# MODULE - II

# Working with Classes and Inheritance

## Introduction of Classes

### The General Form of a Class:

When you specify a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, the class code specifies the interface to its data. A class is notified of the use of the class keyword. The classes that have been used up to this point are very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a class definition is shown here:

```
class ClassName
{
   type instance-variable0;
   type instance-variable1;
   // ...
   type instance-variableP;
   type methodname0(parameter-list)
   {
      // body of method
   }
   type methodname1(parameter-list)
   {
      // body of method
   }
   // ...
   type methodnameP(parameter-list)
   {
      // method's body
   }
}
```

The data or variables, fixed 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, as a general rule, it is the methods that determine how a class' data can be used. Variables enclosed 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 return to this point Shortly, but it is a meaningful 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.

# Class Fundamentals

Classes have been used as the beginning of this book. However, until now, only the most rudimentary form of a class has been used. The classes created in the preceding chapters exist to encapsulate the main( ) method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the restricted ones presented so far.

Maybe the most important thing to learn about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a template for an object and an object is an instance of a class. Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.

## (i) Declaring Objects

The syntax for object Declaration Example of creating the object

To Declaration an object, Developer must

- Give the object the name
- Specify what type the object will be.

**Syntax of object:**

*<class name> <object name>;*

***For example,*** declaring a Student object

*Student student; //declares object*

To declare an object does not generate an object. It just sets up a named location in memory that stores an address to that object.

The object will be declared only once.

The following is incorrect and will generate a compile error.

*Student student; //declares object named student*

*Student student; //declares object named student again*

But the following will not generate any compile error because of different objects name.

*Student student1; //declares object named student1*

*Student student2; //declares different object named student2*

- To creating an object you use the syntax:

  *<object name> =new <Class name>*

  ***For example,*** Creating an object named student

  *Student=new Student //creates object*

- Declaration and creation can be prepared all in one step.

  *Student student=new Student( );//Declares and create objects*

- Often the object name is the equal as the class but not capitalised. Whatever you call the object, it is fine to name it something that makes it rather obvious what type it is.

## (ii) Assigning an Object

Object reference variables act oppositely than you might expect when an assignment takes place. Let's see a simple example, what do you think the following fragment does?

*Box p1 = new Box( );*

*Box p2 = p1;*

You might think that p2 is being assigned a reference to a copy of the object referred to by p1. That is, you might think that p1 and p2 refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, p1 and p2 will both apply to the same object. The assignment of p1 to p2 did not allocate any memory or copy any part of the original object. It simply makes p2 refer to the same object as does p1. Thus, any changes made to the subject matter through p2 will affect the object to which p1 is referring since they are the same object.

**This situation is depicted here:**

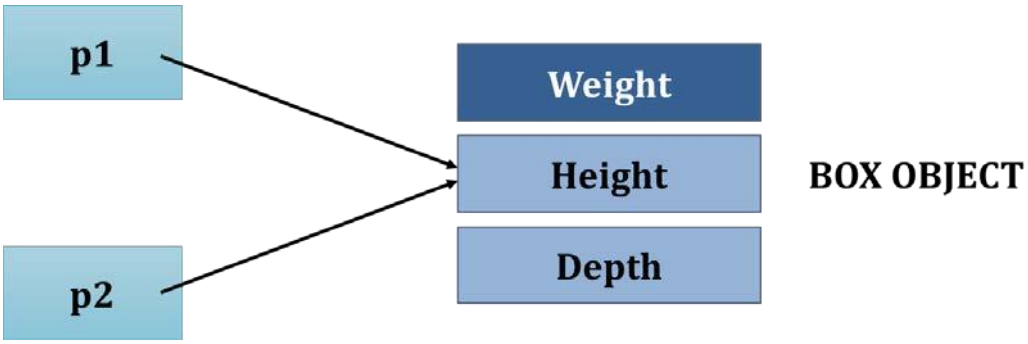


*Figure 2.1.1: Assigning an Object*

Although p1 and p2 both refer to the same object, they are not linked in any other way. ***For example,*** a subsequent assignment to p1 will only unhook p1 from the original object without affecting the object or touching p2.

***For example,***

```
Box p1 = new Box();
Box p2 = p1;
// ...
p1 = null;
```

Here, p1 has been set to null, but p2 still points to the original object.

