UNIT II

Classes: Classes, Objects, Methods, Parameters, Constructors, Garbage Collection, Access modifiers, Pass Objects and arguments, Method and Constructor Overloading, Understanding static, Nested and inner classes.
Inheritance – Basics, Member Access, Usage of Super, Multi level hierarchy, Method overriding, Abstract class, Final keyword.
Interfaces –Creating, Implementing, Using, Extending, and Nesting of interfaces.
Packages – Defining, Finding, Member Access, Importing.

CLASSES AND OBJECTS:

- ✓ A class is a *template* for an object, and an object is an *instance* of a class.

The General Form of a Class

- ✓ When a class is defined, its exact form and nature is declared by specifying the data that it contains and the code that operates on that data.
- ✓ A class is declared by use of the **class** keyword.

```
class class-name{
    // declare instance variables
    type var1;
    type var2;
    //.......
    type varN;
    // declare methods
    type method1( parameters){
    //body of method
    }
    type method2( parameters){
    //body of method
    }
    //.....
    type methodN( parameters){
    //body of method
    }
}
```

- ✓ The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. Collectively, the methods and variables defined within a class are called *members* of the class.
- ✓ Each instance of the class (that is, each object of the class) contains its own copy of these variables

Defining a class:

```
class Vehicle {
  int passengers; // number of passengers
  int fuelCap; // fuel capacity in gallons
  int mpg; // fuel consumption in miles per gallon
}
```

class declaration is only a type description: it does not create an actual object
To actually create a **Vehicle** object, use a statement like the following

```
Vehicle minivan = new Vehicle( );  // create a vehicle object called minivan
```

After this statement executes, **minivan** will be an instance of **Vehicle**. Thus, it will have "physical" reality.

- ✓ Thus, every **Vehicle** object will contain its own copies of the instance variables passengers, fuelCap, mpg. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable.

```
object.member;
minivan.fuelcap=16;
```

```
/* A program that uses the Vehicle class.
   Call this file VehicleDemo.java
*/
class Vehicle {
  int passengers; // number of passengers
  int fuelCap; // fuel capacity in gallons
  int mpg; // fuel consumption in miles per gallon
}

// This class declares an object of type Vehicle.
class VehicleDemo {
  public static void main(String[ ] args) {
    Vehicle minivan = new Vehicle();
    int range;
    // assign values to fields in minivan
    minivan.passengers = 7;
    minivan.fuelCap = 16;
    minivan.mpg = 21;
    // compute the range assuming a full tank of gas
    range = minivan.fuelCap * minivan.mpg;
    System.out.println("Minivan can carry " + minivan.passengers +
                       "with a range of " + range);
  }
}
```

- ✓ When this program is compiled, two **.class** files have been created. The Java compiler automatically puts each class into its own **.class** file.
- ✓ To run this program we must run VehicleDemo.class.

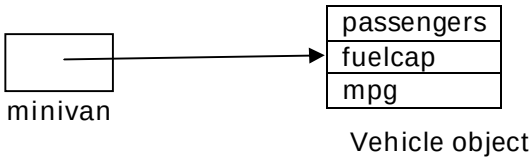
```
// This program creates two Vehicle objects.

class Vehicle {
  int passengers; // number of passengers
  int fuelCap; // fuel capacity in gallons
  int mpg; // fuel consumption in miles per gallon
}

// This class declares an object of type Vehicle.
class TwoVehicles {
  public static void main(String[] args) {
    Vehicle minivan = new Vehicle();
    Vehicle sportscar = new Vehicle();
    int range1, range2;

    // assign values to fields in minivan
    minivan.passengers = 7;
    minivan.fuelCap = 16;
    minivan.mpg = 21;

    // assign values to fields in sportscar
    sportscar.passengers = 2;
    sportscar.fuelCap = 14;
    sportscar.mpg = 12;

    // compute the ranges assuming a full tank of gas
    range1 = minivan.fuelCap * minivan.mpg;
    range2 = sportscar.fuelCap * sportscar.mpg;
    System.out.println("Minivan can carry " + minivan.passengers + " with a range of " + range1);
    System.out.println("Sportscar can carry " + sportscar.passengers +" with a range of " + range2);
  }
}
```

DECLARING OBJECTS

- ✓ Obtaining objects of a class is a two-step process.
- ✓ First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
- ✓ Second, you must acquire an actual, physical copy of the object and assign it to that variable. This can be done by using the **new** operator.
- ✓ The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.
- ✓ This reference is, more or less, the address in memory of the object allocated by **new**.

```
Vehicle minivan; / / declare reference to object
```

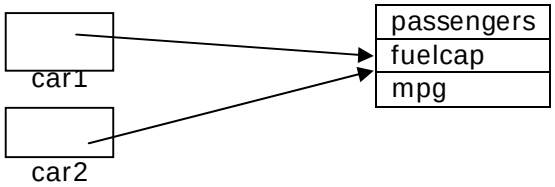

```
minivan = new Vehicle( ); / / allocate a Vehicle object
```



REFERENCE VARIABLES AND ASSIGNMENTS

- ✓ Object reference variables act differently when an assignment takes place.
- ✓ Consider the following fragment:

```
Vehicle car1= new Vehicle( );
Vehicle car2= car1;
```



After this fragment executes, **car1** and **car2** will both refer to the *same* object. The assignment of **car1** to **car2** did not allocate any memory or copy any part of the original object. It simply makes **car2** refer to the same object as does **car1**. Thus, any changes made to the object through **car2** will affect the object to which **car1** is referring, since they are the same object.

```
car1.mpg=26;
```

- ✓ When the following statements are executed display the same value 26

```
System.out.println(car1.mpg);
System.out.println(car2.mpg);
```

- ✓ Although **car1** and **car2** both refer to the same object, they are not linked in any other way.

```
Vehicle car1= new Vehicle( );
Vehicle car2= car1;
Vehicle car3= new Vehicle( );
car2=car3;
```

- ✓ After this statement executes **car2** refers to the same object as **car3**. The object referred to by **car1** is exchanged.

When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Methods

- ✓ A method contains the statements that define its actions.
  This is the general form of a method:

```
type name(parameter-list) {
    // body of method
}
```

- ✓ Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**.
- ✓ The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Adding a Method to the Vehicle Class

- ✓ Most of the time, methods are used to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementer to hide the specific layout of internal data structures behind cleaner method abstractions.

```
// Add range to Vehicle.
class Vehicle {
  int passengers; // number of passengers
  int fuelCap; // fuel capacity in gallons
  int mpg; // fuel consumption in miles per gallon
  // Display the range.
  void range( ) {
    System.out.println("Range is " + fuelCap * mpg);
  }
}

class AddMeth {
  public static void main(String[] args) {
    Vehicle minivan = new Vehicle();
    Vehicle sportscar = new Vehicle();
    int range1, range2;

    // assign values to fields in minivan
    minivan.passengers = 7;
    minivan.fuelCap = 16;
    minivan.mpg = 21;

    // assign values to fields in sportscar
    sportscar.passengers = 2;
    sportscar.fuelCap = 14;
    sportscar.mpg = 12;
    System.out.print("Minivan can carry " + minivan.passengers + ". ");
    minivan.range( ); // display range of minivan
    System.out.print("Sportscar can carry " + sportscar.passengers + ". ");
    sportscar.range( ); // display range of sportscar.
  }
}
```

- ✓ When a method is called, program control is transferred to the method. When the method terminates, control is transferred back to the caller, and execution resumes with the line of code following the call.
- ✓ When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator.

