3.1 ARRAYS

3.1.1 Introduction:

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

3.1.2 Declaration and Initialization of Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type.

Declaration:

The general form of a one-dimensional array declaration is
type var-name[ ];

Here, *type* declares the element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array. Thus, the element type for the array determines what type of data the array will hold.

For example, the following declares an array named **month_days** with the type "array of int":
int month_days[];

Although this declaration establishes the fact that **month_days** is an array variable, no array actually exists. In fact, the value of **month_days** is set to **null**, which represents an array with no value. To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory.

The general form of **new** as it applies to one-dimensional arrays appears as follows:
array-var = **new** *type* [*size*];

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero (for numeric types), **false** (for **boolean**), or **null**.

This example allocates a 12-element array of integers and links them to **month_days**:
month_days = new int[12];

After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Let's review: Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using **new**, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.

A program that creates an array of the number of days in each month:

```
// Demonstrate a one-dimensional array.
class Array {
public static void main(String args[]) {
int month_days[];
month_days = new int[12];
month_days[0] = 31;  month_days[1] = 28;  month_days[2] = 31;  month_days[3] = 30;
month_days[4] = 31;  month_days[5] = 30;  month_days[6] = 31;  month_days[7] = 31;
month_days[8] = 30;  month_days[9] = 31;  month_days[10] = 30; month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");
}
}
```

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is month_days[3] or 30.

Initialization:

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**.

For example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// An improved version of the previous program.
class AutoArray {
public static void main(String args[ ]) {
int month_days[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
                              or
int month_days[ ] = new int [ ] { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };  //you may use new
System.out.println("April has " + month_days[3] + " days.");
}
}
```

When you run this program, you see the same output as that generated by the previous version.

Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range. For example, the run-time system will check the value of each index into **month_days** to make sure that it is between 0 and 11 inclusive. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error.

Alternative Array Declaration Syntax:

Java also supports the following type of code for array declaration. In this declaration the type is followed by square bracket and the name or identifier of the array follows the square bracket.
type [ ] identifier; //one dimensional
type [ ] [ ] identifier; //two dimensional
Example: *int [ ] num;*
NOTE: Use of a square bracket before and after the identifier will create a multi-dimensional array.
int [ ] num [ ];
the above declaration is equivalent to
int [ ] [ ] num; ***or*** *int num [ ] [ ];*

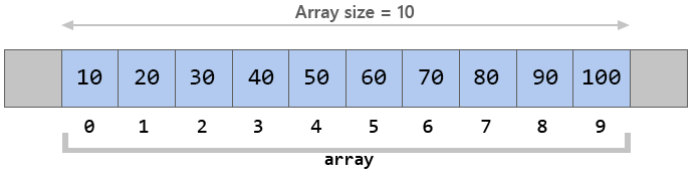
3.1.3 Storage of Array in Computer Memory

The operator new which is a keyword, allocates memory for storing the array elements.
For example with the following declaration

int [] num = new int [4];

The compiler allocates 4 memory spaces each equal to 4 bytes for storing the int type values of elements of array numbers. When an array is created as above, elements of the array are automatically initialized to 0 by the compiler.

int num [ ] = {10,20,30,40};



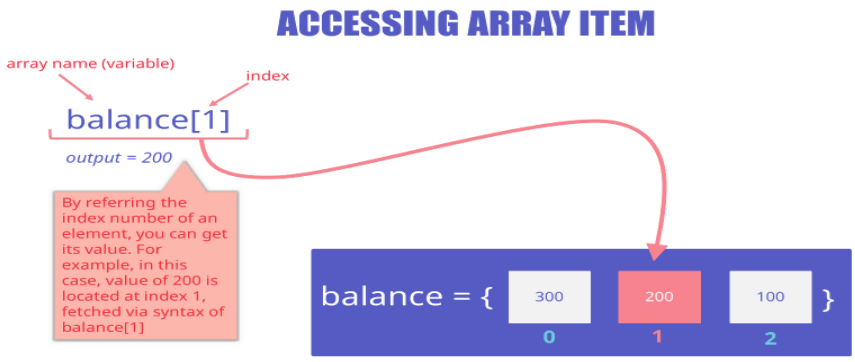
A two dimensional array may be declared and initialized as
int [ ] [ ] num = new int [ ] [ ] ({1,2,3},{4,5,6}};

or

int [ ] [ ] num = ({1,2,3},{4,5,6}};

3.1.4 Accessing Elements of Arrays

The individual member of an array may be accessed by its index value. The index value represents the place of element in the array.



3.1.5 Operations on Array Elements

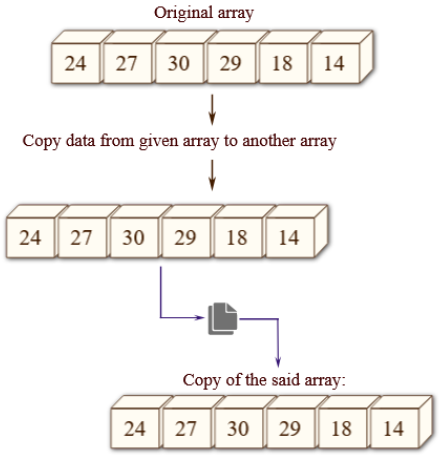
An array element is a variable of the type declared with the array. All the operations that are admissible for that type of a variable can be applied to an individual array element.

Similar to primitive types or objects of classes, the array may also be declared as a parameter of a method.

Acess_modifier type method_identifier (type array [], type other_parameter, ...)

3.1.6 Assigning Array to another Array

Unlike in C and C++ languages, in Java, an array may be assigned as a whole to another array of same data type. In this process, the second array identifier, in fact, becomes the reference to the assigned array. The second array is not a new array, instead only a second reference is created.




