removed. This method returns true if successful and false if the object is not found.

void removeElementAt(int index)

It removes the element at the location specified by ‘index’.

void setElementAt(Object element, int index)

Here, the location specified by ‘index’ is assigned ‘element’.

void setSize(int size)

It sets the number of elements in the vector to ‘size’. If the new size is less than the old size, elements are lost. If the new size is larger than the old size, ‘null’ elements are added.

int size()

It returns the number of elements currently in the vector.

String toString()

It returns the string equivalent of the vector.

void trimToSize()

It sets the vector’s capacity equal to the number of elements that it currently holds. That is it makes capacity equal to size.

Program 3.7 demonstrates the use of some Vector methods and operations.

```
// Demonstration various Vector methods.
  import java.util.Vector;
  class VectorDemo
  {
      public static void main(String args[])
      {
        Vector v = new Vector(3, 2);
        System.out.println("Initial size: " + v.size());
        System.out.println("Initial capacity: " +
                       v.capacity());

        v.addElement(new Integer(23));
        v.addElement(new Float(9.6f));
        v.addElement(new String("Hello"));
        v.addElement(new Integer(42));

        System.out.println("Capacity after 4 additions: " +
                       v.capacity());
        v.addElement(new Double(63.2));
        System.out.println("Current capacity: " +
                       v.capacity());
        System.out.println("Current size: " +
                       v.size());
        v.addElement(new Double(19.44));
        v.addElement(new Integer(7));

        System.out.println("First element: " +
                       (Integer)v.firstElement());
        System.out.println("Last element: " +
                       (Integer)v.lastElement());

        if(v.contains(new Integer(42)))
            System.out.println("Vector contains 3.");

        System.out.println("Vector : "+v.toString());
        v.setSize(8);
        System.out.println("Vector : "+v.toString());

        Object o = new Object();
        o = v.clone();
        System.out.println("New Vector: "+o);

        v.removeAllElements();
        if(v.isEmpty())
           System.out.println("Vector is empty");
  }
}
```

Vector methods demonstration

Output:

```
Initial size: 0
Initial capacity: 3
Capacity after 4 additions: 5
Current capacity: 5
Current size: 5
First element: 23
Last element: 7
```

```
Vector contains 3.
Vector : [23, 9.6, Hello, 42, 63.2, 19.44, 7]
Vector : [23, 9.6, Hello, 42, 63.2, 19.44, 7, null]
New Vector: [23, 9.6, Hello, 42, 63.2, 19.44, 7, null]
Vector is empty
```

For adding elements, the Object representation of the primitive data types is used. We can not directly add the values inside the vector. This object representation is referred as wrapper class. We will see this in next topic.

Wrapper classes

The primitive data types of Java such as int, char, float are not the part of any object hierarchy. They are always passed by value to the method not by reference. Also, there is no way for two different methods to refer same instance of a data type. For such purpose, it is necessary to create object representation of primitive data types. These object representations of primitive data types are called as wrapper classes. In essence, these classes encapsulate or wrap the primitive data types within the class. All the wrapper classes are defined in package java.lang.

Table shows the wrapper classes of respective data types.

Data type	**Wrapper class**
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short
byte	Byte

Table 3.1 Listing of wrapper classes

The abstract class ‘Number’ is a super class that is implemented by the classes that wrap the numeric types: byte, short, int, long, float, and double. ‘Number’ has abstract methods that return the value of the object in each of the different number formats. That is, shortValue( ) returns the value as a short, floatValue( ) returns the value as a float, and so on. These methods are shown here:

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

The values returned by all these methods can be rounded. Number has six concrete subclasses that hold explicit values of each numeric type: Double, Float, Byte, Short, Integer, and Long. All these subclasses use the methods mentioned above.

Float and Double

Float and Double are wrapper classes for floating-point values of type float and double, respectively. The constructors for class Float are shown below:

```
Float(double num)
Float(float num)
Float(String str)
```

Float objects can be constructed with values of type float or double. They can also be constructed from the string representation of a floating-point number. It throws exception (NumberFormatException) when type is mismatched with the string value in 'str'.

The constructors for Double are shown below:

```
Double(double num)
Double(String str)
```

Double objects can be constructed with a double value or a string containing a floating-point value. Here also second constructor throws exception (NumberFormatException) when type is mismatched with the string value in 'str'.