Returning from a method:

In general, two conditions can cause a method to return- first, when the method's closing brace is encountered. The second is when a return statement is executed. There are two forms of return – one for use in void methods and one for returning values.

Returning a value:

✓ Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

return *value*;

Here, *value* is the value returned.

```
// Use a return value.
class Vehicle {
  int passengers; // number of passengers
  int fuelCap; // fuel capacity in gallons
  int mpg; // fuel consumption in miles per gallon
  // Return the range.
  int range( ) {
    return mpg * fuelCap;
  }
}
class RetMeth {
  public static void main(String[ ] args) {
    Vehicle minivan = new Vehicle( );
    Vehicle sportscar = new Vehicle( );
    int range1, range2;
    // assign values to fields in minivan
    minivan.passengers = 7;
    minivan.fuelCap = 16;
    minivan.mpg = 21;
    // assign values to fields in sportscar
    sportscar.passengers = 2;
    sportscar.fuelCap = 14;
    sportscar.mpg = 12;
    // get the ranges
    range1 = minivan.range();
    range2 = sportscar.range();
    System.out.println("Minivan can carry " + minivan.passengers +"with range of " + range1 + " miles");
    System.out.println("Sportscar can carry " + sportscar.passengers + "with range of "+range2 + " miles");
  }
}
```

Using parameters:

✓ It is possible to pass one or more values to a method when the method is called.

✓ Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.

✓ There are two important things to understand about returning values:

• The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.

• The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

```
// A simple example that uses a parameter.
class ChkNum {
  // Return true if x is even.
  boolean isEven(int x) {
    if((x%2) == 0) return true;
    else return false;
  }
}
class ParmDemo {
  public static void main(String[ ] args) {
    ChkNum e = new ChkNum( );
    if(e.isEven(10)) System.out.println("10 is even.");
    if(e.isEven(9)) System.out.println("9 is even.");
    if(e.isEven(8)) System.out.println("8 is even.");
  }
}
```

A method can have more than one parameter. Simply declare each parameter, separating one from the next with a comma.

```
class Factor {
  // Return true if a is a factor of b.
  boolean isFactor(int a, int b) {
    if( (b % a) == 0) return true;
    else return false;
  }
}
class IsFact {
  public static void main(String[] args) {
    Factor x = new Factor( );
    if(x.isFactor(2, 20)) System.out.println("2 is factor");
    if(x.isFactor(3, 20)) System.out.println("this won't be displayed");
  }
}
```

Adding a parameterized method to Vehicle:

```
/*
   Add a parameterized method that computes the    fuel required for a given distance.
*/
class Vehicle {
  int passengers; // number of passengers
  int fuelCap; // fuel capacity in gallons
  int mpg; // fuel consumption in miles per gallon
  // Return the range.
  int range( ) {
    return mpg * fuelCap;
  }
  // Compute fuel needed for a given distance.
  double fuelNeeded(int miles) {
    return (double) miles / mpg;
  }
}
class CompFuel {
  public static void main(String[ ] args) {
    Vehicle minivan = new Vehicle( );
    Vehicle sportscar = new Vehicle( );
    double gallons;
    int dist = 252;
    // assign values to fields in minivan
    minivan.passengers = 7;
    minivan.fuelCap = 16;
    minivan.mpg = 21;
    // assign values to fields in sportscar
    sportscar.passengers = 2;
    sportscar.fuelCap = 14;
    sportscar.mpg = 12;
    gallons = minivan.fuelNeeded(dist);
    System.out.println("To go " + dist + " miles minivan needs " + gallons + " gallons of fuel.");
    gallons = sportscar.fuelNeeded(dist);
    System.out.println("To go " + dist + " miles sportscar needs " + gallons + " gallons of fuel.");
  }
}
```

CONSTRUCTORS:

- ✓ 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 is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.

```
// A simple constructor.
class MyClass {
  int x;
  MyClass( ) {
        x = 10;
  }
}
class ConsDemo {
  public static void main(String[ ] args) {
    MyClass t1 = new MyClass( );
    MyClass t2 = new MyClass( );
    System.out.println(t1.x + " " + t2.x);
  }
}
```

Parameterized Constructors

- ✓ Parameters are added to a constructor in the same way that they are added to a method: just declare them inside the parenthesis after the constructor's name.

```
// A parameterized constructor.
class MyClass {
  int x;
  MyClass(int i) {
    x = i;
  }
}
class ParmConsDemo {
  public static void main(String[ ] args) {
    MyClass t1 = new MyClass(10);
    MyClass t2 = new MyClass(88);
    System.out.println(t1.x + " " + t2.x);
  }
}
```

Adding a Constructor to a Vehicle calss

```
// Add a constructor.
class Vehicle {
  int passengers; // number of passengers
  int fuelCap;    // fuel capacity in gallons
  int mpg;        // fuel consumption in miles per gallon
  // This is a constructor for Vehicle.
  Vehicle(int p, int f, int m) {
    passengers = p;
    fuelCap = f;
    mpg = m;
  }
  // Return the range.
  int range( ) {
    return mpg * fuelCap;
  }
  // Compute fuel needed for a given distance.
  double fuelNeeded(int miles) {
    return (double) miles / mpg;
  }
}
```

```
class VehConsDemo {
  public static void main(String[] args) {
    / / construct complete vehicles
    Vehicle minivan = new Vehicle(7, 16, 21);
    Vehicle sportscar = new Vehicle(2, 14, 12);
    double gallons;
    int dist = 252;
    gallons = minivan.fuelNeeded(dist);
    System.out.println("To go " + dist + " miles minivan needs " +
                       gallons + " gallons of fuel.");
    gallons = sportscar.fuelNeeded(dist);
    System.out.println("To go " + dist + " miles sportscar needs " +
                       gallons + " gallons of fuel.");
  }
}
```

The this Keyword

- ✓ Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword.
- ✓ **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked.

```
class MyClass {
        int x;
        MyClass( int i) {
                this.x = i;
        }
       }
       class ConsDemo {
        public static void main(String[ ] args) {
          MyClass t1 = new MyClass(10);
          MyClass t2 = new MyClass(88);
          System.out.println(t1.x + " " + t2.x);
        }
       }
```

Output of the above code is

10 88

- ✓ **this** has some important uses. For example java syntax permits the name of a parameter or a local variable to be the same of an instance variable. When this happens, the local name hides the instance variable. The hidden instance variable can gain access by referring to it through **this.**

```
class MyClass {
        int x;
        MyClass( int x) {
                x = x;
        }
}
class ConsDemo {
        public static void main(String[ ] args) {
                        MyClass t1 = new MyClass(10);
                        MyClass t2 = new MyClass(88);
                        System.out.println(t1.x + " " + t2.x);
        }
}
```

Output is

0 0

If we use this key word we can gain access to the hidden instance variables

```
class MyClass {
        int x;
        MyClass( int x) {
                this.x = x;
        }
    }

class ConsDemo {
    public static void main(String[ ] args) {
                MyClass t1 = new MyClass(10);
                MyClass t2 = new MyClass(88);
                System.out.println(t1.x + " " + t2.x);
    }
}
```

O/P is 10 88

new OPERATOR REVISITED

- ✓ When you allocate an object, you use the following general form:
  class-var = new *classname* ( );
- ✓ Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called.
- ✓ When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class.
- ✓ The default constructor automatically initializes all instance variables to zero. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones.
- ✓ Once you define your own constructor, the default constructor is no longer used.