```
public class Test {
   public static void main(String[] args)
   {
      int a[] = { 1, 8, 3 };                    // Create an array b[] of same size as a[]
      int b[] = new int[a.length];       // Doesn't copy elements of a[] to b[], only makes b refer to same location
      b = a;
                  // Change to b[] will also reflect in a[] as 'a' and 'b' refer to same location.
      b[0]++;
       System.out.println("Contents of a[] ");
      for (int i = 0; i < a.length; i++)
         System.out.print(a[i] + " ");
       System.out.println("\n\nContents of b[] ");
      for (int i = 0; i < b.length; i++)
         System.out.print(b[i] + " ");
   }
}
```

Output:

Contents of a[ ]
2 8 3
Contents of b[ ]
2 8 3

3.1.7 Dynamic Change of Array Size

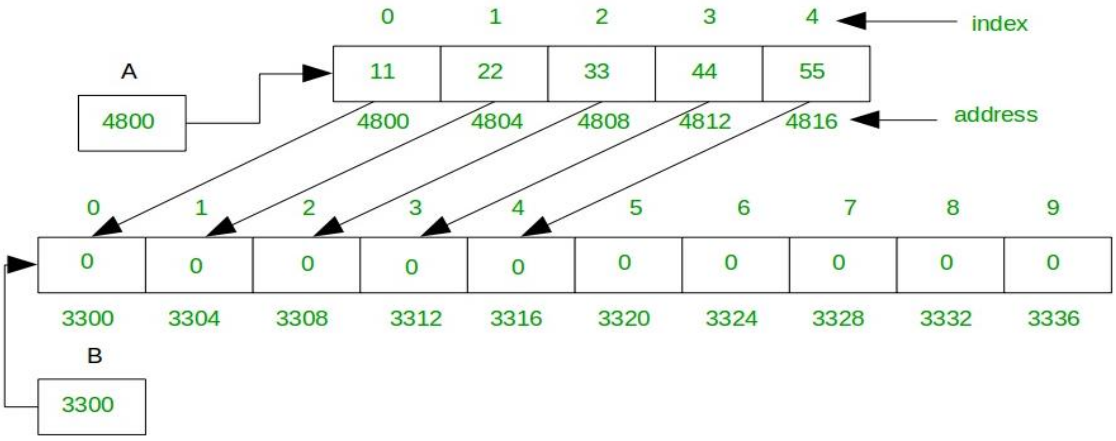
The number of elements (size) of the array may change during the execution of the program. This feature is unlike C and C++ wherein the array once declared is of fixed size, that is, the number of elements cannot be changed.

In Java, however, you may change the number of elements by dynamically retaining the array name. In this process, the old array is destroyed along with the values of elements.

Example:

int [ ] num = new int [5];

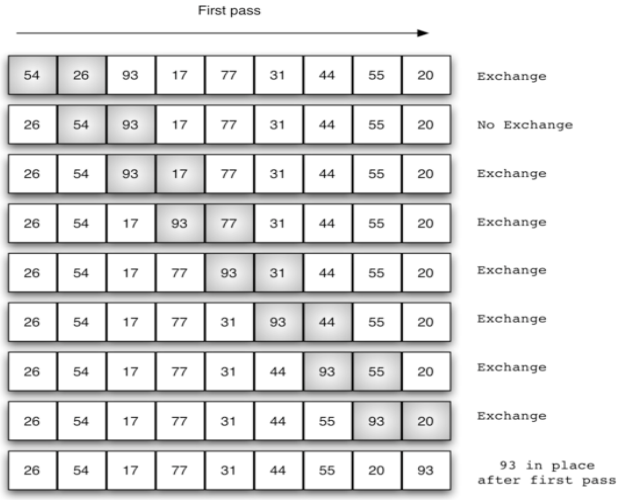
num = new int [10];



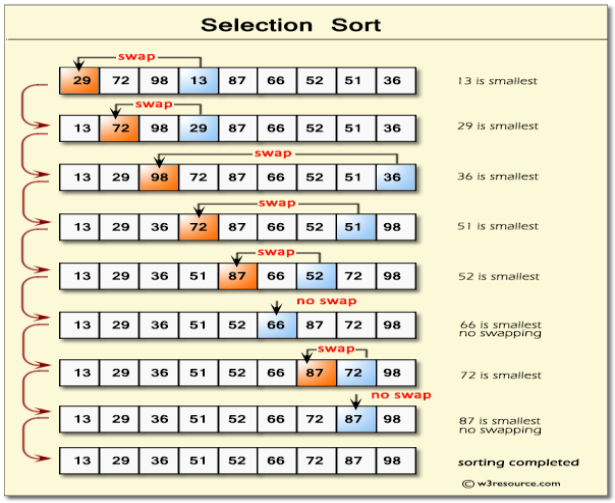
3.1.8 Sorting of Arrays