Both Float and Double define the constants shown in table 3.2. All these constants are static and final. We can use them in the program for various pruposes.

Constant	**Meaning**
MAX_VALUE	Maximum positive value
MIN_VALUE	Minimum positive value
NaN	Not a number
POSITIVE_INFINITY	Positive infinity
NEGATIVE_INFINITY	Negative infinity
TYPE	The Class object for float or double

Table 3.2 Constants defined by Float and Double

Methods of Float and Double

int compareTo(Float f)

It compares the numerical value of the invoking object with that of ‘f’. It returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.

int compareTo(Object obj)

Operates identically to compareTo(Float) if ‘obj’ is of class Float.

boolean equals(Object FloatObj)

It returns true if the invoking Float object is equivalent to ‘FloatObj’. Otherwise, it returns false.

int hashCode()

This method returns the hash code for the invoking object.

boolean isInfinite()

It returns true if the invoking object contains an infinite value. Otherwise, it returns false.

static boolean isInfinite(float num)

It returns true if ‘num’ specifies an infinite value. Otherwise, it returns false. An infinite value can be assigned using any real number divided by 0.0 or directly by constant Float.POSITIVE_INFINITY and Float.NEGATIVE_INFINITY.

boolean isNaN( )

It returns true if the invoking object contains a value that is not a number. Otherwise, it returns false. A NaN is created when 0.0 is divided by 0.0 or using constant Float.NaN.

static boolean isNaN(float num)

It returns true if ‘num’ specifies a value that is not a number. Otherwise, it returns false.

static float parseFloat(String str)

It returns the float equivalent of the number contained in the string specified by ‘str’ using radix 10. It throws exception if number format of ‘str’ is not matched with float constant.

static Float valueOf(String str)

It returns the Float object that contains the value specified by the string in ‘str’.

The methods written above are especially for Float class. Same methods can be used for Double class also. Except we have to change float to double and Float to Double.

Program 3.8 demonstrates the use of Float and Double methods and constants.

```
//Use of Float and Double class
  class WrapperFloat
  {
      public static void main(String args[])
      {
          Float f1 = new Float(5.6f);
          Double d1 = new Double(0.0/0.0);
          Double d2 = new Double(Double.MAX_VALUE);
          float f = Float.parseFloat("6.3");
          System.out.println("Value : "+f);
          System.out.println("Max Double : "+d2);
          System.out.println("String :  "+f);
          if(d1.isNaN())
            System.out.println("d1 : "+d1);
      }
  }
```

Using Float and Double class

Output:

```
Value : 6.3
Max Double : 1.7976931348623157E308
String :  6.3
d1 : NaN
```

Byte, Short, Integer and Long

The Byte, Short, Integer, and Long are wrapper classes for byte, short, int, and long integer types, respectively. Their constructors are shown below:

```
Byte(byte num)
Byte(String str)

Short(short num)
Short(String str)

Integer(int num)
Integer(String str)

Long(long num)
Long(String str)
```

All these objects can be constructed from numeric values or from strings that contain valid whole number values. The constructors having string parameters might throw the exception (NumberFormatException) when the type mismatch occurs. The constants defined by all these classes are shown in table 3.3.

Constant	**Meaning**
MAX_VALUE	Maximum positive value
MIN_VALUE	Minimum positive value
TYPE	The Class object for byte, short, int, long

Constants defined by Byte, Short, Integer and Long

Methods of Byte, Short, Integer and Long

int compareTo(Byte b)

It compares the numerical value of the invoking object with that of ‘b’. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.

int compareTo(Object obj)

It operates identically to compareTo(Byte) if ‘obj’ is of class Byte. Otherwise, throws a ClassCastException.

static Byte decode(String str) throws NumberFormatException

It returns a Byte object that contains the value specified by the string in ‘str’.

boolean equals(Object ByteObj)

It returns true if the invoking Byte object is equivalent to ‘ByteObj’. Otherwise, it returns false.

int hashCode()

It returns the hash code for the invoking object.

static byte parseByte(String str) throws NumberFormatException

It returns the byte equivalent of the number contained in the string specified by ‘str’ using radix 10.

static byte parseByte(String str, int radix)

It returns the byte equivalent of the number contained in the string specified by ‘str’ using the specified ‘radix’. This method also throws NumberFormatException.