## (iii) Constructors

Constructors are defined as special method used for object initialisation (When the loop first starts, the initialisation division of the loop is executed. It is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialisation expression is only executed once). During the time of object creation, constructors are invoked. Constructors are used to create **object** from a **class**

### How to create a Java Constructor?

There are two directives to create a constructor:

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

### Types

In Java programming, there are **two** types of constructors are used. They are:

1. Default Constructor

2. Parameterised Constructors

### Default Constructor

Before pushing on, let's reconsider 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 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 the client does not explicitly define a constructor for a class, then Java creates a default constructor for the class. It is why the preceding line of code worked in earlier versions of Box that did not define a constructor. The default constructor automatically initialises 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 the client defines their constructor, the default constructor is no longer used.

### Access modifiers:

A Java access modifier specifies in which classes can access an assigned class and its fields, constructors and methods. Access modifiers can be defined separately for a class, its constructors, fields and methods. Java access modifiers are also sometimes referred to in standard speech as Java access specifiers, but the exact name is Java access modifiers. Classes, fields, constructors and methods can have one of **four** various Java access modifiers:

1. private
2. default (package)
3. protected
4. public

To Assigning an access modifier to class, constructor, field or method is also sometimes referred to as "marking" that class, constructor, field or method as that which the access modifier specifies. For instance, assigning the Java access modifier public to a method would be referred to as marking the method as public.

### Parameterised Constructor

While the Box( ) constructor in the preceding example does initialise 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 sizes. 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 parameterised constructor that sets the dimensions of a box as specified by those parameters. Pay particular attention to how Box objects are created.

/* *Here, Box uses a parameterised constructor to*

*initialise 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 BoxDemo
{
   public static void main(String [] args)
   {
      // declare, allocate and initialise Box objects
      Box myfirstbox = new Box(5,10, 15);
      Box mysecondbox = new Box(1, 5, 7);
      double vol;
      // get volume of first box
      vol = myfirstbox.volume();
      System.out.println("Volume is " + vol);
      // get volume of second box

      vol = mysecondbox .volume();
      System.out.println("Volume is " + vol);
   }
}
```

**Output from this program is shown here:** inner class

Volume is 750.0

Volume is 35.0

As you can observe in this example, each object is initialised as specified in the parameters to its constructor. ***For example,*** in the following line, Box myfirstbox=new Box(5, 10, 15); the values 5, 10 and 15 are passed to the Box( ) constructor when new creates the object. Thus, my first box's copy of width, height and depth will contain the values 5, 10 and 15, respectively.

## (iv) "this" Keyword

Seldom a method will need to refer to the object that requested it. To allow 'this' keyword, 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. To properly understand what this leads to, consider the following version of Box( ):

```
// an excessive use of this.
Box(double w, double h, double d)
{
   this.width = w;
   this.height = h;
   this.depth = d;
}
```

This version of Box( ) operates exactly like the earlier version. The use of this is redundant but entirely correct. Inside Box( ), it will always refer to the invoking object. While it is unnecessary in this case, this is useful in other contexts, one of which is explained in the next section.

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

**A word of warning:** The use of this in such a context can sometimes be complicated and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other developers believe the contrary - that it is a good convention to use the same names for clarity and use this to overcome the occurrence variable masking. It is a matter of taste which address you adopt.

## (v) Garbage Collection

Garbage collection is a process of regaining the unused memory automatically. In Java, it is used to destroy the remaining objects. Since objects are dynamically designated by using the new operator, you might be admiring how such objects are destroyed and their memory released for succeeding reallocation. Java takes an another approach; it manages deallocation for you automatically. The technique that accomplishes this is called garbage collection. It works like this: when references to an object exist, that object is assumed to be no longer needed and the memory maintained by the object can be reclaimed. Garbage collection occurs during the execution of your program. It will not occur just 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.

**Advantage of using garbage collection**

- Memory efficient – removes unreferenced objects from heap memory.
- The process of regaining the unused memory is done automatically.

**Simple example of garbage collection in java**