Sorting of arrays if often needed in many applications of arrays. For example, in the preparation of examination results, you may require to arrange the entries in order of grades acquired by students or in alphabetical order in dictionary style. The arrays may be sorted in ascending or descending order. Several methods are used for sorting the arrays that include the following:

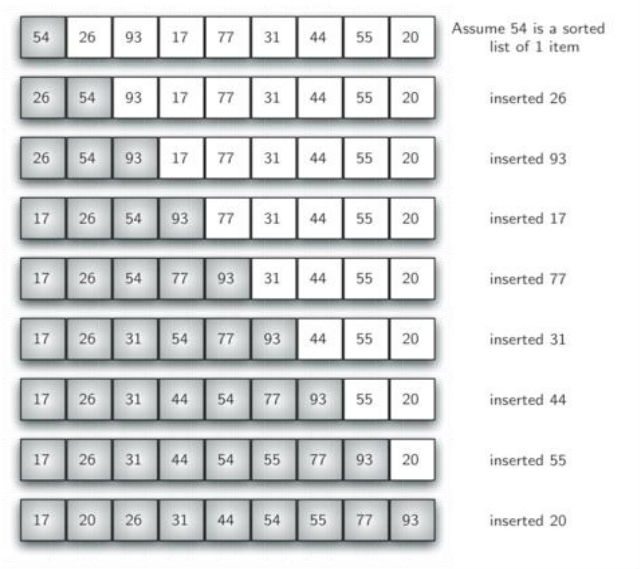
1. Bubble sort
2. Selection sort
3. Sorting by insertion method
4. Quick sort



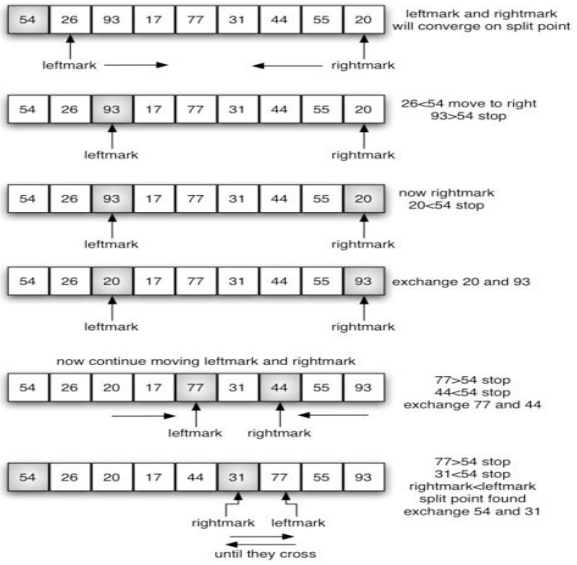
1. Bubble Sort



2. Selection Sort



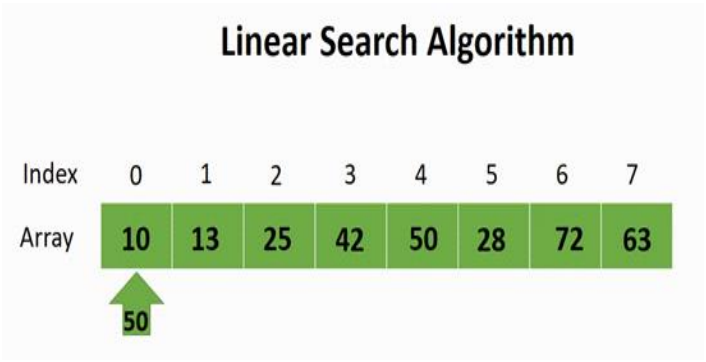
3. Insertion Sort



4. Quick Sort

3.1.9 Search for Values in Arrays

Searching an array for a value is often needed. Let us consider the example of searching for your name among the reserved seats in a retail reservation chart, air travel reservation chart, searching for a book in a library, and so on.
Two methods are employed: linear search and binary search for sorted arrays.



3.1.10 Class Arrays

The package java.util defines the class arrays with static methods for general processes that are carried out on arrays such as sorting an array for full length of the array or for part of an array, binary search of an array for the full array or part of array, for comparing two arrays if they are equal or not, for filling a part of the full array with elements having a specified value, and for copying an array to another array. The sort method of arrays class is based on quick sort technique.

