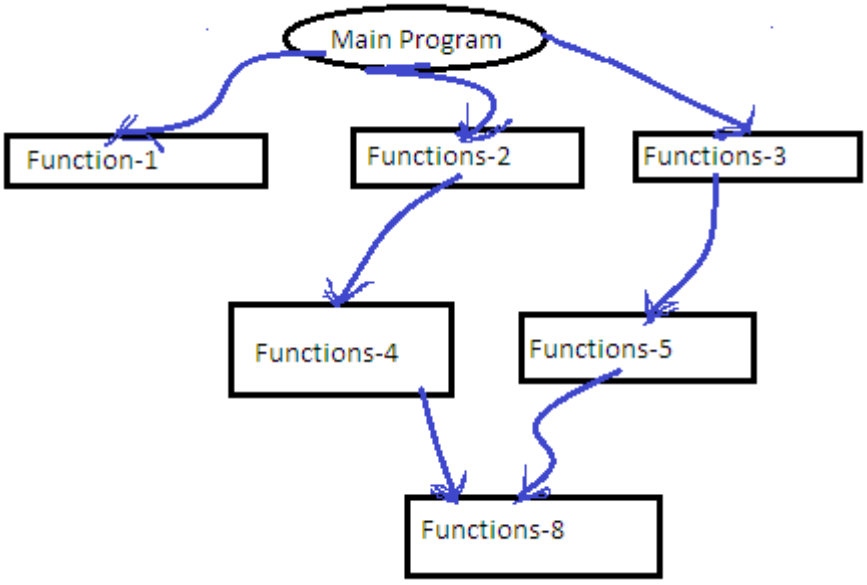
Chapter-4

Introduction to OOPs

POP

Languages like Pascal, C, FORTRAN, and COBOL are called procedure oriented programming languages. Since in these languages, a programmer uses procedures or functions to perform a task. When the programmer wants to write a program, he will first divide the task into separate sub tasks, each of which is expressed as functions/ procedures. This approach is called procedure oriented approach.



A typical POP structure is shown in above:

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Global data are more vulnerable to an in advent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we should also revise all the functions that access the data. This provides an opportunity for bugs to creep in.

Drawback: It does not model real world problems very well, because functions are action oriented and do not really corresponding to the elements of the problem.

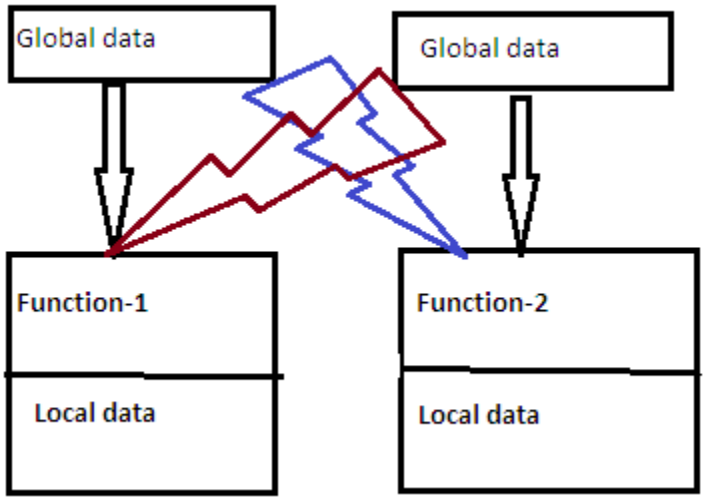
Characterstics of POP:

- Emphasis is on doing actions.
- Large programs are divided into smaller programs known as functions.
- Most of the functions shared global data.

- Data move openly around the program from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

OOP:

The languages like C++ and Java use classes and object in their programs and are called Object Oriented Programming languages. The main task is divided into several modules and these are represented as classes. Each class can perform some tasks for which several methods are written in a class. This approach is called Object Oriented approach.