```
java.lang.System.gc( )
public void finalise( )
{
   System.out.println("object is garbage collected");
   public static void main(String args[])
   {
      TestGarbage1 s1=new TestGarbage1( );
      TestGarbage1 s2=new TestGarbage1( );
      s1=null;
      s2=null;
      System.gc( );
   }
}
```

**This program would produce an output:**

object is garbage collected

object is garbage collected

In this above program.the java.lang.System.gc( ) method runs the garbage collector. Calling this implies that the JVM expend effort toward recycling unused objects to make the memory they currently occupy available for quick reuse.

# Methods and Classes

In Java programming, methods and classes are used to define and structure the programming

language by providing the required elements to implement the function.

**Let's see some of the Java methods subsequently:**

### a) finalise method :

Seldom 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 character font, then you might want to make sure these resources are freed before an object is destroyed. To deal with such situations, Java provides a mechanism called finalisation. By using finalisation, the client can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. To add a finaliser to class, you just define the finalise( ) method. The Java runtime calls that method when it is about to reuse an object of that class. Within the finalise( ) method, client/coder will specify those activities that must be performed before an object is terminated.

The garbage collector regularly runs, monitoring 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 runtime calls the finalise( ) method on the object. The finalise( ) method has this general form:

```
protected void finalise()
{
   // finalisation code here
}
```

Here, the keyword protected is a specifier that prevents access to finalise( ) by code defined outside its class. It is necessary to understand that finalise( ) method is only called just before garbage collection.

It is not called when an object goes out-of-scope, ***for example,*** it means that you cannot know when - or even if - finalise( ) will be executed. Therefore, your program should provide other ways of releasing system resources, etc., used by the object. It must not rely on finalise( ) for normal program operation.

## (i) Overloading Methods

"In Java, it is possible to define multiple methods within the similar class that share the similar name, as long as their parameter declarations are different. When this is the case, theme thods are said to be overloaded and the process is known as method overloading." Method overloading is one of the ways that Java supports polymorphism. If the client has never used a java language that allows the overloading of methods, then the concept may seem strange at first. But as client will see, method overloading is one of Java's most exciting and useful features.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class OverloadDemo
{
   void test()
   {
      System.out.println("No parameters");
   }
   // Overload test for one integer parameter.
   void test(int i)
   {
      System.out.println("i: " + i);
   }
   // Overload test for two integer parameters.
   void test(int i, int j)
   {
      System.out.println("i and j: " + i+ " " + j);
   }
   // overload test for a double parameter
   double test(double i)
   {
      System.out.println("double i: " + i);
      return i*i;
   }
}
```

```
class Overload
{
   public static void main(String args[])
   {
      OverloadDemo obj = new OverloadDemo();
      double result;
      // call all versions of test()
      obj.test();
      obj.test(20);
      obj.test(20, 30);
      result = obj.test(124.26);
      System.out.println("Result of obj.test(124.26): " + result);
   }
}
```

**This program generates the following output:**

No parameters

i: 10

i and j: 20 30

double i: 124.26

Result of obj.test(124.26): 15440.5476

As you can obeserve, test( ) is overloaded four times in preceding program. The first one takes no parameters, the second one takes one integer parameter, the third one takes two integer parameters and the fourth one takes one double parameter in the code. The fourth version of test( ) method also returns a value with no consequence relative to overloading, since return types do not play a role in overload resolution.

**Some Variations of Method Overloading:**

Que) 1: Method Overloading is not possible by changing the return type of method, Why?

In Java, method overloading is not feasible by modifying the return type of the method because there may occur uncertainty. Let's see how ambiguity may occur:

**Because there was the problem:**

```
class Calculation
{
   int sum(int i,int j){System.out.println(i+j);}
   double sum(int i,int j){System.out.println(i+j);}
   public static void main(String args[])
   {
     Calculation obj=new Calculation();
```

```
    int result=obj.sum(10,10); //Compile Time Error
  }
}
```

int result=obj.sum(10,10); //Here how can java determine which sum( ) method should be called

**Ques 2) Can we overload main( ) method?**

Yes, by method overloading. You can have any number of main methods in a class by method overloading.