The methods are applicable to all primitive types as well as to class objects.

The class arrays are declared as
public class Arrays extends Object

Methods of Class Arrays

The methods of class arrays are as follows:

- Sort
- Binary Search
- Equals
- Fill
- CopyOf
- AsList
- ToString
- deepToString
- hashCode

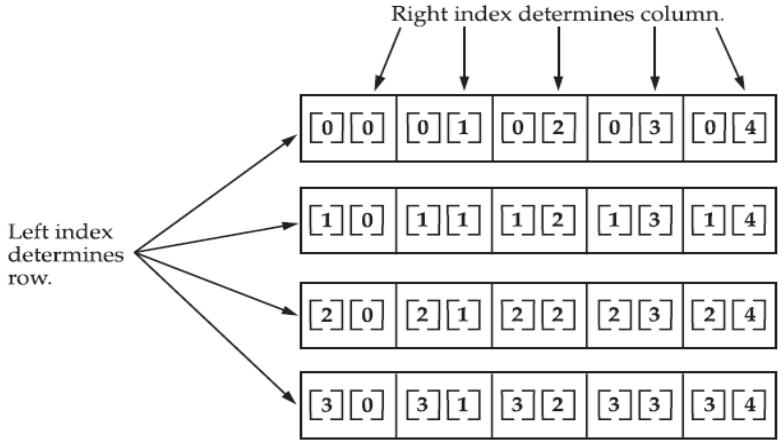
3.1.11 Two-dimensional or multi-dimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets.

For example, the following declares a twodimensional array variable called twoD:

int twoD[][] = new int[4][5];

This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int. Conceptually, this array will look like the one shown in below figure.



Given: int twoD [ ] [ ] = new int [4] [5] ;

A conceptual view of a 4 by 5, two-dimensional array

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```
// Demonstrate a two-dimensional array.
class TwoDArray {
        public static void main(String args[]) {
                int twoD[][]= new int[4][5];
                int i, j, k = 0;
                        for(i=0; i<4; i++)
                                for(j=0; j<5; j++) {
                                        twoD[i][j] = k;
```

```
                        k++;
                    }
                for(i=0; i<4; i++) {
                    for(j=0; j<5; j++)
                        System.out.print(twoD[i][j] + " ");
                    System.out.println();
                }
        }
}
```