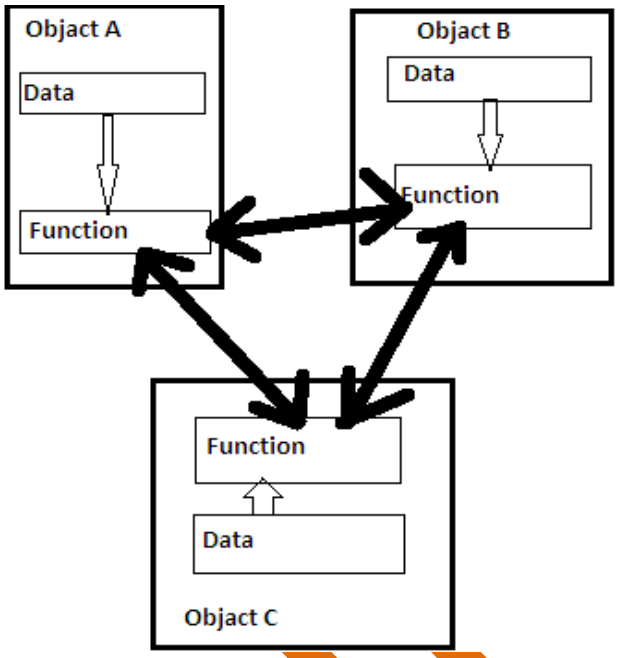


OOP allows us to decompose a problem into a number of entities called objects and then builds data and methods around these entities.

DEF: OOP is an approach that provides a way of modularizing programs by creating portioned memory area for both data and methods that can used as templates for creating copies of such modules on demand.

That is, an object a considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

Organization of OOP:



Characterstics OOP:

- Emphasis on data.
- Programs are divided into what are known as methods.
- Data structures are designed such that they characterize the objects.
- Methods that operate on the data of an object are tied together.
- Data is hidden.
- Objects can communicate with each other through methods.
- Reusability.
- Follows bottom-up approach in program design.

Class: In object-oriented programming, a class is a programming language construct that is used as a blueprint to create objects. This blueprint includes attributes and methods that the created objects all share. Usually, a class represents a person, place, or thing - it is an abstraction of a concept within a computer program. Fundamentally, it encapsulates the state and behavior of that which it conceptually represents. It encapsulates state through data placeholders called member variables; it encapsulates behavior through reusable code called methods.

class *classname* {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(*parameter-list*) {
// body of method

```
}
type methodname2(parameter-list) {
// body of method

}
// ...
type methodnameN(parameter-list) {
// 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. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that determine how a class' data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.

Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early. All methods have the same general form as **main( )**, which we have been using thus far. However, most methods will not be specified as **static** or **public**. Notice that the general form of a class does not specify a **main( )** method. Java classes do not need to have a **main( )** method. You only specify one if that class is the starting point for your program. Further, applets don't require a **main( )** method at all.

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called **Book** that defines three instance variables: **name**, **authorName**, and **nopages**. Currently, **Book** does not contain any methods (but some will be added soon).

```
class Book
{
  String Name;
  String authorName;
  int  nopages;
}
```

As stated, a class defines a new type of data. In this case, the new data type is called **Book**. You will use this name to declare objects of type **Book**. It is important to remember that a **class** declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Book** to come into existence.

To actually create a **Book** object, you will use a statement like the following:

```
Book mybook = new Book(); // create a Book object called mybook
```

After this statement executes, **mybook** will be an instance of **Book**. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement. Again, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Book** object will contain its own copies of the instance variables **Name**, **authorName**, and **nopages**. 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. For example, to assign the nopages variable of **mybook** the value 500, you would use the following statement:

```
mybook.nopages = 500;
```

This statement tells the compiler to assign the copy of **nopages** that is contained within the **mybook** object the value of 500. In general, you use the dot operator to access both the instance variables and the methods within an object.

Here is a complete program that uses the **Book** class:

```
/* A program that uses the Box class.
Call this file BookDemo.java
*/
class Book {
String Name;
String authorName;
int  nopages;
}
// This class declares an object of type Book.
class BookDemo {
public static void main(String args[]) {
Book mybook = new Book();

// assign values to mybook's instance variables

mybook.name = "Java Programming";
mybook.authorName = Mahesh Manchanda;
mybook.nopages = 500;


// Display the contents of mybook object
System.out.println("Book Name  is " + mybook.name);
System.out.println("Author Name  is " + mybook.authorName);
System.out.println("Number of pages in a book " + mybook.nopages);

}
```

```
}
```

You should call the file that contains this program **BookDemo.java**, because the **main( )** method is in the class called **BookDemo**, not the class called **Book**. When you compile this program, you will find that two **.class** files have been created, one for **Book** and one for **BookDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Book** and the **BookDemo** class to actually be in the same source file. You could put each class in its own file, called **Book.java** and **BookDemo.java**, respectively.

To run this program, you must execute **BoookDemo.class**. When you do, you will see the following output:

Output:

Book Name is Java: Programming
Author name is: Mahesh Manchanda
Number of pages in a book is: 500

Another Sample Program of class and Object Implementation