Garbage Collection

- ✓ Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator.
- ✓ Java takes a different approach; it handles deallocation automatically. The technique that accomplishes this is called *garbage collection*.
- ✓ When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- ✓ Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

The finalize( ) Method

- ✓ Sometimes an object will need to perform some action when it is destroyed.
- ✓ To handle such situations, Java provides a mechanism called *finalization*. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- ✓ To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed.
- ✓ The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

- ✓ Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class.

Access Modifiers:

- ✓ Encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control*
- ✓ How a member can be accessed is determined by the *access modifier* attached to its declaration. Java supplies a rich set of access modifiers.
- ✓ Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved.
- ✓ When a member of a class is modified by **public**, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.
- ✓ When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.
- ✓ An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

public int i;
private double j;
private int myMethod(int a, char b) { //…

- ✓ To understand the effects of public and private access, consider the following program:

/* This program demonstrates the difference between public and private. */

```
class Test {
        int a; // default access
        public int b; // public access
        private int c; // private access
        // methods to access c
        void setc(int i) { // set c's value
                c = i;
        }
        int getc() { // get c's value
                return c;
        }
}
class AccessTest {
        public static void main(String args[ ]) {
                Test ob = new Test();
                // These are OK, a and b may be accessed directly
                ob.a = 10;
                ob.b = 20;
                // This is not OK and will cause an error
                // ob.c = 100; // Error!
                // You must access c through its methods
                ob.setc(100); // OK
                System.out.println("a, b, and c: " + ob.a + " " +ob.b + " " + ob.getc());
        }
}
```

Pass objects to methods:

- ✓ It is possible to pass objects to a methods

```
// Objects can be passed to methods.
class Block {
  int a, b, c;
  int volume;
  Block(int i, int j, int k) {
    a = i;
    b = j;
    c = k;
    volume = a * b * c;
  }
  // Return true if ob defines same block.
  boolean sameBlock(Block ob) {
    if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
    else return false;
  }
  // Return true if ob has same volume.
  boolean sameVolume(Block ob) {
    if(ob.volume == volume) return true;
    else return false;
  }
}
class PassOb {
  public static void main(String[] args) {
    Block ob1 = new Block(10, 2, 5);
    Block ob2 = new Block(10, 2, 5);
    Block ob3 = new Block(4, 5, 5);
    System.out.println("ob1 same dimensions as ob2: " + ob1.sameBlock(ob2));
    System.out.println("ob1 same dimensions as ob3: " + ob1.sameBlock(ob3));
    System.out.println("ob1 same volume as ob3: " + ob1.sameVolume(ob3));
  }
}
```

How Arguments are passed:

- ✓ There are two ways to pass an argument to a subroutine.
- ✓ The first way is *call-by-value*. This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- ✓ The second way an argument can be passed is *call-by-reference*. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.
- ✓ When you pass a primitive type to a method, it is passed by value.

```
// Primitive types are passed by value.
class Test {
  /* This method causes no change to the arguments
     used in the call. */
  void noChange(int i, int j) {
    i = i + j;
    j = -j;
  }
}
class CallByValue {
  public static void main(String[] args) {
    Test ob = new Test();
    int a = 15, b = 20;
    System.out.println("a and b before call: " + a + " " + b);
    ob.noChange(a, b);
    System.out.println("a and b after call: " + a + " " + b);
  }
}
```

- ✓ When you pass an object to a method, objects are passed by call-by-reference.

```
// Objects are passed through their references.
class Test {
  int a, b;
  Test(int i, int j) {
    a = i;
    b = j;
  }
  /* Pass an object. Now, ob.a and ob.b in object
     used in the call will be changed. */
  void change(Test ob) {
    ob.a = ob.a + ob.b;
    ob.b = -ob.b;
  }
}
class PassObjRef {
  public static void main(String[] args) {
    Test ob = new Test(15, 20);
    System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
    ob.change(ob);
    System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
  }
}
```

Returning objects:

✓ A method can return any type of data, including class types that you create.

```
// Return a String object.
class ErrorMsg {
  String[] msgs = {  "Output Error", "Input Error", "Disk Full", "Index Out-Of-Bounds"};
  // Return the error message.
  String getErrorMsg(int i) {
    if(i >=0 & i < msgs.length)
      return msgs[i];
    else
      return "Invalid Error Code";
  }
}
class ErrMsgDemo {
  public static void main(String[] args) {
    ErrorMsg err = new ErrorMsg();
    System.out.println(err.getErrorMsg(2));
    System.out.println(err.getErrorMsg(19));
  }
}
```

O/P:

```
Disk Full
Invalid Error Code
```

✓ We can also return objects of classes that we create.

```
// Return a programmer-defined object.
class Err {
  String msg; // error message
  int severity; // code indicating severity of error
  Err(String m, int s) {
    msg = m;
    severity = s;
  }
}
class ErrorInfo {
  String[] msgs = {
    "Output Error",
    "Input Error",
    "Disk Full",
    "Index Out-Of-Bounds"
  };
  int[] howbad = { 3, 3, 2, 4 };
  Err getErrorInfo(int i) {
    if(i >= 0 & i < msgs.length)
      return new Err(msgs[i], howbad[i]);
    else
      return new Err("Invalid Error Code", 0);
  }
}
class ErrInfoDemo {
  public static void main(String[] args) {
    ErrorInfo err = new ErrorInfo( );
    Err e;
    e = err.getErrorInfo(2);
    System.out.println(e.msg + " severity: " + e.severity);
    e = err.getErrorInfo(19);
    System.out.println(e.msg + " severity: " + e.severity);
  }
}
```

METHOD OVERLOADING:

- ✓ In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is 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 that Java supports polymorphism.
- ✓ 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.

```
// Demonstrate method overloading.
class Overload {
  void ovlDemo() {
    System.out.println("No parameters");
  }
  // Overload ovlDemo for one integer parameter.
  void ovlDemo(int a) {
    System.out.println("One parameter: " + a);
  }
  // Overload ovlDemo for two integer parameters.
  int ovlDemo(int a, int b) {
    System.out.println("Two parameters: " + a + " " + b);
    return a + b;
  }
  // Overload ovlDemo for two double parameters.
  double ovlDemo(double a, double b) {
    System.out.println("Two double parameters: " + a + " " + b);
    return a + b;
  }
}
class OverloadDemo {
  public static void main(String[] args) {
    Overload ob = new Overload();
    int resI;
    double resD;
    // call all versions of ovlDemo()
    ob.ovlDemo();
    System.out.println();
    ob.ovlDemo(2);
    System.out.println();
    resI = ob.ovlDemo(4, 6);
    System.out.println("Result of ob.ovlDemo(4, 6): " +resI);
    System.out.println();
    resD = ob.ovlDemo(1.1, 2.32);
    System.out.println("Result of ob.ovlDemo(1.1, 2.32): " +
                       resD);
  }
}
```

O/P:

```
No parameters
One parameter: 2
Two parameters: 4 6
Result of ob.ovlDemo(4, 6): 10
Two double parameters: 1.1 2.32
Result of ob.ovlDemo(1.1, 2.32): 3.42
```