**Let's see the simple example:**

```
class Overloading1
{
  public static void main(int a)
  {
    System.out.println(a);
  }
  public static void main(String args[])
  {
    System.out.println("main() method invoked");
    main(10);
  }
}
```

**Output:**

main( ) method invoked

10

## (ii) Using Object as Parameters

So distant, we have only been using secure types as parameters to methods. Despite, it is both correct and standard to pass objects to methods. ***For example,*** consider the following program:

```
// Objects may be passed to methods.
class Test
{
  int x, y;
  Test(int p, int q)
  {
    x = p;
    y= q;
  }
  // return true if o is equivalent to the invoking object
  boolean equals(Test o)
```

```
    {
      if(o.x== x&& o.y== y) return true;
      else return false;
    }
}
class PassObj
{
   public static void main(String args[])
   {
     Test obj1 = new Test(100, 22);
     Test obj2 = new Test(100, 22);
     Test obj3 = new Test(-1, -1);
     System.out.println("obj1 == obj2: " + obj1.equals(obj2));
     System.out.println("obj1 == obj3: " + obj1.equals(obj3));
   }
}
```

**This program generates the following output:**

obj1 == obj2: true

obj1 == obj3: false

As you can observe, the equals( ) method inside Test relates two objects for equivalence and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns true. Otherwise, it returns false. Notice that the setting o in equals( ) defines Test as its kind. Although Test is a class type created by the program, it is used in just the same way as Java's built-in types.

## (iii) Argument Passing

Argument passing passes the type and number of arguments for that method or constructor. There are two type of argument passing. They are:

1. Pass by value
2. Pass by reference

In Java, when you pass a fundamental type to a method, it is passed by value. Hence, what occurs to the parameter that receives the argument has no effect outside the method. Java uses only pass by value. It does not support pass by reference.

### Pass by value:

Pass by value, passes actual parameter expressions to a method which evaluates and then derives a value. This value is stored in a location and it becomes a formal parameter to the invoked method. This mechanism is called as pass by value.

***For example,*** consider the following program:

```
// Primitive types are passed by value.
class Test
{
   void math(int x, int y)
   {
     x*= 2;
     y/= 2;
   }
}
class CallByValue
{
   public static void main(String args[])
   {
     Test obj = new Test();
     int p= 15, q = 20;
     System.out.println("p and qbefore call: " +p+ " " +p);
     ob.meth(p, q);
     System.out.println("p and q after call: " +
     p+ " " + q);
   }
}
```

**Output from this program is shown here:**

p and q before call: 15 20

p and q after call: 15 20

As you can observe, the operations that occur inside math( ) have no effect on the values of p and q used in the call; their values here did not change to 30 and 10.

### Pass by reference:

In pass by reference, the formal parameter is otherwise called as actual parameter which in turn refers to the actual argument. Any modification done to the formal argument will instantly reflects in actual argument and vice versa. This mechanism is called as pass by reference.

When the client passes an object to a method, the position changes dramatically, because objects are passed by what is effectively call-by-reference. Have in mind that when you create a

variable of a class type, you are only creating a reference to an object. Therefore, when you pass this source to a method, the parameter that holds it will refer to the identical object as that referred to by the argument. It effectively means that objects are passed to methods by use of call-by-reference. Modifications to the object within the method do affect the object used as an argument.

***For example,*** consider the following program:

```
// Objects are passed by reference.
class Test
{
   int x, y;
   Test(int p, int q)
   {
     x= p;
     y= q;
   }
   // pass an object
   void meth(Test o)
   {
     o.x*= 2;
     o.y/= 2;
   }
}
class CallByRef
{
   public static void main(String args[])
   {
     Test obj = new Test(15, 20);
     System.out.println("ob.x and ob.y before call: " +
     obj.x+ " " + obj.y);
     ob.math(obj);
     System.out.println("obj.x and obj.y after call: " +
     obj.x+ " " + obj.y);
   }
}
```

**This program generates the following output:**

obj.x and obj.y before call: 15 20

obj.x and obj.y after call: 30 10

As you can see, in this case, the actions inside math( ) have changed the object used as an argument.

## (iv) Returning Objects

A process can return any data, including class natures that you create. ***For example,*** in the coming program, the incByTen( ) method returns an object in which the value of a is ten greater than it is in the invoking object.

```
// Returning an object.
class Test
{
   int p;
   Test(int x)
   {
     p = x;
   }
   Test incByTen()
   {
     Test temp = new Test(p+10);
     return temp;
   }
}
class RetObj
{
   public static void main(String args[])
   {
     Test obj1 = new Test(2);
     Test obj2;
     obj2 = obj1.incByTen();
     System.out.println("obj1.p: " + obj1.p);
     System.out.println("obj2.p: " + obj2.p);
     obj2 = obj2.incByTen();
     System.out.println("obj2.pafter second increase: "
     + obj2.p);
   }
}
```

**Output for this program is shown here:**

obj1.p: 2

obj2.p: 12

obj2.p after second increase: 22

As you can see, each time incByTen( ) is invoked, a new object is created and a reference to it is returned to the calling routine.