```
// This program declares two Box objects.
class Box {
double width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// compute volume of first box
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
```

Note: Variables inside a class are called as instance variables.
Variables inside a method are called as method variables.

Decelerating Object:

An Object is a real time entity. An object is an instance of a class. Instance means physically happening. An object will have some properties and it can perform some actions. Object contains variables and methods. The objects which exhibit similar properties and actions are grouped under one class. "To give a real world analogy, a house is constructed according to a specification. Here, the specification is a blueprint that represents a class, and the constructed house represents the object".

To access the properties and methods of a class, we must declare a variable of that class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.

We must acquire an actual, physical copy of the object and assign it to that variable. We can do this using **new** operator. The new operator dynamically allocates 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. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object

mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. After this line executes, **mybox** contains the value **null**, which indicates that it does not yet point to an actual object. Any attempt to use **mybox** at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds the memory address of the actual **Box** object.

A Closer Look at new

As just explained, the **new** operator dynamically allocates memory for an object. It has this general form:

class-var = new *classname*( );

Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**.

At this point, you might be wondering why you do not need to use **new** for such things as integers or characters. The answer is that Java's simple types are not implemented as objects. Rather, they are implemented as "normal" variables.

Let's once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.)

It is important to keep this distinction clearly in mind.

Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects.

However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object. This situation is depicted here:

Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```
Box b1 = new Box();
```

Box b2 = b1;

// ...

b1 = null;

Here, **b1** has been set to **null**, but **b2** still points to the original object.

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

Introducing Methods

As mentioned at the beginning of this chapter, classes usually consist of two things: instance variables and methods.

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 name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope. 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. 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.

In the next few sections, you will see how to create various types of methods, including those that take parameters and those that return values.

Using a Method with Box Class

Let's begin by adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is

dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// This program includes a method inside the box class.
class Box {
double width;
double height;
double depth;
// display volume of a box
double volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}

class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// display volume of first box
mybox1.volume();
// display volume of second box
mybox2.volume();
}
}
```


This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

There is something very important to notice inside the **volume( )** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator. 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. This means

that **width**, **height**, and **depth** inside **volume( )** implicitly refer to the copies of those variables found in the object that invokes **volume( )**.

Method Returning a Value

While the implementation of **volume( )** does move the computation of a box's volume inside the **Box** class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement **volume( )** is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that changes:

In Box class volume() method defined like the code given below:

```
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

and in main method invoke the volume( ) method like code given below:

```
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

As you can see, when **volume( )** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume( )**. Thus, after

```
vol = mybox1.volume();
```

executes, the value of **mybox1.volume( )** is 3,000 and this value then is stored in **vol**. There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

One more point: The preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. The call to **volume( )** could have been used in the **println( )** statement directly, as shown here:

```
System.out.println("Volume is " + mybox1.volume());
```

In this case, when **println( )** is executed, **mybox1.volume( )** will be called automatically and its value will be passed to **println( )**.

Adding a Method That Takes Parameters

While some methods don't need parameters, most do. 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. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()
{
return 10 * 10;
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make **square( )** much more useful.

```
int square(int i)
{
return i * i;
}
```

Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4

In the first call to **square( )**, the value 5 will be passed into parameter **i**. In the second call, **i** will receive the value 9. The third invocation passes the value of **y**, which is 2 in this example. As these examples show, **square( )** is able to return the square of whatever data it is passed.

It is important to keep the two terms *parameter* and *argument* straight. A *parameter* is a variable defined by a method that receives a value when the method is called.

For example, in **square( )**, **i** is a parameter. An *argument* is a value that is passed to a method when it is invoked. For example, **square(100)** passes 100 as an argument. Inside **square( )**, the parameter **i** receives that value.

Variable Argument (Varargs):

The varrags allows the method to accept zero or multiple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

Advantage of Varargs:

We don't have to provide overloaded methods so less code.

Syntax of varargs:

The varargs uses ellipsis[…] i.e. three dots after the data type. Syntax is as follows:

```
<return_type>  <method_name>(<data_type>...< variableName>)
{
}
```

Simple Example of Varargs in java:

```
class VarargsExample1
```

```
{
   static void display(String... values)
   {
    System.out.println("display method invoked ");
   }

  public static void main(String args[])
  {
   display();//zero argument
   display("my","name","is","varargs");//four arguments
  }
}
```

```
Output:display method invoked
       display method invoked
```

Another Program of Varargs in java:

```
class VarargsExample
{
   static void display(String... values)
   {
    System.out.println("display method invoked ");
    for(String  s : values)
    {
     System.out.println(s);
    }
   }
  public static void main(String args[])
  {
    display();//zero argument
    display("hello");//one argument
    display("my","name","is","varargs");//four arguments
  }
}
```