- ✓ The difference in their return types is insufficient for the purpose of overloading.

```
// one ovlDemo(int a) is ok
void ovlDemo(int a) {
   System.out.println("One parameter: " + a);
 }
 // Error. two  ovlDemo(int a) are not ok even though their return types are different
 int ovlDemo(int a) {
   System.out.println("One parameter: " + a);
   return a *a;
 }
```

- ✓ Java provides certain automatic type conversions. These conversions also apply to parameters of overloaded methods. For example consider the following:

```
/* Automatic type conversions can affect overloaded method resolution. */
class Overload2 {
  void f(int x) {
    System.out.println("Inside f(int): " + x);
  }

  void f(double x) {
    System.out.println("Inside f(double): " + x);
  }
}

class TypeConv {
  public static void main(String[ ] args) {
    Overload2 ob = new Overload2();
    int i = 10;
    double d = 10.1;

    byte b = 99;
    short s = 10;
    float f = 11.5F;

    ob.f(i); // calls ob.f(int)
    ob.f(d); // calls ob.f(double)

    ob.f(b); // calls ob.f(int) - type conversion
    ob.f(s); // calls ob.f(int) - type conversion
    ob.f(f); // calls ob.f(double) - type conversion
  }
}
```

O/P

```
Inside f(int) : 10
Inside f(double) : 10.1
Inside f(int) : 99
Inside f(int) : 10
Inside f(double) : 11.5
```

In the case of byte and short java automatically converts them to int. In the case of float the value is converted to double and f(double) is called.

The automatic type conversions apply only if there is no direct match between a parameter and an argument.

OVERLOADING CONSTRUCTORS:

✓ Like methods constructors can also be overloaded. This allows to construct objects in a variety of ways.

```
// Demonstrate an overloaded constructor.
class MyClass{
  int x;
  MyClass() {
    System.out.println("Inside MyClass().");
    x = 0;
  }
  MyClass(int i) {
    System.out.println("Inside MyClass(int).");
    x = i;
  }
  MyClass(double d) {
    System.out.println("Inside MyClass(double).");
    x = (int) d;
  }
  MyClass(int i, int j) {
    System.out.println("Inside MyClass(int, int).");
    x = i * j;
  }
}
class OverloadConsDemo {
  public static void main(String[] args) {
    MyClass t1 = new MyClass();
    MyClass t2 = new MyClass(88);
    MyClass t3 = new MyClass(17.23);
    MyClass t4 = new MyClass(2, 4);
    System.out.println("t1.x: " + t1.x);
    System.out.println("t2.x: " + t2.x);
    System.out.println("t3.x: " + t3.x);
    System.out.println("t4.x: " + t4.x);
  }
}
```

O/P:

```
Inside MyClass().
Inside MyClass(int).
Inside MyClass(double).
Inside MyClass(int, int).
t1.x: 0
t2.x: 88
t3.x: 17
t4.x: 8
```

// Initialize one object with another.

```
class Summation {
  int sum;
  // Construct from an int.
  Summation(int num) {
    sum = 0;
    for(int i=1; i <= num; i++)
      sum += i;
  }
  // Construct from another object.
  Summation(Summation ob) {
    sum = ob.sum;
  }
}
class SumDemo {
  public static void main(String[] args) {
    Summation s1 = new Summation(5);
    Summation s2 = new Summation(s1);
    System.out.println("s1.sum: " + s1.sum);
    System.out.println("s2.sum: " + s2.sum);
  }
}
```

O/P:

```
s1.sum: 15
s2.sum: 15
```

UNDERSTANDING static:

- ✓ Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- ✓ To create such a member, precede its declaration with the keyword **static**.
- ✓ When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- ✓ You 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.

static variables:

- ✓ 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** variable.

```
// Use a static variable.
class StaticDemo {
  int x; // a normal instance variable
  static int y; // a static variable
  // Return the sum of the instance variable x and the static variable y.
  int sum() {
    return x + y;
  }
}
class SDemo {
  public static void main(String[] args) {
    StaticDemo ob1 = new StaticDemo();
    StaticDemo ob2 = new StaticDemo();
    // Each object has its own copy of an instance variable.
    ob1.x = 10;
    ob2.x = 20;
    System.out.println("ob1.x: " + ob1.x + "\nob2.x: " + ob2.x);
    System.out.println();
    StaticDemo.y = 19;
    System.out.println("ob1.sum(): " + ob1.sum());
    System.out.println("ob2.sum(): " + ob2.sum());
    System.out.println();
    StaticDemo.y = 100;
    System.out.println("ob1.sum(): " + ob1.sum());
    System.out.println("ob2.sum(): " + ob2.sum());
    System.out.println();
  }
}
```

```
O/P:
ob1.x: 10
ob2.x: 20

ob1.sum(): 29
ob2.sum(): 39

ob1.sum(): 110
ob2.sum(): 120
```

static Methods:

- ✓ Methods declared static are, essentially, global methods. They are called independently of any object. Instead a static method is called through its class name.
- ✓ Methods declared as **static** have several restrictions:
  - They can only directly call other **static** methods.
  - They can only directly access **static** data.
  - They cannot refer to **this** or **super** in any way.

```
// Use a static method.
class StaticMeth {
  static int val = 1024; // a static variable
  // A static method.
  static int valDiv2() {
    return val/2;
  }
}
class SDemo2 {
  public static void main(String[] args) {
    System.out.println("val is " + StaticMeth.val);
    System.out.println("StaticMeth.valDiv2(): " +StaticMeth.valDiv2());
    StaticMeth.val = 4;
    System.out.println("val is " + StaticMeth.val);
    System.out.println("StaticMeth.valDiv2(): " + StaticMeth.valDiv2());
  }
}
```

O/P:

```
val is 1024
StaticMeth.valDiv2(): 512
val is 4
StaticMeth.valDiv2(): 2
```

static Blocks:

- ✓ A static block is executed when the class is first loaded. Thus, it is executed before the class can be used for any other purpose.

```
// Use a static block
class StaticBlock {
  static double rootOf2;
  static double rootOf3;
  static {
    System.out.println("Inside static block.");
    rootOf2 = Math.sqrt(2.0);
    rootOf3 = Math.sqrt(3.0);
  }
  StaticBlock(String msg) {
    System.out.println(msg);
  }
}
class SDemo3 {
  public static void main(String[] args) {
    StaticBlock ob = new StaticBlock("Inside Constructor");
    System.out.println("Square root of 2 is " +StaticBlock.rootOf2);
    System.out.println("Square root of 3 is " +StaticBlock.rootOf3);
  }
}
```

O/P:

```
Inside static block.
Inside Constructor
Square root of 2 is 1.4142135623730951
Square root of 3 is 1.7320508075688772
```