The above program makes an important point: Since all objects are dynamically allotted using new, client don't have to worry about an object going out-of-scope since the method in which

it was performed eliminates. The object will continue to be until there is a source to it somewhere in the program. When there are no sources to it, the object will be reclaimed the next time garbage collection takes place.

## (v) Recursion

Recursion is a process of calling by itself. In Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be **recursive**. Recursion are always managed using Stack.

Recursion looks like a never ending loop because it can never get back or it seems the method will never end .This might be true in certain cases but in practice, one can check if a certain condition is true and in that case the program can exit or return from previous method, the condition which is used to end our recursion is called a **Base Case.** The perfect example of recursion is the sum of the factorial of a number. The factorial of a number P is the product of all the whole numbers between 1 and P. ***For example,*** 4 factorial is 1*2*3*4 or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial
{
   // this is a recursive method
   int fact(int p)
   {
     int result;
     if(p==1) return 1;
     result = fact(p-1) * p;
     return result;
   }
}
class Recursion
{
   public static void main(String args[])
   {
     Factorial fa = new Factorial();
     System.out.println("Factorial of 3 is " + fa.fact(3));
     System.out.println("Factorial of 4 is " + fa.fact(4));
     System.out.println("Factorial of 5 is " + fa.fact(5));
     System.out.println("Factorial of 6 is " + fa.fact(6));
   }
}
```

**The output from this program is shown here:**

```
Factorial of 3 is 6
```

Factorial of 4 is 24

Factorial of 5 is 120

Factorial of 6 is 720

If you are unknown with recursive methods, then the operation of fact( ) may seem a bit complex. Here is how it works. When fact( ) is called with an argument of 1, the function returns 1; otherwise, it returns the product of fact(p–1)*p. To evaluate this expression, fact( )is called with p–1. This process repeats until p equals 1 and the calls to the method begin returning.

## (vi) Access Control

In Java, the unit of programming is the **class** from which objects are created. An Object encapsulates data (**attributes**) and methods (**behaviours**) inside a class. The implementation details are hidden within the objects themselves.

Java code consist of classes, whereas variables and methods are declared under class from which objects are eventually created.

**Let's see what are the four access levels and its visibility:**

1. Visible to the package ( by default no modifiers are needed)
2. Visible to the class only (private).
3. Visible to the world (public).
4. Visible to the package and all subclasses (protected).

### Access control and Inheritance:

The following rules for inherited methods are enforced:

- Methods declared public in superclass must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared private are not inherited at all, so there is no rule for them.

## (vii) Understanding Static

When the client will want to define a class member that will be used freely of any object of that class. Though, it is possible to build a member that can be utilised by itself, without the source to a particular instance. To create such a member, precede its statement with the keyword static. When a member is named static, it can be obtained from any objects of its class are built and without reference to any object. You can demonstrate 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.

**Methods declared as static have some limitations:**

- Methods can only call other static methods.
- Methods must only access static data.
- Methods cannot apply to this or super in any behaviour.

If you want to do computation in order to start your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded. The coming example shows a class that has a static method, some static variables and a static initialisation block:

```
// Demonstrate static variables, methods and blocks.
class StaticUse
{
   static int i = 3;
   static int j;
   static void meth(int p)
   {
     System.out.println("p = " + p);
     System.out.println("i = " + i);
     System.out.println("j= " + j);
   }
   static
   {
     System.out.println("Static block initialised.");
     j= i* 4;
   }
   public static void main(String args[])
   {
     math(42);
   }
}
```

Until the StaticUse 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 initialises j to i* 4 or 12. Then main()

is called, which calls meth( ), passing 42 to 'p'. The three println( ) statements refer to the two static variables i and j, as well as to the local variable p.

**Here is the output of the program:**

Static block initialised.

p= 42

i= 3

j= 12

The 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, you can do so using the following general form:

*class name.method( )*

Here, the class name is the name of the class in which the static method is disclosed. As you can see, this format is similar to that used to call non-static methods through object reference variables. A static variable can be accessed in the same way - by use of the dot operator on the name of the class. It is how Java implements a controlled version of global methods and global variables.

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