```
Output:display method invoked
       display method invoked
       hello
       display method invoked
       my
```

```
name
is
varargs
```

Rules for varargs:

While using the varargs, you must follow some rules otherwise program code won't compile. The rules are as follows:

- There can be only one variable argument in the method.
- Variable argument (varargs) must be the last argument.

Examples of varargs that fails to compile:

```
void method(String... a, int... b) //Compile time error
{
}

void method(int... a, String b)//Compile time error
{
}
```

Example of Varargs that is the last argument in the method:

```
class VarargsExample3
{

 static void display(int num, String... values)
{
  System.out.println("number is "+num);
  for(String s:values)
  {
    System.out.println(s);
  }
 }

 public static void main(String args[])
{
    display(500,"hello");//one argument
    display(1000,"my","name","is","varargs");//four arguments
 }
}
```

```
Output:number is 500
       hello
       number is 1000
       my
       name
       is
       varargs
```

A varargs method can also be overloaded by a non-varargs method. For example, **vaTest(int x)** is a valid overload of **vaTest( )** in the foregoing program. This version is invoked only when one **int** argument is present. When two or more **int** arguments are passed, the varargs version **vaTest(int…v)** is used.

Varargs and Ambiguity

Somewhat unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method. For example, consider the following program:

```
// Varargs, overloading, and ambiguity.




class VarArgs
{
  static void vaTest(int ... v)
  {
    System.out.print("vaTest(int ...): " + "Number of args: " + v.length +
     " Contents: ");

    for(int x : v)
        System.out.print(x + " ");

    System.out.println();
  }


  static void vaTest(boolean ... v)
```

```
  {
    System.out.print("vaTest(boolean ...) " + "Number of args: " +   v.length + "
Contents: ");
      for(boolean x : v)
        System.out.print(x + " ");

      System.out.println();
  }
  public static void main(String args[])
  {
    vaTest(1, 2, 3); // OK
    vaTest(true, false, false); // OK
    vaTest(); // Error: Ambiguous! Compile
  }
}
```

In this program, the overloading of **vaTest( )** is perfectly correct. However, this program will not compile because of the following call:

```
vaTest(); // Error: Ambiguous!
```

Constructors

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. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim( )** with a constructor.

Recap:

- A constructor is similar to a method that initializes the instance variables of a class.
- A constructor name and classname must be same.
- A constructor may have or may not have parameters. Parameters are local variables to receive data.
- A constructor without any parameters is called default constructor.

Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println( )** statement inside **Box( )** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object.

Before moving on, let's reexamine the **new** operator. As you know, 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. Thus, in the line

Box mybox1 = new Box();

new Box( ) is calling the **Box( )** constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor. 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.

Parameterized Constructors

While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes them much more useful. For example, the following version of **Box** defines a parameterized constructor which sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.

```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;

// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

```
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

The output from this program is shown here:

Volume is 3000.0
Volume is 162.0

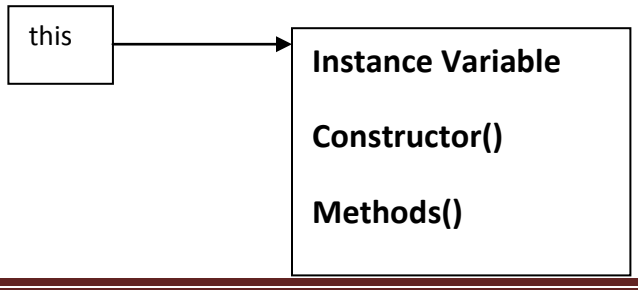
As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,

Box mybox1 = new Box(10, 20, 15);

the values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

this Keyword

'this' is a keyword that refers to the object of the class where it is used. In other words, 'this' refers to the object of the present class. Generally, we write instance variables, constructors and methods in a class. All these members are referenced by 'this'. When an object is created to a class, a default reference is also created internally to the object, as shown in Figure. This default reference is nothing but 'this'. So 'this' can refer to all the things of the present object.



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. You can use **this** anywhere a reference to an object of the current class' type is permitted.

Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box( )** constructor inside the **Box** class. If they had been, then **width** would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables. For example, here is another version of **Box( )**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.

Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```


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 for you automatically. The technique that accomplishes this is called *garbage collection.* It works like this: 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. There is no explicit need to destroy objects as in C++. 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. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The finalize( ) Method

We have seen that a constructor method is used to initialize an object when it is declared. This process is known as *initialization*. Similarly Java supports a concept called *finalization*, which is just opposite to initialization. We know that Java run-time is an automatic garbage collecting system. It automatically frees up the memory resources used by the objects. Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.* Since garbage collector cannot free these resources, so to free these resources we must use a finalizer method. This is similar to destructor in C++.

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 reclaimed an object of that class. Inside the **finalize( )** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.

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. It is important to understand that **finalize( )** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( )** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

Overloading Methods

Method overloading means method name will be same but each method should be different parameter list.