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 does not exist independently of A.
- ✓ A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.
- ✓ A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.
- ✓ There are two types of nested classes: *static* and *non-static*.
- ✓ A static nested class is one that has the **static** modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly.
- ✓ The most important type of nested class is the *inner* class. An inner class is a non-static nested class.

```
// Use an inner class.
class Outer {
  int[] nums;
  Outer(int[] n) {
    nums = n;
  }
  void analyze() {
    Inner inOb = new Inner();
    System.out.println("Minimum: " + inOb.min());
    System.out.println("Maximum: " + inOb.max());
    System.out.println("Average: " + inOb.avg());
  }
  // This is an inner class.
  class Inner {
    // Return the minimum value.
    int min() {
      int m = nums[0];
      for(int i=1; i < nums.length; i++)
        if(nums[i] < m) m = nums[i];
      return m;
    }
    // Return the maximum value.
    int max() {
      int m = nums[0];
      for(int i=1; i < nums.length; i++)
        if(nums[i] > m) m = nums[i];
      return m;
    }
    // Return the average.
    int avg() {
      int a = 0;
      for(int i=0; i < nums.length; i++)
        a += nums[i];
      return a / nums.length;
    }
  }
}
class NestedClassDemo {
  public static void main(String[] args) {
    int[] x = { 3, 2, 1, 5, 6, 9, 7, 8 };
    Outer outOb = new Outer(x);
    outOb.analyze();
  }
}
```

O/P:
Minimum: 1
Maximum: 9
Average: 5

INHERITANCE:

Basics:

- ✓ Java supports inheritance by allowing one class to incorporate another class into its declaration. This is done by using **extends** keyword.
- ✓ In java a class that is inherited is called a superclass. The class that does the inheriting is called a subclass.
- ✓ The general form of a **class** declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {
    / / body of class
}
```

- ✓ We can only specify one superclass for any subclass that we create. Java does not support the inheritance of multiple superclasses into a single subclass.
- ✓ A major advantage of inheritance is that once we had created a superclass that defines the atteibutes common to a set of objects, it can be used to create any number of more specific subclasses.
- ✓ The following program creates a superclass called TwoDShape and subclass called Triangle

```
// A class for two-dimensional objects.
class TwoDShape {
  double width;
  double height;
  void showDim() {
    System.out.println("Width and height are " + width + " and " + height);
  }
}
// A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
  String style;
  double area() {
    return width * height / 2;
  }
  void showStyle() {
    System.out.println("Triangle is " + style);
  }
}
class Shapes {
  public static void main(String[] args) {
    Triangle t1 = new Triangle();
    Triangle t2 = new Triangle();
    t1.width = 4.0;
    t1.height = 4.0;
    t1.style = "filled";
    t2.width = 8.0;
    t2.height = 12.0;
    t2.style = "outlined";
    System.out.println("Info for t1: ");
    t1.showStyle();
    t1.showDim();
    System.out.println("Area is " + t1.area());
    System.out.println();
    System.out.println("Info for t2: ");
    t2.showStyle();
    t2.showDim();
    System.out.println("Area is " + t2.area());
  }
}
```

```
O/P:
Info for t1:
Triangle is filled
Width and height are 4.0 and 4.0
Area is 8.0

Info for t2:
Triangle is outlined
Width and height are 8.0 and 12.0
Area is 48.0
```

MEMBER ACCESS AND INHERITANCE:

- ✓ Inheriting a class does not overrule the private access restriction. Thus even though a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

```
// Private members of a superclass are not accessible by a subclass.
// This example will not compile.
// A class for two-dimensional objects.
class TwoDShape {
  private double width; // these are
  private double height; // now private
  void showDim() {
    System.out.println("Width and height are " + width + " and " + height);
  }
}
// A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
  String style;
  double area() {
    return width * height / 2; // Error! can't access
  }
  void showStyle() {
    System.out.println("Triangle is " + style);
  }
}
```

- ✓ The Triangle class will not compile because the reference to width and height inside the area() method causes an access violation. Since width and height are declared private in TwoDShape, they are accessible only by the other members of TwoDShape.
- ✓ To access private members of superclass we can use accessor methods.

USAGE OF super:

- ✓ Both the superclass and subclass have their own constructors.
- ✓ Constructors for the superclass construct the superclass portion of the object, and the constructor for the subclass constructs the subclass part.
- ✓ When both the superclass and the subclass define constructors, the process is a bit complicated because both the superclass and subclass constructors must be executed. In this case we need to use **super** keyword.
- ✓ **super** has two general forms. The first calls the superclass constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

- ✓ A subclass can call a constructor defined by its superclass by use of the following form of **super**:

```
super(arg-list);
```

- ✓ **super( )** must always be the first statement executed inside a subclass constructor.

// Add constructors to TwoDShape.

```
class TwoDShape {
  private double width;
  private double height;
  // Parameterized constructor.
  TwoDShape(double w, double h) {
    width = w;      height = h;
  }
  // Accessor methods for width and height.
  double getWidth() { return width; }
  double getHeight() { return height; }
  void setWidth(double w) { width = w; }
  void setHeight(double h) { height = h; }
```

```
  void showDim() {
    System.out.println("Width and height are " + width + " and " + height);
  }
}
// A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
  private String style;
  Triangle(String s, double w, double h) {
    super(w, h); // call superclass constructor
    style = s;
  }
  double area() {
    return getWidth() * getHeight() / 2;
  }
  void showStyle() {
    System.out.println("Triangle is " + style);
  }
}
class Shapes4 {
  public static void main(String[] args) {
    Triangle t1 = new Triangle("filled", 4.0, 4.0);
    Triangle t2 = new Triangle("outlined", 8.0, 12.0);
    System.out.println("Info for t1: ");
    t1.showStyle();
    t1.showDim();
    System.out.println("Area is " + t1.area());
    System.out.println();
    System.out.println("Info for t2: ");
    t2.showStyle();
    t2.showDim();
    System.out.println("Area is " + t2.area());
  }
}
```

Using super to Access Superclass Members:

- ✓ The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

  super. *member*

- ✓ Here, *member* can be either a method or an instance variable

// Using super to overcome name hiding.

```
class A {
  int i;
}
// Create a subclass by extending class A.
class B extends A {
  int i; // this i hides the i in A
  B(int a, int b) {
    super.i = a; // i in A
    i = b; // i in B
  }
  void show() {
    System.out.println("i in superclass: " + super.i);
    System.out.println("i in subclass: " + i);
  }
}
class UseSuper {
  public static void main(String[] args) {
    B subOb = new B(1, 2);
    subOb.show();
  }
}
```

Creating a Multilevel Hierarchy

- ✓ We can build hierarchies that contain as many layers of inheritance. It is perfectly acceptable to use a subclass as a superclass of another.
- ✓ For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**.
- ✓ To see how a multilevel hierarchy can be useful, consider the following program.