String toString( )

It returns a string that contains the decimal equivalent of the invoking object.

static String toString(byte num)
It returns a string that contains the decimal equivalent of ‘num’.

static Byte valueOf(String str) throws NumberFormatException
It returns a Byte object that contains the value specified by the string in ‘str’.

static Byte valueOf(String str, int radix)
It returns a Byte object that contains the value specified by the string in ‘str’ using the specified ‘radix’. This method also throws NumberFormatException.

All the above methods are given in terms of Byte class. These can also be applied to other numeric classes. Only we have to change Byte to Short or Integer or Long and byte to short or int or long. The ‘parseByte( )’ method is having the form ‘parseInt( )’ for Integer class.

Integer and Long class has added three useful methods:

static String toBinaryString(int num)
It returns a string that contains the binary equivalent of ‘num’.

static String toHexString(int num)
It returns a string that contains the hexadecimal equivalent of ‘num’.

static String toOctalString(int num)
It returns a string that contains the octal equivalent of ‘num’.

```
// Demonstration of Integer wrapper class
class WrapperInt
{
    public static void main(String args[])
    {
       Integer x = new Integer(141);
       byte b = x.byteValue();
       System.out.println("Byte value: "+b);

       int hash = x.hashCode();
       System.out.println("Hash code: "+hash);

       int val = Integer.parseInt("63");
       System.out.println("Parsed value: "+val);

       String s = x.toString();
       System.out.println("String value: "+s);
```

```
        Integer y = new Integer(Integer.valueOf("22"));
        System.out.println("Short value: "+y.shortValue());

        String bin = Integer.toBinaryString(59);
        String oct = Integer.toOctalString(59);
        String hex = Integer.toHexString(59);
        System.out.println("Binary of 59: "+bin);
        System.out.println("Octal of 59: "+oct);
        System.out.println("Hex of 59: "+hex);
     }
 }
```

Demonstrating the Integer wrapper class and methods

Output:

```
Byte value: -115
Hash code: 141
Parsed value: 63
String value: 141
Short value: 22
Binary of 59: 111011
Octal of 59: 73
Hex of 59: 3b
```

Character

Character is the wrapper class of data type char. It is having only one constructor:

```
Character(char ch)
```

Here, 'ch' is the character value for which the wrapper class is to be created. Like the numeric wrapper classes, the Character class contains a method to obtain the character value of wrapper class.

```
char charValue()
```

It returns the character which is stored in the Character class. Except this method all other methods of Character are static. Character also defines several constants. These are shown in table 3.4.

Constant	**Meaning**
MAX_RADIX	Largest radix
MIN_RADIX	Smallest radix
MAX_VALUE	Largest character value
MIN_VALUE	Smallest character value
TYPE	The Class object for char

Constants defined by Character

Methods of Character

Following are some important and useful methods of Character class.

static boolean isDigit(char ch)
It returns true if ‘ch’ is a digit. Otherwise, it returns false.

static boolean isLetter(char ch)
It returns true if ‘ch’ is a letter. Otherwise, it returns false.

static boolean isLetterOrDigit(char ch)
It returns true if ‘ch’ is a letter or a digit. Otherwise, it returns false.

static boolean isLowerCase(char ch)
It returns true if ‘ch’ is a lowercase letter. Otherwise, it returns false.

static boolean isSpaceChar(char ch)
It returns true if ‘ch’ is a Unicode space character. Otherwise, it returns false.

static boolean isTitleCase(char ch)
It returns true if ‘ch’ is a Unicode title-case character. Otherwise, it returns false.

static boolean isUpperCase(char ch)
It returns true if ‘ch’ is an uppercase letter. Otherwise, it returns false.

static boolean isWhitespace(char ch)
It returns true if ‘ch’ is whitespace. Otherwise, it returns false.

static char toLowerCase(char ch)
It returns lowercase equivalent of ‘ch’.

static char toTitleCase(char ch)
It returns titlecase equivalent of ‘ch’.

static char toUpperCase(char ch)
It returns uppercase equivalent of ‘ch’.