```
class prg1
{
int x=5,y=5,z=0;
public void sum()
{
z=x+y;
System.out.println("Sum is "+z);
}

public void sum(int a,int b)
{
x=a;
y=b;
z=x+y;
System.out.println("Sum is "+z);
}
public int sum(int a)
{
x=a;
y=a;
z=x+y;
return z;
}
}
class Demo
{
public static void main(String args[])
{
prg1 obj=new prg1();
obj.sum();
obj.sum(10,12);
System.out.println(+obj.sum(15));
}
}
```


Output:
sum is 10
sum is 22
27

Overloading Constructors
In addition to overloading normal methods, you can also overload constructor methods.

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
*/

class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
```

```
}
```

In general, there are two ways that a computer language can pass an argument to a method. The first way is ***call-by-value***. This approach copies the *value* of an *argument* into the *formal parameter* of the method. Therefore, changes made to the *parameter* of the method have no effect on the *argument*.

Example:

```
class ByValueExample
{
public void get(int x,int y)
{
x=x*x; //Changing the values of passed arguments
y=y*y; //Changing the values of passed arguments
}
}
class DemoByValue
{
public static void main(String args[])
{
int a,b;
a=1;
b=2;
System.out.println("Initial Values of a & b "+a+" "+b);
ByValueExample obj=new ByValueExample ();
obj.get(a,b);
System.out.println("Final Values "+a+" "+b);
}
}
```

Output:

```
Initial Values of a & b 1 2
Final Values 1 2
```

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.

Passing Objects as Parameters

Objects can even be passed as parameters.

```
// Objects may be passed to methods.
```

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

// return true if o is equal to the invoking object
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
}
}
```

This program generates the following output:
ob1 == ob2: true
ob1 == ob3: false

Copy Constructor

One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter. For example, the following version of **Box** allows one object to initialize another:

Adding below segment of code in Box class

```
// Notice this constructor. It takes an object of type Box.
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
```

Adding below segment of code in main method

```
Box mybox1 = new Box(10, 20, 15);
Box myclone = new Box(mybox1); // create copy of mybox1
```

REMEMBER: *When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.*

Understanding static

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

There will be times when you will want to define a class member that will be used independently of any object of that class. That is, the member belongs to the class as whole rather than the objects created from the class. 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.

When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable. Methods declared as **static** have several restrictions:

- They can only call other **static** methods.
- They must only access **static** data.
- They cannot refer to **this** or **super** in any way.

Static Block

Static block is a set of statements, which will be executed by the JVM before execution of main method. At the time of class loading if we want to perform any activity we have to define that activity inside static block because this block execute at the time of class loading. In a class we can take any number of static block but all these blocks will be execute from top to bottom.

Syntax

```
static
{
.....
//Set of Statements
.....
}
```

Note: In real time application static block can be used whenever we want to execute any instructions or statements before execution of main method.

Example of Static Block

```
class StaticDemo
{
static
{
System.out.println("Hello how are u ?");
}
public static void main(String args[])
{
System.out.println("This is main()");
}
}
```

Output

```
Hello how are u ?
This is main()
```

Run java program without main method

```
class StaticDemo
{
static
{
System.out.println("Hello how are u ?");
}
}
```

Output

```
Output:
Hello how are u ?
Exception is thread "main" java.lang.no-suchmethodError:Main
```

Note: "Exception is thread "main" java.lang.no-suchmethodError:Main" warning is given in java 1.7 and its above versions

More than one static block in a program

```
class StaticDemo
{
static
{
System.out.println("First static block");
}
static
{
System.out.println("Second Static block");
}
public static void main(String args[])
{
System.out.println("This is main()");
```

```
}
}
```

Output

```
Output:
First static block
Second static block
This is main()
```

Note: "Here static block run according to there order (sequence by) from top to bottom.

Why a static block executes before the main method ?

A class has to be loaded in main memory before we start using it. Static block is executed during class loading. This is the reason why a static block executes before the main method.

Example of static Members and static block

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes, which prints a message and then initializes **b** to **a * 4** or **12**. Then **main( )** is called, which calls **meth( )**, passing **42** to **x**. The three **println( )** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
```

b = 12

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class.