```
// A multilevel hierarchy.
class TwoDShape {
  private double width;
  private double height;
  // A default constructor.
  TwoDShape() {
    width = height = 0.0;
  }
  // Parameterized constructor.
  TwoDShape(double w, double h) {
    width = w;
    height = h;
  }
  // Construct object with equal width and height.
  TwoDShape(double x) {
    width = height = x;
  }
  // Accessor methods for width and height.
  double getWidth() { return width; }
  double getHeight() { return height; }
  void setWidth(double w) { width = w; }
  void setHeight(double h) { height = h; }
  void showDim() {
    System.out.println("Width and height are " +
                       width + " and " + height);
  }
}
// Extend TwoDShape.
class Triangle extends TwoDShape {
  private String style;
  // A default constructor.
  Triangle() {
    super();
    style = "none";
  }
  Triangle(String s, double w, double h) {
    super(w, h); // call superclass constructor
    style = s;
  }
  // One argument constructor.
  Triangle(double x) {
    super(x); // call superclass constructor
    // default style to filled
    style = "filled";
  }
  double area() {
    return getWidth() * getHeight() / 2;
  }
  void showStyle() {
    System.out.println("Triangle is " + style);
  }
}
```

```
// Extend Triangle.
class ColorTriangle extends Triangle {
  private String color;
  ColorTriangle(String c, String s, double w, double h) {
    super(s, w, h);
    color = c;
  }
  String getColor() { return color; }
  void showColor() {
    System.out.println("Color is " + color);
  }
}
class Shapes6 {
  public static void main(String[] args) {
    ColorTriangle t1 =new ColorTriangle("Blue", "outlined", 8.0, 12.0);
    ColorTriangle t2 =new ColorTriangle("Red", "filled", 2.0, 2.0);
    System.out.println("Info for t1: ");
    t1.showStyle();
    t1.showDim();
    t1.showColor();
    System.out.println("Area is " + t1.area());
    System.out.println();
    System.out.println("Info for t2: ");
    t2.showStyle();
    t2.showDim();
    t2.showColor();
    System.out.println("Area is " + t2.area());
  }
}
```

When Constructors Are Called

- ✓ When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called?
- ✓ In a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- ✓ Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used. If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed.
- ✓ The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are executed.
class A {
  A() {
    System.out.println("Constructing A.");
  }
}
class B extends A {
  B() {
    System.out.println("Constructing B.");
  }
}
class C extends B {
  C() {
    System.out.println("Constructing C.");
  }
}
class OrderOfConstruction {
  public static void main(String[] args) {
    C c = new C();
  }
}
```

Superclass references and subclass objects:

- ✓ A reference variable for one class type cannot normally refer to an object of another class type.

```
// This will not compile.
class X {
  int a;
  X(int i) { a = i; }
}
class Y {
  int a;
  Y(int i) { a = i; }
}
class IncompatibleRef {
  public static void main(String[] args) {
    X x = new X(10);
    X x2;
    Y y = new Y(5);
    x2 = x; // OK, both of same type
    x2 = y; // Error, not of same type
  }
}
```

- ✓ A reference variable of a superclass can be assigned a reference to an object of any subclass derived from that superclass.

```
// A superclass reference can refer to a subclass object.
class X {
  int a;
  X(int i) { a = i; }
}
class Y extends X {
  int b;
  Y(int i, int j) {
    super(j);
    b = i;
  }
}
class SupSubRef {
  public static void main(String[] args) {
    X x = new X(10);
    X x2;
    Y y = new Y(5, 6);
    x2 = x; // OK, both of same type
    System.out.println("x2.a: " + x2.a);
    x2 = y; // still OK because y is derived from X
    System.out.println("x2.a: " + x2.a);
      // X references know only about X members
    x2.a = 19; // OK
   //  x2.b = 27; // Error, X doesn't have a b member
  }
}
```

- ✓ When a reference to a subclass object is assigned to a superclass reference variable we can only access to those parts of the objet defined by the superclass.
- ✓ When constructors are called in a class hierarchy, subclass references can be assigned to a superclass variable.

METHOD OVERRIDING:

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

```
class A1{
    int i, j;
    A1(int a, int b) {
            i = a;
            j = b;
    }
  // display i and j
     void show(){
            System.out.println("i and j: " + i + " " + j);
    }
}
class B1 extends A1 {
    int k;
    B1(int a, int b, int c) {
            super(a, b);
            k = c;
    }
    // display k - this overrides show() in A
    void show() {
            System.out.println("k: " + k);
    }
}
class MetOver{
  public static void main(String[] args) {
      B1 subOb = new B1(1,2,3);
      subOb.show(); // this calls show() in B
  }
}
O/P:
k: 3
```

- ✓ When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used.
- ✓ If you wish to access the superclass version of an overridden method, you can do so by using **super**.

```
class B extends A {
  int k;
  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
  void show() {
    super.show(); // this calls A's show()
    System.out.println("k: " + k);
  }
}
O/P:
i and j: 1 2
k: 3
```

- ✓ Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

```
/* Methods with differing signatures are overloaded and not overridden. */
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }
  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
// Create a subclass by extending class A.
class B extends A {
  int k;
  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
  // overload show()
  void show(String msg) {
    System.out.println(msg + k);
  }
}
class Overload {
  public static void main(String[] args) {
    B subOb = new B(1, 2, 3);
    subOb.show("This is k: "); // this calls show() in B
    subOb.show(); // this calls show() in A
  }
}
```

O/P:

```
this is k: 3
i and j: 1 2
```

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 how Java implements run-time polymorphism.
- ✓ When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- ✓ When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

```
// Demonstrate dynamic method dispatch.
class Sup {
  void who() {
    System.out.println("who() in Sup");
  }
}
class Sub1 extends Sup {
  void who() {
    System.out.println("who() in Sub1");
  }
}
```

```
class Sub2 extends Sup {
  void who() {
    System.out.println("who() in Sub2");
  }
}
class DynDispDemo {
  public static void main(String[] args) {
    Sup superOb = new Sup();
    Sub1 subOb1 = new Sub1();
    Sub2 subOb2 = new Sub2();
    Sup supRef;
    supRef = superOb;
    supRef.who();
    supRef = subOb1;
    supRef.who();
    supRef = subOb2;
    supRef.who();
  }
}
```

O/P:

who() in Sup
who() in Sub1
who() in Sub2

Why Overridden Methods?

- ✓ Polymorphism is essential to object-oriented programming for one reason: it 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.
- ✓ Dynamic, run-time polymorphism is one of the most powerful mechanisms that object oriented design brings to bear on 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.

USING ABSTRACT CLASSES:

- ✓ A class which contains the **abstract** keyword in its declaration is known as abstract class.
  - ✓ Abstract classes may or may not contain *abstract methods* ie., methods without body (public void get();)
  - ✓ But, if a class has at least one abstract method, then the class **must** be declared abstract.
  - ✓ If a class is declared abstract it cannot be instantiated.
  - ✓ To use an abstract class you have to inherit it from another class, provide implementations to the abstract methods in it.
  - ✓ If you inherit an abstract class you have to provide implementations to all the abstract methods in it.

Abstract Methods:

- ✓ If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract.
  - ✓ **abstract** keyword is used to declare the method as abstract.
  - ✓ You have to place the **abstract** keyword before the method name in the method declaration.
  - ✓ An abstract method contains a method signature, but no method body.
  - ✓ Instead of curly braces an abstract method will have a semi colon (;) at the end.
- ✓
- ✓

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

Using final

- ✓ To prevent a method from being overridden or a class from being inherited by using the keyword **final.**

final prevents overriding:

- ✓ To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

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

Using final to Prevent Inheritance

- ✓ To prevent a class from being inherited, precede the class declaration with **final**.
- ✓ Declaring a class as **final** implicitly declares all of its methods as **final**, too.
- ✓ It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

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

final with data members:

- ✓ **final** can also be applied to member variables . If a class variable's name precede with **final**, its value cannot be changed throughout the lifetime of the program.