Program 3.10 demonstrates the use of Character class and its methods.

```
 // Demonstration of character wrapper class
```

```
class WrapperChar
{
    public static void main(String args[])
    {
        Character ch = new Character('X');
        System.out.print("Char value: ");
        System.out.println(ch.charValue());

        if(Character.isDigit('0'))
          System.out.println("0 is digit");

        if(Character.isLowerCase('R'))
          System.out.println("R is lower case");
        else
        {
          System.out.println("R is upper case");
          System.out.print("Lower case : ");
          System.out.println(Character.toLowerCase('R'));
        }
        if(Character.isWhitespace(' '))
          System.out.println("Character is white space");

    }
}
```

Demonstrating Character wrapper class

Output:

```
Char value: X
0 is digit
R is upper case
Lower case : r
Character is white space
```

Boolean

It is a wrapper class for boolean data type which is mostly useful when we pass boolean value to a method by reference. It contains the constants TRUE and FALSE, which define true and false Boolean objects. Boolean also defines the TYPE field, which is the Class object for boolean. Boolean defines following constructors:

```
Boolean(boolean boolValue)
Boolean(String boolString)
```

In the first version, ‘boolValue’ must be either true or false. In the second version, if ‘boolString’ contains the string “true” (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.

Boolean methods

Some of the Boolean methods are as below:

boolean booleanValue( )
It returns boolean equivalent value of the invoking object.

boolean equals(Object boolObj)
It returns true if the invoking object is equivalent to ‘boolObj’. Otherwise, it returns false.

int hashCode( )
It returns the hash code for the invoking object.

String toString( )
This method returns the string equivalent of the invoking object.

static String toString(boolean boolVal)
It returns the string equivalent of ‘boolVal’.

static Boolean valueOf(boolean boolVal)
It returns the Boolean equivalent of ‘boolVal’.

static Boolean valueOf(String boolString)
It returns true if ‘boolString’ contains the string “true” (in uppercase or lowercase). Otherwise, it returns false.

Program 3.11 shows the use of Boolean class methods.

```
//Demonstration of Boolean
class WrapperBool
{
    public static void main(String args[])
    {
        Boolean a = new Boolean("TRUE");
        Boolean b = new Boolean(false);
        System.out.println("First : "+a.booleanValue());
        System.out.println("Second : "+b.booleanValue());
        if(a.equals(b))
             System.out.println("Both equal");
        else
             System.out.println("Both not equal");
        System.out.println("String : "+a.toString());
        System.out.println("HashCode: "+b.hashCode());
    }
}
```

Program 3.11 Demonstration of Boolean wrapper class

Output:

```
First : true
Second : false
Both not equal
String : true
HashCode: 1237
```

Command line arguments

Sometimes we will want to pass information into a program when we run it. This is accomplished by passing command-line arguments to main( ). A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in the String array passed to main( ). For example, the following program displays all of the command-line arguments that it is called with:

```
// Using command line arguments
  class CommandLine
  {
       public static void main(String args[])
       {
            System.out.println("Arguments are : ");
            for(int i=0;i<args.length;i++)
            {
               System.out.print("args["+i+"] = ");
               System.out.println(args[i]);
            }
       }
  }
```

Program 3.12 Demonstration of command line arguments

This program will be compiled and they executed as

```
java CommandLine hi This is Java 6
```

We will get the output:

```
Arguments are :
args[0] = hi
args[1] = This
args[2] = is
args[3] = Java
args[4] = 6
```

Java has taken this concept from C/C++, but implemented effectively. Though there are always arguments given to main( ) method it is not necessary to pass arguments each time. It depends upon the programmer who wrote the program. We can eliminate the use of input statements for taking input such as Scanner class. Program 3.13 shows one of the usefulness of command line argument. By this we will get the idea about it.

```
//Addition of numbers using command line
  class Add
  {
      public static void main(String args[])
      {
          int add = 0;
          for(int i=0;i<args.length;i++)
          {
             add = add + Integer.parseInt(args[i]);
          }
          System.out.println("Addition : "+add);
      }
  }
```

Finding addition of numbers passed from command line

Running program as,

```
     java Add 45 55 24 85 632
```

Output:

```
Addition : 841
```