Here is an example. Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
class StaticDemo
{
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}

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

Here is the output of this program:
a = 42
b = 99

Recall:
Static Methods:

- Static methods can read and act upon static variables.
- Static methods cannot read and act upon instance variables.
- Static variable is a variable whose single copy is shared by all the objects.
- Static methods are declared using keyword static.
- Static methods can be called using classname.methodname.
- From any object, if static variable is modified it affects all the objects. Static variables are stored on method area.

Access Specifiers:
An access specifier is a key word that represents how to access a member of a class. There are four access specifiers in java.

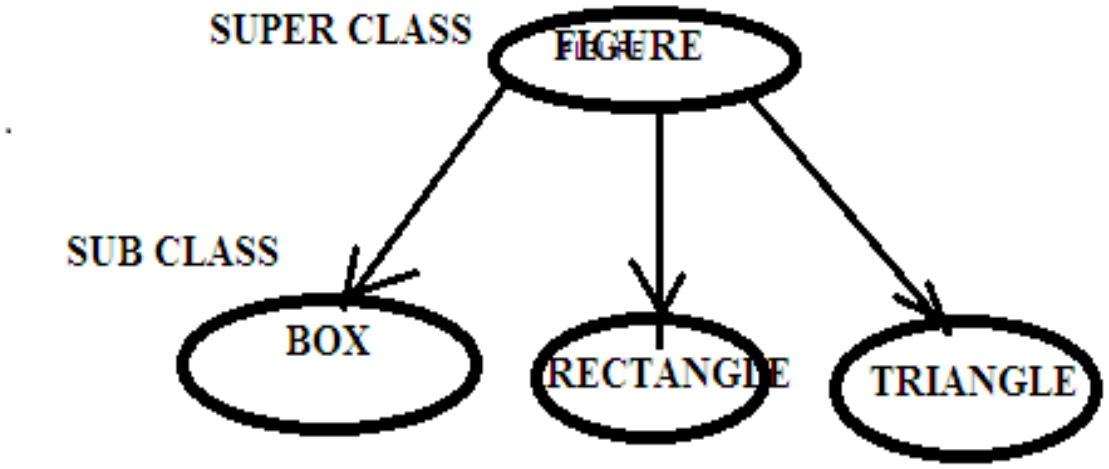
- **private: ‘**private’ members of a class are not accessible any where outside the class. They are accessible only within the class by the methods of that class.
- **public: ‘**public’ members of a class are available anywhere outside the class. So any other program can read them and use them.
- **protected: ‘**protected’ members of a class are available outside the class using inheritance, but generally, within the same directory
- **default:** if no access specifier is written by the programmer, then the Java compiler uses a ‘default’ access specifier. ‘default’ members are accessible outside the class, but within the same directory (package).

INHERITANCE: EXTENDING CLASS

Inheritance is the process by which objects of one class acquire the properties of objects of another class. Inheritance supports the concept of hierarchical classification. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the class hierarchy.

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way.

Inheritance: A new class (derived class, child class or Sub class) is derived from the existing class(base class, parent class or super class).



Main uses of Inheritance:

1. Reusability
2. Abstraction

Syntax:

Class Sub-classname **extends** Super-classname
{
    Declaration of variables;
    Declaration of methods;
}

Super class: In Java a class that is inherited from is called a super class.
Sub class: The class that does the inheriting is called as subclass.

Therefore, a subclass is a specialized version of a super class. It inherits all of the instance variables and methods defined by the super class and add its own, unique elements.

The "**extends**" keyword indicates that the properties of the super class name are extended to the subclass name. The sub class now contain its own variables and methods as well those of the super class. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying the super class members.