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of twoD when it is declared. It allocates the second dimension manually.

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others. For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension. As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. For example, the following program creates a two-dimensional array in which the sizes of the second dimension are unequal:

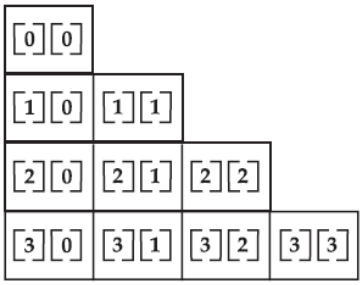
```
// Manually allocate differing size second dimensions.
class TwoDAgain {
    public static void main(String args[]) {
        int twoD[][] = new int[4][];
        twoD[0] = new int[1];
        twoD[1] = new int[2];
        twoD[2] = new int[3];
        twoD[3] = new int[4];
            int i, j, k = 0;
            for(i=0; i<4; i++)
                for(j=0; j<i+1; j++) {
                    twoD[i][j] = k;
                        k++;
                }
            for(i=0; i<4; i++) {
```

```
                    for(j=0; j<i+1; j++)
                        System.out.print(twoD[i][j] + " ");
                        System.out.println();
                    }
            }
}
```

This program generates the following output:

```
0
1 2
3 4 5
6 7 8 9
```

The array created by this program looks like this:



The use of uneven (or irregular) multidimensional arrays may not be appropriate for many applications, because it runs contrary to what people expect to find when a multidimensional array is encountered. However, irregular arrays can be used effectively in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution.

It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces. The following program creates a matrix where each element contains the product of the row and column indexes. Also notice that you can use expressions as well as literal values inside of array initializers.

```
// Initialize a two-dimensional array.
class Matrix {
    public static void main(String args[]) {
        double m[][] = {
                    { 0*0, 1*0, 2*0, 3*0 },
                    { 0*1, 1*1, 2*1, 3*1 },
                    { 0*2, 1*2, 2*2, 3*2 },
                    { 0*3, 1*3, 2*3, 3*3 }
                };
        int i, j;
            for(i=0; i<4; i++) {
                for(j=0; j<4; j++)
                    System.out.print(m[i][j] + " ");
                    System.out.println();
                }
    }
}
```

This program generates the following output:
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12
0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

type[ ] var-name;

Here, the square brackets follow the type specifier, and not the name of the array variable.
For example, the following two declarations are equivalent:

int al[] = new int[3];
int[] a2 = new int[3];

The following declarations are also equivalent:

char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

int[] nums, nums2, nums3; // create three arrays

creates three array variables of type int. It is the same as writing

int nums[], nums2[], nums3[]; // create three arrays

The alternative declaration form is also useful when specifying an array as a return type for a method.

3.1.12 Arrays as Vectors

Similar to arrays, vectors are another kind of data structure that is used for storing information. Using vector, we can implement a dynamic array. As we know, an array can be declared in the following way:
int marks[ ] = new int [7];

the basic difference between arrays and vectors is that vectors are dynamically allocated, where as arrays are static. The size of vector can be changed as and when required, but this is not true for arrays.

The vector class is contained in java.util package. Vector stores pointers to the objects and not objects themselves. The following are the vector constructors.

Vector vec = new Vector(5); // size of vector is

3.2 Inheritance

3.2.1 Introduction:

Inheritance is the technique which allows us to inherit the data members and methods from base class to derived class.

• Base class is one which always gives its features to derived classes.

• Derived class is one which always takes features from base class.

A Derived class is one which contains some of features of its own plus some of the data members from base class.

3.2.2 Process of inheritance

Syntax for INHERITING the features from base class to derived class:

```
class <clsname-2> extends <clsname-1>
{
 Variable declaration;
 Method definition;
};
```

Here, **clsname-1** and **clsname-2** represents derived class and base class respectively.
Extends is a keyword which is used for inheriting the data members and methods from base class to the derived class and it also improves functionality of derived class.

For example:

```
class c1;
{
 int a;
 void f1()
 {
 ……;
 }
};
class c2 extends c1
{
 int b;
 void f2()
 {
 ……;
 }
};
```

Whenever we inherit the base class members into derived class, when we creates an object of derived class, JVM always creates the memory space for base class members first and later memory space will be created for derived class members.

Example:

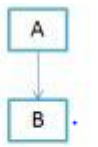
Write a JAVA program computes sum of two numbers using inheritance?

Answer:

```
class Bc
{
 int a;
};
class Dc extends Bc
{
        int b;
        void set (int x, int y)
        {
           a=x;
            b=y;
         }
        void sum ()
       {
            System.out.println ("SUM = "+(a+b));
        }
};
class InDemo
{
           public static void main (String k [])
          {

                      Dc do1=new Dc ();
                      do1.set (10,12);
                     do1.sum ();
           }
};
```

3.2.3 Types of Inheritances



(a) Single Inheritance

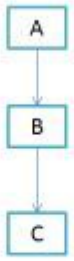
<u>Single Inheritance</u>: It means when a base class acquired the properties of super class

```
class Animal
{
void eat()
```

```
{
 System.out.println("eating...");
}
}
class Dog extends Animal
{
void bark()
{
System.out.println("barking...");
}
}
class TestInheritance
{
public static void main(String args[])
{
Dog d=new Dog();
d.bark();
d.eat();
}
}
```