```
class StaticDemo
{
   static int i= 42;
   static int j= 99;
   static void callme()
   {
     System.out.println("i= " + i);
   }
}
class StaticByName
{
   public static void main(String args[])
   {
     StaticDemo.callme();
     System.out.println("j= " + StaticDemo.b);
   }
}
```

**Here is the output of this program:**

i= 42

j= 99

## (viii) Introducing Final

Final is used to construct an object and considered as a very important concept because it prevents refereeing to another thread when already a thread is in progress. This is also called as safe publication in Java programming.

It ensures the correct values of an object's final fields. It is a part of Java Virtual Machine which effectively ensures that object pointers are available to other threads.

A variable can be named as final. Doing so stops its contents from being changed. It means that you must initialise a final variable when it is declared.

***For example,***

final int FILE_NEW = 1;

final int FILE_OPEN = 2;

final int FILE_SAVE = 3;

final int FILE_SAVEAS = 4;

final int FILE_QUIT = 5;

Following parts of the program can now use FILE_OPEN, etc., as if they were constants, without fear that a value has been changed.

It is a primary coding convention to choose all uppercase identifiers for final variables. Variables declared as final do not extend memory on a per-instance data. Thus, a final variable is essentially a constant.

The keyword final can also be applied to methods, but its meaning is considerably different than when it is applied to variables. This second usage of final is explained in the next chapter when inheritance is described.

## (ix) Introducing Nested and Inner Class

Defining a class within another class is called as nested class. Nested class has access to the members including private members of the class in which it is nested.

**There are two types of nested classes:**

### 1. Static

Static modifier is applied in static nested class. Through an object, it must access the members of its enclosing class. Directly it cannot refer to members of its enclosing class.

### 2. Non-Static-

Non – static class is the most important type of nested class.

An inner class is the most important type of nested class. An inner class is a non-static nested class. It has access to each of the variables and methods of its outer class and may apply to them directly in the same way that other non-static members of the outer class do.

The resulting program demonstrates how to determine and implement an inner class in code. The class named Outer has one occurrence variable called outer_p, one case method named test( ) and defines one inner class called Inner.

```
// Demonstrate an inner class.
class Outer
{
   int outer_p = 100;
   void test()
   {
     Inner inner = new Inner();
     inner.display();
   }
   // this is an inner class
   class Inner
   {
     void display()
     {
       System.out.println("display: outer_p= " + outer_p);
     }
   }
}
class InnerClassDemo
{
   public static void main(String args[])
   {
     Outer outer = new Outer();
     outer.test();
   }
}
```

**Output from this application is shown here:**

display: outer_p = 100

In this program, an inner class named Inner is defined within the scope of class Outer. Therefore, any code in class Inner can directly access the variable outer_p. An instance method named display( ) is defined inside Inner. This method displays outer_p on the standard output stream. The main( ) method of InnerClassDemo creates an instance of class Outer class and invokes its test( ) method. That method produce an instance of class methods and the display( ) method is called.

**Advantages of inner class:**

There are **three** advantages of inner classes in Java. They are as follows:

1. Nested classes represent a particular type of relationship that is it can access all the members (data members and methods) of an outer class including private.
2. Nested classes are used to develop more readable and maintainable code because it logically groups classes and interfaces in one place only.
3. **Code Optimisation:** It requires less code to write.

## (x) Using Command Line Argument

Seldom you will want to pass information into a program when you run it. This is achieved by passing command-line arguments to main( ). A command-line argument is the data that directly follows the program's name on the command line when it is performed. To access the command-line arguments inside a Java application is quite easy - they are stored as strings in a String array passed to the args parameter of main( ). The first command-line argument is collected at args[0], the second at args[1] and so on.

***For example,*** the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine
{
   public static void main(String args[])
   {
     for(int p=0; p<args.length; p++)
     System.out.println("args[" + p + "]: " +
     args[p]);
   }
```

```
}
```

Try executing this program, as shown here:

The Java Command Line this is a test 100 -1

**When you do, you can see the following output:**

args[0]: this

args[1]: is

args[2]: a

args[3]: test

args[4]: 100

args[5]: -1