To see how, let's begin with a short example. The following program creates a super class called A and a subclass called B. Notice how the keyword extends is used to create a subclass of A.


```
// A simple example of inheritance.
// Create a superclass.
class A
{
  int i, j;
   void showij()
  {
    System.out.println("i and j: " + i + " " + j);
  }
}
// Create a subclass by extending class A.
```

```
class B extends A
{
   int k;
  void showk()
  {
    System.out.println("k: " + k);
}
void sum()
{
System.out.println("i+j+k: " + (i+j+k));
}
}
class SimpleInheritance {
public static void main(String args[]) {
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

As you can see, the subclass B includes all of the members of its super class, A. This is why *subOb* can access i and j and call *showij( )*. Also, inside *sum( )*, i and j can be referred to directly, as if they were part of B. Even though A is a super class for B, it is also a completely independent, stand-alone class. Being a super class for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a super class for another subclass.

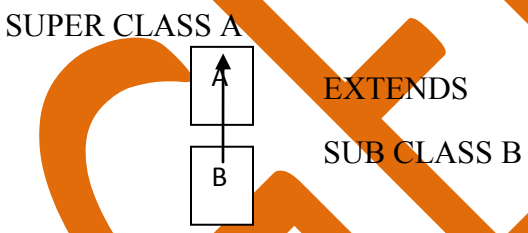
Types of Inheritance

Up to this point, we have been using simple class hierarchies that consist of only a super class and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a super class 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 super classes. In this case, C inherits all aspects of B and A.

Types of Inheritance are use to show the Hierarchical abstractions. They are:

- Single Inheritance
- Multiple Inheritance (with interface)
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance

Single Inheritance: Simple Inheritance is also called as single Inheritance. Here One subclass is deriving from one super class.



Example:

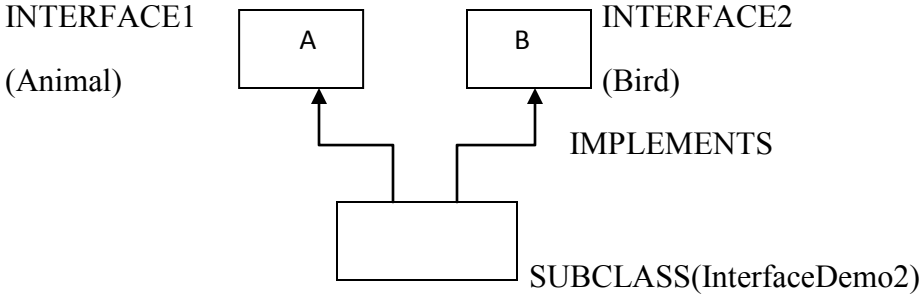
```
import java.io.*;
```

```
abstract class A
{
   void display()
        {
           System.out.println(“Display version of Parent Class”);
        }
}
class B extends A
{
        void display()
        {
                System.out.println("hello");

        }
        public static void main(String args[])
        {
                B b=new B();
                b.display();
                super.display();

        }
}
```

Output:

Hello

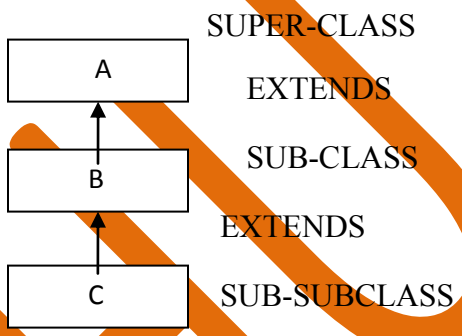
Multiple Inheritance: Deriving one subclass from more than one super classes is called multiple inheritance.



We know that in multiple inheritance, sub class is derived from multiple super classes. If two super classes have same names for their members then which member is inherited into the sub class is the main confusion in multiple inheritance. This is the reason, Java does not support the concept of multiple inheritance,. This confusion is reduced by using multiple interfaces to achieve multiple inheritance. The concept of interface learn in next chapter.

Multilevel Inheritance:

In multilevel inheritance the class is derived from the derived class.



Example: As mentioned, 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.

```
// Create a super class.
class A {
A() {
System.out.println("Inside A's constructor.");
}
}
// Create a subclass by extending class A.
class B extends A {
B() {
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor.");
}
}
class CallingCons {
```

```
public static void main(String args[]) {
C c = new C();
}
}
```

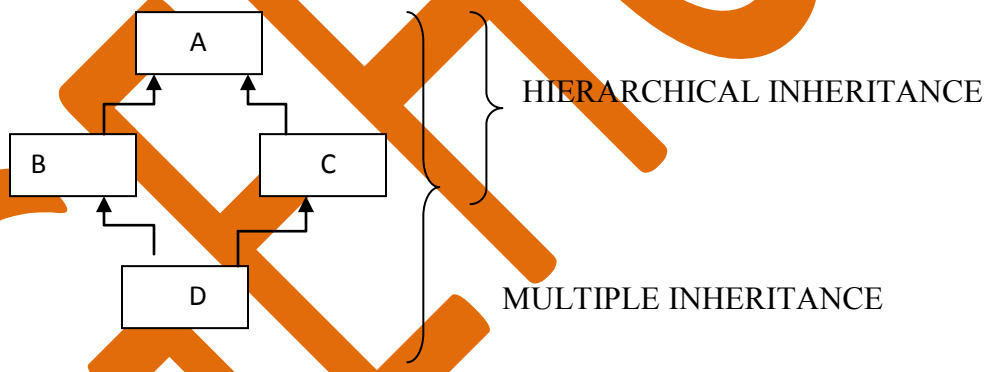
The output from this program is shown here:

Inside A's constructor
Inside B's constructor
Inside C's constructor

As you can see, the constructors are called in order of derivation. If you think about it, it makes sense that constructors are executed in order of derivation. Because a super class has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

Hybrid Inheritance:

It is a combination of multiple and hierarchical inheritance.



Subclass Constructor

A subclass constructor is used to construct the instance variable of both the subclass and the superclass. The subclass constructor uses the keyword **super** to invoke the constructor method of the superclass. **super** has the two general forms:

1. The first calls the superclass constructor. (super (args-list))
2. The second is used to access a member of the superclass that has been hidden by a member of a subclass. (super**.**member)

1. 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*);

Remember
The keyword **super** is used in the following conditions:

- Super may only be used within a subclass constructor method.
- The call to superclass constructor *super(args-list)* must appear as the first statement within the subclass constructor.
- The parameters in the super call must match the order and type of the instance variable declared in the superclass.
- Super class default constructor is available to sub class by default.
- First super class default constructor is executed then sub class default constructor is executed.