INTERFACES:

- ✓ Using the keyword **interface**, you can fully abstract a class' interface from its implementation.
- ✓ Java Interface also **represents IS-A relationship**.
- ✓ Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- ✓ Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.
- ✓ To implement an interface, a class must create the complete set of methods defined by the interface.
- ✓ By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- ✓ Interfaces are designed to support dynamic method resolution at run time.
- ✓ **Defining an Interface**
- ✓ An interface is defied much like a class. This is a simplified general form of an interface:

```
access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    //...
    return-type method-nameN(parameter-list);
}
```

- ✓ *The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.*
- ✓ An interface is different from a class in several ways, including:
  - You cannot instantiate an interface.
  - An interface does not contain any constructors.
  - All of the methods in an interface are abstract.
  - An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
  - An interface is not extended by a class; it is implemented by a class.
  - An interface can extend multiple interfaces.

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 is:

```
class classname extends superclass implements interface {
    // class-body
}
```

- ✓ If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public**.
- ✓ Differences between abstract class and interface that are given below.

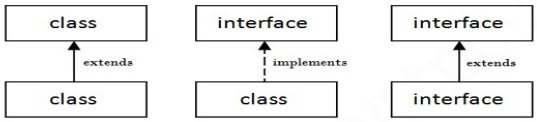
Abstract class	Interface
1) Abstract class can **have abstract and non-abstract**methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

- ✓ Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Understanding relationship between classes and interfaces

- ✓ As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.




```
public interface Series {
  int getNext(); // return next number in series
  void reset(); // restart
  void setStart(int x); // set starting value
}
// Implement Series.
class ByTwos implements Series {
  int start;
  int val;
  ByTwos() {
    start = 0;
    val = 0;
  }
  // Implement the methods specified by Series.
  public int getNext() {
    val += 2;
    return val;
  }
  public void reset() {
    val = start;
  }
  public void setStart(int x) {
    start = x;
    val = x;
  }
}
class SeriesDemo {
  public static void main(String[] args) {
    ByTwos ob = new ByTwos();
    for(int i=0; i < 5; i++)
      System.out.println("Next value is " + ob.getNext());
    System.out.println("\nResetting");
    ob.reset();
    for(int i=0; i < 5; i++)
      System.out.println("Next value is " + ob.getNext());
    System.out.println("\nStarting at 100");
    ob.setStart(100);
    for(int i=0; i < 5; i++)
      System.out.println("Next value is " + ob.getNext());
  }
}
```

O/P:

```
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10
```

```
Resetting
Next value is 2
Next value is 4
Next value is 6
Next value is 8
Next value is 10

Starting at 100
Next value is 102
Next value is 104
Next value is 106
Next value is 108
Next value is 110
```

- ✓ The classes that implement an interface not only limited to those methods in an interface. The class can provide whatever additional functionality is desired.
- ✓ Any number of classes can implement an interface.

```
// Implement Series a different way.
class ByThrees implements Series {
  int start;
  int val;
  ByThrees() {
    start = 0;
    val = 0;
  }
  // Implement the methods specified by Series.
  public int getNext() {
    val += 3;
    return val;
  }
  public void reset() {
    val = start;
  }
  public void setStart(int x) {
    start = x;
    val = x;
  }
}
```

Using interface reference:

- ✓ An interface declaration creates a new reference type. When a class implements an interface, it is adding that interface's type to its type.
- ✓ Interface reference variable can refer to any object that implements the interface.

```
class SeriesDemo2 {
  public static void main(String[] args) {
    ByTwos twoOb = new ByTwos();
    ByThrees threeOb = new ByThrees();

    Series iRef; // an interface reference

    for(int i=0; i < 5; i++) {
      iRef = twoOb; // refers to a ByTwos object
      System.out.println("Next ByTwos value is " +  iRef.getNext());
      iRef = threeOb; // refers to a ByThrees object
      System.out.println("Next ByThrees value is " + iRef.getNext());
    }
  }
}
```

Implementing multiple interfaces:

- ✓ A class can implement more than one interface.
- ✓ Multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

```
interface IfA {
  void doSomething();
}
interface IfB {
  void doSomethingElse();
}
// Implement both IfA and IfB.
class MyClass implements IfA, IfB {
  public void doSomething() {
    System.out.println("Doing something.");
  }
  public void doSomethingElse() {
    System.out.println("Doing something else.");
  }
}
```

- ✓ If a class implements two interfaces that declare the same method, then the same method implementation will be used for both interfaces. This means that only one version of the method os defined by the class.

```
// Both IfA and IfB declare the method doSomething().
interface IfA {
  void doSomething();
}
interface IfB {
  void doSomething();
}
// Implement both IfA and IfB
class MyClass implements IfA, IfB {
  // This method implements both IfA and IfB.
  public void doSomething() {
    System.out.println("Doing something.");
  }
}
class MultiImpDemo {
  public static void main(String[] args) {
    IfA aRef;
    IfB bRef;
    MyClass obj = new MyClass();

    // Both interfaces use the same doSomething().
    aRef = obj;
    aRef.doSomething();
    bRef = obj;
    bRef.doSomething();
  }
}
```

Constants in Interfaces:

- ✓ The primary purpose of an interface is to declare methods that provide a well defined interface to functionality. An interface can also include variables, but these are not instance variables instead they are implicitly **public, final, static** and must be initialized.
- ✓ To define a set of shared constants, simply create an interface that contains only those constants without any methods. Each class that needs to access the constants simply “implements” the interface.

```
// An interface that contains constants.
interface IConst {
  int MIN = 0;
  int MAX = 10;
  String ERRORMSG = "Boundary Error";
}
// Gain access to the constants by implementing IConst.
class IConstDemo implements IConst {
  public static void main(String[] args) {
     int[] nums = new int[MAX];
     for(int i=MIN; i < (MAX + 1); i++)
     {
              if(i >= MAX) System.out.println(ERRORMSG);
              else
              {
                       nums[i] = i;
                       System.out.print(nums[i] + " ");
              }
    }
  }
}
```

INTERFACES CAN BE EXTENDED:

- ✓ 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 inherits 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[] args) {
    MyClass ob = new MyClass();
    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

O/P:

```
Implement meth1().
Implement meth2().
Implement meth3().
```

Nested Interfaces

- ✓ An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*.
- ✓ A nested interface can be declared as **public**, **private**, or **protected**. This differs from a top-level interface, which must either be declared as **public** or use the default access level.
- ✓ When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

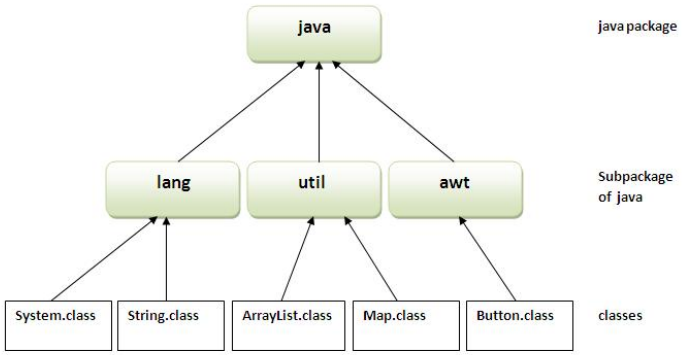
```
// A nested interface example.
// This interface contains a nested interface.
interface A {
  // this is a nested interface
  public interface NestedIF {
    boolean isNotNegative(int x);
  }
  void doSomething();
}

// This class implements the nested interface.
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false: true;
  }
}
class NestedIFDemo {
  public static void main(String[] args) {
    // use a nested interface reference
    A.NestedIF nif = new B( );
    if(nif.isNotNegative(10))
      System.out.println("10 is not negative");
    if(nif.isNotNegative(-12))
      System.out.println("this won't be displayed");
  }
}
```

PACKAGE

- ✓ A **java package** is a group of similar types of classes, interfaces and sub-packages.
- ✓ Package in java can be categorized in two form, built-in package and user-defined package.
- ✓ There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.