In this example base class is Dog and super class is Animal:

Multilevel Inheritance:



(d) Multilevel Inheritance

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A

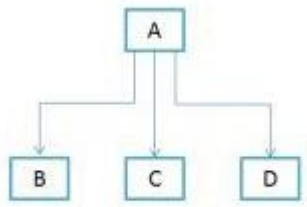
Example:

```
Class X
{
  public void methodX()
  {
    System.out.println("Class X method");
  }
}
```

```
Class Y extends X
{
public void methodY()
{
System.out.println("class Y method");
}
}
Class Z extends Y
{
  public void methodZ()
  {
    System.out.println("class Z method");
  }
  public static void main(String args[])
  {
    Z obj = new Z();
    obj.methodX(); //calling grand parent class method
    obj.methodY(); //calling parent class method
    obj.methodZ(); //calling local method
 }
}
```

Hierarchical Inheritance

In such kind of inheritance one class is inherited by many sub classes. In below example class B,C and D inherits the same class A. A is parent class (or base class) of B,C & D



(c) Hierarchical Inheritance

Example:

```
class A
{
  public void methodA()
  {
     System.out.println("method of Class A");
  }
}
class B extends A
{
  public void methodB()
  {
```

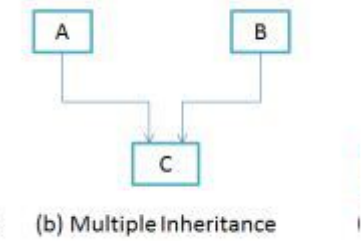
```
    System.out.println("method of Class B");
  }
}
class C extends A
{
 public void methodC()
 {
   System.out.println("method of Class C");
 }
}
class D extends A
{
 public void methodD()
 {
   System.out.println("method of Class D");
 }
}
class JavaExample
{
 public static void main(String args[])
 {
   B obj1 = new B();
   C obj2 = new C();
   D obj3 = new D();
 }
}
```

Output:

```
method of Class A
method of Class A
method of Class A
```

Multiple Inheritance:

Multiple Inheritance" refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes.

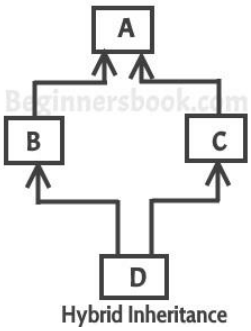


(b) Multiple Inheritance

Note: Multiple Inheritance is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.

Hybrid Inheritance in Java

A hybrid inheritance is a combination of more than one types of inheritance. For example when class A and B extends class C & another class D extends class A then this is a hybrid inheritance, because it is a combination of single and hierarchical inheritance.



Hybrid Inheritance

The diagram is just for the representation, since multiple inheritance is not possible in java

```
class C
{
  public void disp()
  {
        System.out.println("C");
  }
}

class A extends C
{
  public void disp()
  {
        System.out.println("A");
  }
}

class B extends C
{
  public void disp()
  {
        System.out.println("B");
  }

}

class D extends A
{
```

```
  public void disp()
  {
        System.out.println("D");
  }
  public static void main(String args[]){

        D obj = new D();
        obj.disp();
  }
}
```

This example is just to demonstrate the hybrid inheritance in Java. Although this example is meaningless, you would be able to see that how we have implemented two types of inheritance(single and hierarchical) together to form hybrid inheritance.

Class A and B extends class C → Hierarchical inheritance
Class D extends class A → Single inheritance

3.2.4 Inhibiting Inheritance of Class Using Final

The final keyword in java is used to restrict the user. The java final keyword can be used in many context.
Final can be:
1. variable
2. method
3. class

The main purpose of using a class being declared as final is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

Example:

```
final class Bike
{
}
 class Honda1 extends Bike
{
 void run()
{
System.out.println("running safely with 100kmph");
}
 public static void main(String args[])
{
 Honda1 honda= new Honda1();
 honda.run();
 }
}
```

Output:
Compile time error

Since class **Bike** is declared as final so the derived class **Honda** cannot extend **Bike**

3.2.5 Access Control and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.

```
class A
 {
int i; // public by default
private int j; // private to A
        void setij(int x, int y)
       {
                i = x;
                j = y;
        }
}
// A's j is not accessible here.
class B extends A
 {
        int total;
        void sum()
         {
                total = i + j; // ERROR, j is not accessible here
        }
}
class Access
{
        public static void main(String args[])
         {
                B subOb = new B();
                subOb.setij(10, 12);
                subOb.sum();
                System.out.println("Total is " + subOb.total);
        }
}
```

This program will not compile because the reference to j inside the sum( ) method of B causes an access violation. Since j is declared as private, it is only accessible by other members of its own class. Subclasses have no access to it.