```
// A complete implementation of BoxWeight.
Class Box {
private double width;
private double height;
private double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
```

```
double volume() {
return width * height * depth;
}
}
// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight() {
super();
weight = -1;
}
// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
}
}
class DemoSuper {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
BoxWeight mybox3 = new BoxWeight(); // default
BoxWeight mycube = new BoxWeight(3, 2);
BoxWeight myclone = new BoxWeight(mybox1);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
System.out.println();
vol = mybox3.volume();
System.out.println("Volume of mybox3 is " + vol);
```

```
System.out.println("Weight of mybox3 is " + mybox3.weight);
System.out.println();
vol = myclone.volume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println();
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println();
}
}
```

```
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
```

Notice that **super( )** is passed an object of type **BoxWeight**—not of type **Box**. This still invokes the constructor **Box(Box ob)**. As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. Of course, **Box** only has knowledge of its own members.

2. A Second Use for super (Accessing the member of a super class)

Super.member;

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

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

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Method Overriding

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

```
// Method overriding.
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);
}
}
class B extends A {
int k;
B(int a, int b, int c)
{
    Super(a,b);
k = c;
}
// display k – this overrides show() in A
void show() {
super.show();  // this calls a A's 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
}
}
```

Why Overridden Method

Overridden methods allow Java to support run-time polymorphism. 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. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface.

Dynamic Method Dispatch

Dynamic method dispatch is an important mechanism in Java that is used to implement runtime polymorphism. In this mechanism, method overriding is resolved at runtime instead of compile time. That means, the choice of the version of the overridden method to be executed in response to a method call is done at runtime.

Dynamic dispatch is a mechanism by which a call to Overridden function is resolved at runtime rather than at Compile time , and this is how Java implements Run time Polymorphism. In dynamic method dispatch, super class refers to subclass object and implements method overriding.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme method");
```

```
}
}
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

Using Final with Inheritance

The keyword final has three uses:

- Create named constant
- To prevent overriding
- To Prevent Inheritance

Create Named Constant: This use was described in the previous chapter.

Using Final to Prevent Overriding

All method and variables can be overridden by default in subclasses. If we wish to prevent the subclasses from overriding the members of the superclass, we can declare them as **final** using the keyword final as a modifier. In this way you will never be altered the method in any way.

Methods declared as final cannot be overridden. The following fragment illustrates final:

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

```
System.out.println("Illegal!");
}
}
```

Because **show( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Using final to Prevent Inheritance

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

Remember:
Declaring a class as final implicitly declares all of its methods as final, too. As you might expect, 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.

Here is an example of a final class:

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

Abstract Method and Classes

We have seen that by making the method *final* we ensure that the method is not redefined in a subclass. That is, the method can never be subclassed (i.e. inherited) Java allows us to do something opposite to this. That is, we can indicate that a method must always to redefined in a subclass, thus making overriding compulsory. This is done using the modifier keyword **abstract** in the method definition.

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

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

When a class that contains one or more abstract methods must also be declared **abstract**. While using abstract classes, we must satisfy the following conditions:

- We cannot use abstract classes to instantiate object directly. For example,
  A oba = new A();
  is illegal because class A is an abstract class.

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

```
// Using abstract methods and classes.
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
```

```
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class AbstractAreas {
public static void main(String args[]) {
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
}
}
```

As the comment inside **main( )** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override **area( )**. To prove this to yourself, try creating a subclass that does not override **area( )**. You will receive a compile-time error. Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**.