Defining a Package:

- ✓ To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.
- ✓ This is the general form of the **package** statement:

  package *pkg*;
- ✓ Here, *pkg* is the name of the package. For example, the following statement creates a package called **MyPackage**:

  package MyPackage;
- ✓ More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong.
- ✓ You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

  package *pkg1*[.*pkg2*[.*pkg3*]];
- ✓ A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

  package java.awt.image;

Finding Packages and CLASSPATH

- ✓ The Java run-time system looks for packages in three ways.
- ✓ First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
- ✓ Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
- ✓ Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.
- ✓ When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to* **MyPack**. For example, in a Windows environment, if the path to **MyPack** is

  C:\MyPrograms\Java\MyPack
- ✓ Then the class path to **MyPack** is

  C:\MyPrograms\Java

- ✓ **A simple package example.**
- ✓ Save the following file as A.java in a folder called **pack.**

```
package pack;
public class A {
    public void msg()
    {
            System.out.println("Hello");}
    }
```

- ✓ Save the following file as B.java in a folder called **mypack.**

```
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

- ✓ Assume that **pack** and **mypack** folders are in the directory **E:/java**.
- ✓ **compile:**

```
E:/java>javac mypack/B.java
```

Running:

```
E:/java>java mypack/B
Hello
```

PACKAGES AND MEMBER ACCESS:

- ✓ Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.
- ✓ Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- ✓ Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.
- ✓ The class is Java's smallest unit of abstraction.
- ✓ Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:
    - Subclasses in the same package
    - Non-subclasses in the same package
    - Subclasses in different packages
    - Classes that are neither in the same package nor subclasses

CLASS MEMBER ACCESS

	Private Member	Default Member	Protected Member	Public Member
Visible within same class	YES	YES	YES	YES
Visible within same package by subclass	NO	YES	YES	YES
Visible within same package by non-subclass	NO	YES	YES	YES
Visible within different package by subclass	NO	NO	YES	YES
Visible within different by non-subclass	NO	NO	NO	YES

A package access example:

The following is saved as **Protection.java** in package p1

```
package p1;
public class Protection {
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;
        public Protection() {
                System.out.println("base constructor");
                System.out.println("n = " + n);
                System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

Derived.java in package p1

```
package p1;
class Derived extends Protection {
        Derived() {
                System.out.println("derived constructor");
                System.out.println("n = " + n);
                / / class only
                / / System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

SamePackage.java in package p1

```
package p1;
class SamePackage {
        SamePackage() {
                Protection p = new Protection();
                System.out.println("same package constructor");
                System.out.println("n = " + p.n);
                / / class only
                / / System.out.println("n_pri = " + p.n_pri);
                System.out.println("n_pro = " + p.n_pro);
                System.out.println("n_pub = " + p.n_pub);
        }
}
```

Protection2.java in package p2

```
package p2;
class Protection2 extends p1.Protection {
        Protection2() {
                System.out.println("derived other package constructor");
                / / class or package only
                / / System.out.println("n = " + n);
                / / class only
                / / System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

OtherPackage.java in package p2

```
package p2;
class OtherPackage {
        OtherPackage() {
                p1.Protection p = new p1.Protection();
                System.out.println("other package constructor");
                / / class or package only
                / / System.out.println("n = " + p.n);
                / / class only
                / / System.out.println("n_pri = " + p.n_pri);
                / / class, subclass or package only
                / / System.out.println("n_pro = " + p.n_pro);
                System.out.println("n_pub = " + p.n_pub);
        }
}
```

```
/ / Demo package p1.
package p1;
/ / Instantiate the various classes in p1.
public class Demo {
        public static void main(String args[]) {
                Protection ob1 = new Protection();
                Derived ob2 = new Derived();
                SamePackage ob3 = new SamePackage();
        }
}
```

O/P for Demo in p1

```
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
derived constructor
n = 1
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
same package constructor
n = 1
n_pro = 3
n_pub = 4
```

```
/ / Demo package p2.
package p2;
/ / Instantiate the various classes in p2.
public class Demo {
        public static void main(String args[]) {
                Protection2 ob1 = new Protection2();
                OtherPackage ob2 = new OtherPackage();
        }
}
```

O/P for Demo in p2

```
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
derived other package constructor
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
other package constructor
n_pub = 4
```

IMPORTING PACKAGES:

- ✓ Java includes the import statement to bring certain classes, or entire packages, into visibility.
- ✓ Once imported, a class can be referred to directly, using only its name.
- ✓ In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.
- ✓ This is the general form of the **import** statement:

  import *pkg1 .pkg2.classname* | *;
- ✓ Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**).

  Eg: **import** mypack.MyClass;
  import mypack.*;
- ✓ ***** indicates that the Java compiler should import the entire package.
- ✓ All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called **java.lang**.
- ✓ It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

- ✓ The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
}
```

JAVA'S Standard packages:

Sub package	Description
java.lang	Contains a large number of general –purpose classes
java.io	Contains the I/O classes
java.net	Contains those classes that support networking
java.applet	Contains classes for creating applets
java.awt	Contains classes that support the Abstract Window Toolkit
java.util	Contains various utility classes, plus the Collections Framework

STATIC IMPORT:

- ✓ Java includes a feature called *static import* that expands the capabilities of the **import** keyword. By following import with the keyword static, an import statement can be used to import the static members of a class or interface.
- ✓ When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class.

```
// Compute the hypotenuse of a right triangle.
import java.lang.Math.sqrt;
import java.lang.Math.pow;
class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot;
        side1 = 3.0;
        side2 = 4.0;
        // Notice how sqrt() and pow() must be qualified by
        // their class name, which is Math.
        hypot = Math.sqrt(Math.pow(side1, 2) +Math.pow(side2, 2));
        System.out.println("Given sides of lengths " +side1 + " and "
                + side2 +" the hypotenuse is " +hypot);
    }
}
```

Given sides of lengths 3.0 and 4.0 the hypotenuse is 5.0

- ✓ Because **pow( )** and **sqrt( )** are static methods, they must be called through the use of their class' name, **Math**. This results in a somewhat unwieldy hypotenuse calculation:

```
hypot = Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));
```

- ✓ We can eliminate the tedium of specifying the class name through the use of static import, as shown in the following version of the preceding program:

```
// Compute the hypotenuse of a right triangle.
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot;
        side1 = 3.0;
        side2 = 4.0;
        // Notice how sqrt() and pow() must be qualified by
        // their class name, which is Math.
        hypot = sqrt(pow(side1, 2) +pow(side2, 2));
        System.out.println("Given sides of lengths " +side1 + " and "
                + side2 +" the hypotenuse is " +hypot);
    }
}
```

- ✓ The second form of static import imports all static members of a given class or interface. Its general form is shown here:

```
import static pkg.type-name.*;
```

- ✓ One other point: in addition to importing the static members of classes and interfaces defined by the Java API, you can also use static import to import the static members of classes and interfaces that you create.