3.2.6 Application of Keyword Super

Super keyword is used for differentiating the base class features with derived class features. Super keyword is placing an important role in three places.

- variable level
- method level
- constructor level

Super at variable level:
Whenever we inherit the base class members into derived class, there is a possibility that base class members are similar to derived class members.

In order to distinguish the base class members with derived class members in the derived class, the base class members will be preceded by a keyword super.

For example:

```
class Bc
{
 int a;
};
class Dc extends Bc
{
 int a;
 void set (int x, int y)
 {
 super.a=x;
 a=y; //by default 'a' is preceded with 'this.' since 'this.' represents current class
 }
 void sum ()
 {
 System.out.println ("SUM = "+(super.a+a));
 }
};
class InDemo1
{
 public static void main (String k [])
 {

 Dc do1=new Dc ();
 do1.set (20, 30);
 do1.sum ();
 }
};
```

Super at method level:
Whenever we inherit the base class methods into the derived class, there is a possibility that base class methods are similar to derived methods.

To differentiate the base class methods with derived class methods in the derived class, the base class methods must be preceded by a keyword super.

Syntax for super at method level: super. base class method name

For example:

```
class Bc
{
 void display ()
 {
 System.out.println ("BASE CLASS - DISPLAY...");
 }
};
class Dc extends Bc
{
 void display ()
 {
super.display (); //refers to base class display method
 System.out.println ("DERIVED CLASS- DISPLAY...");
 }
};
class InDemo2
{
public static void main (String k [])
 {
 Dc do1=new Dc ();
 do1.display ();
 }
};
```

3.2.7 Constructor Method and Inheritance

Super at constructor level:
super() can be used to invoke immediate parent class constructor.

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

```
  }
  void show()
  {
   System.out.println("i and j values are"+i+" "+j);
  }
}
class B extends A
{
 int k;
  B(int a, int b, int c)
  {
   super(a, b);//super class constructor
   k = c;
  }
   // display k – this overrides show() in A
  void show()
   {
   super.show();
   System.out.println("k: " + k);
   }

}

class Override
{

public static void main(String args[])
 {
  B subOb = new B(1, 2, 3);
  subOb.show(); // this calls show() in B

}
}
```

3.2.8 Method Overriding // Refer 2nd unit

3.2.9 Dynamic Method Dispatch // Refer 2nd unit

3.2.10 Abstract Classes:

In JAVA we have two types of classes. They are concrete classes and abstract classes.

• A concrete class is one which contains fully defined methods. Defined methods are also known as implemented or concrete methods. With respect to concrete class, we can create an object of that class directly.

- **An abstract class is one which contains some defined methods and some undefined methods. Undefined methods are also known as unimplemented or abstract methods. Abstract method is one which does not contain any definition.**

To make the method as abstract we have to use a keyword called abstract before the function declaration.

Syntax for ABSTRACT CLASS: abstract return_type method_name (parameters list);

```
// A Simple demonstration of abstract.
abstract class A
{
        abstract void callme();

// concrete methods are still allowed in abstract classes
        void callmetoo()
        {
                System.out.println("This is a concrete method.");
        }
}
class B extends A
 {
        void callme()
        {
                System.out.println("B's implementation of callme.");
        }
}
class AbstractDemo
 {
        public static void main(String args[])
        {
                B b = new B();
                b.callme();
                b.callmetoo();
        }
}
```

Notice that no objects of class A are declared in the program. As mentioned, it is not possible to instantiate an abstract class.

One other point: class A implements a concrete method called callmetoo( )

3.3 Interface in Java

3.3.1 Introduction

An interface in java is a blueprint of a class.
It has static constants and abstract methods. The interface in java is a mechanism to achieve abstraction. There can be only abstract methods in the java interface not method body.
It is used to achieve abstraction and multiple inheritance in Java.

- Using the keyword interface, you can fully abstract a class' interface from its implementation.
- That is, using interface, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body
- Variables can be declared inside of interface declarations.
- They are implicitly final and static, meaning they cannot be changed by the implementing class.
- They must also be initialized. All methods and variables are implicitly public.

3.3.2 Declaring an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

Here is an example of an interface definition. It declares a simple interface that contains one method called callback( ) that takes a single integer parameter.

```
interface Callback
 {
void callback(int param);
}
```

3.3.3 Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]] {
// class-body
}
```

If a class implements more than one interface, the interfaces are separated with a comma.

Here is a small example class that implements the Callback interface shown earlier.

```
class Client implements Callback
 {
// Implement Callback's interface
public void callback(int p)
 {
System.out.println("callback called with " + p);
}
}
```

Notice that callback( ) is declared using the public access specifier.

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of Client implements callback( ) and adds the method nonIfaceMeth( ):

```
class Client implements Callback
 {
// Implement Callback's interface
public void callback(int p)
 {
System.out.println("callback called with " + p);
}
void nonIfaceMeth()
 {
System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
}
}
```

Accessing Implementations Through Interface References:

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.

The following example calls the callback( ) method via an interface reference variable:

```
class TestIface
 {
public static void main(String args[])
 {
Callback c = new Client();
c.callback(42);
}
}
```

The output of this program is shown here:
callback called with 42

3.3.4 Multiple Interfaces

Multiple inheritance in java can be achieved through interfaces:
Example:

```
interface Printable
{
void print();
}

interface Showable
{
void show();
}

class A7 implements Printable,Showable
{
        public void print()
        {
                System.out.println("Hello");
        }
        public void show()
        {
                System.out.println("Welcome");
        }
        public static void main(String args[])
        {
                A7 obj = new A7();
                obj.print();
                obj.show();
        }
```

```
}
```

Output: Hello
Welcome

3.3.5 Nested Interfaces:

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

There are given some points that should be remembered by the java programmer.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitly.

Example:

```
interface Showable
{
  void show();
          interface Message
         {
           void msg();
          }
}
class TestNestedInterface1 implements Showable.Message
{
          public void msg()
         {
         System.out.println("Hello nested interface");
         }

 public static void main(String args[])
{
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

As you can see in the above example, we are acessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first.

3.3.6 Inheritance of Interfaces

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
// One interface can extend another.
interface A
 {
        void meth1();
        void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A
{
        void meth3();
}
// This class must implement all of A and B
class MyClass implements B
 {
        public void meth1()
        {
                System.out.println("Implement meth1().");
        }
        public void meth2()
         {
                System.out.println("Implement meth2().");
        }
        public void meth3()
        {
                System.out.println("Implement meth3().");
        }
}
class IFExtend
 {
        public static void main(String arg[])
         {
                MyClass ob = new MyClass();
                ob.meth1();
                ob.meth2();
                ob.meth3();
        }
}
```

3.3.7 Default methods in interfaces

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface

Example:

```
interface TestInterface
{
   // abstract method
   public void square(int a);

   // default method
   default void show()
   {
     System.out.println("Default Method Executed");
   }
}

class TestClass implements TestInterface
{
   // implementation of square abstract method
   public void square(int a)
   {
      System.out.println(a*a);
   }

   public static void main(String args[])
   {
      TestClass d = new TestClass();
      d.square(4);

      // default method executed
      d.show();
   }
}.
```

Output:
16
Default Method Executed

Default methods are also known as defender methods or virtual extension methods.

3.3.8 Static methods in interfaces

The interfaces can have static methods as well which is similar to static method of classes.

Example:

```
interface TestInterface
{
   // abstract method
```

```
    public void square (int a);

    // static method
    static void show()
    {
        System.out.println("Static Method Executed");
    }
}

class TestClass implements TestInterface
{
    // Implementation of square abstract method
    public void square (int a)
    {
        System.out.println(a*a);
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.square(4);

        // Static method executed
        TestInterface.show();
    }
}
```

Output:
16
Static Method Executed

3.3.9 Functional Interfaces

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.

A functional interface can have any number of default methods. Runnable, ActionListener, Comparable are some of the examples of functional interfaces.

3.3.10 FunctionalInterface annotation is used to ensure that the functional interface can't have more than one abstract method. In case more than one abstract methods are present, the compiler flags an 'Unexpected @FunctionalInterface annotation' message.

```
@FunctionalInterface
interface Square
{
    int calculate(int x);
}

class Test
{
```

```
    public static void main(String args[])
    {
        int a = 5;

        // lambda expression to define the calculate method
        Square s = (int x)->x*x;

        // parameter passed and return type must be
        // same as defined in the prototype
        int ans = s.calculate(a);
        System.out.println(ans);
    }
}
```

Output:
25