Lab Exercise 1

a. Demonstrate Constructor Overloading and Method Overloading in JAVA.
b. Implement Inner Classes and demonstrate its access protection.
c. Programs to create constructors and methods with a case study.

Constructor overloading and method overloading in Java

When more than a single constructor is defined in a class, it is known as constructor overloading. Similarly, when more than one method with the same name is defined in the same class, it is known as method overloading. However, there is a restriction on such overloading. Constructors or methods can be overloaded only in a way that would allow calls to these constructors or methods to be resolved i.e. when an overloaded method is invoked, the compiler should be able to decide which of the overloaded methods is to be called by checking the arguments. For example consider an overloaded method print() having two versions. The first of these does not take any arguments while the second one takes a single argument. A statement like print("Example") is a call to definitely the second version of the overloaded method which requires a single String argument.

However, if both the versions of the overloaded methods take a single String argument, then the call is non resolvable. A statement like the above could be a call to either the first or to the second version of the method. Therefore, the overloaded methods here would give compilation errors. In short, it can be stated that overloading of methods or constructors can be done only if the parameter list varies in at least one of the following things: number of arguments, types of arguments or order of arguments. The return types of the overloaded method doesn't matter.

Methods calls to overloaded versions are resolved by looking at the above stated three parameters. Following statements shows a few sets of overloaded methods and also specify if that particular set is allowed in a single class. The reason is also stated in each case.

Set 1:

```
public void print()
public void print ( String str)
```

This set is allowed since the number of arguments differ in the two cases.

Set 2:

```
public void print ( int a)
public void print ( String s)
```

This set is also allowed since the type of parameters are different even though the number of arguments are the same. One of the overloaded versions of print accepts an integer argument while the other accepts a String argument.

Set 3:

public void print (int x)

public void print (int y)

This set may appear as acceptable at the first look but it is not. These versions do not differ in either the number, order or type of arguments. The variable name in the parameter list doesn't really matter. For example, consider the following call:

print(34);

This might be a call to either the first method or the second method. Hence this set is not allowed. The ultimate rule to check the validity of overloaded methods is to see if a method call is resolvable i.e. there should be a one to one correspondence between a method call and a method. A method call should always point to a single method and there should be no doubt as to which of the methods would be called.

Set 4:

public void print (int a, String s)

public void print (String s, int a)

This set is an acceptable set for overloading the methods since the parameters differ in the order in which they are specified even though the number and type of parameters is the same.

Set 5:

public void print (int a)

public int print (int a)

The above sets of methods cannot be declared in the same class. As already said, the return type doesn't matter when deciding the mutual co-existence of overloaded methods. One might argue that the call can be resolved depending on whether the call requires a value to be returned. But such an argument isn't valid because, it isn't necessary that a returned value should always be used in one or the other way. Consider the following method call.

print(34);

Looking at this call, one can't say that the print() method doesn't return a value. If we had only the second version of print (the one that returns an int) out of the two stated versions above, that version would be the one to be called. The returned int would be simply ignored. In short, values returned by a method need not always be put to use and therefore the return type isn't checked to verify the validity of overloaded methods. In a similar way, we can check the extent to which we can overload constructors for a class.

Method Overloading:

class Overload

```
{
   int a=0,b=0;
   void add()
   {
      System.out.println("NO or DEFAULT argument add method, sum= "+(a+b));
   }
   //overloaded Method add
   void add(int k,int j)
   {
      a=k;
      b=j;
      System.out.println("Parameterized argument add method, sum= "+(a+b));
   }
}

class One {
   public static void main(String args[])
   {
      Overload ga=new Overload();

      System.out.println("Default Method call");
      ga.add();
      System.out.println("Parameterized Method call");
      ga.add(12,23);
   }
}
```

Constructor Overloading

```
class Generic
{
   int a=0,b=0;
```

```
    Generic()
    {
        a=b=1;
    }
    //overloaded constructor
    Generic(int a,int b)
    {
        this.a=a;
        this.b=b;
    }
    void print()
    {
        System.out.println("a="+a+" b="+b);
    }
}
public class One {
    public static void main(String args[])
    {
        System.out.println("Default constructor call");
        Generic ga=new Generic();
        ga.print();
        System.out.println("Parameterized constructor call");
        Generic ag=new Generic(10,20);
        ag.print();
    }
}
```

Inner Class (Non-static nested classes): An inner class is declared inside the curly braces of another enclosing class. **Inner class** acts as a member of the enclosing class and can have any access modifiers: **abstract, final, public, protected and private**. Inner class can access all

members of the outer class including those marked private. The methods and fields of an outer class are used as if they were part of the inner class. An inner class is declared in the same way as any other class but should be inside some other class.

The basic structure of creating an inner class is as follows,

```
class OuteClass{
      class InnerClass{
        // body of InnerClass
       }
  //More members of OuterClass
}
```

From the above structure, it is clear that the class InnerClass acts as a member of the enclosing class OuterClass in much the same way as its other members. So it can be marked with an access specifier, to specify whether the class should be public, protected or private. If none of the access specifiers is specified then default package access specifier is used. These access specifiers determine whether other classes can access the inner class or not.

More often it is the outer class that creates an instance of the inner class since it is usually the outer class that wants to use the inner instance as a helper object for its personal private use. To create an instance of the inner class, you must have the instance of the enclosing class so that you can associate them. To create an object of type InnerClass you have to write the following statement.

```
OuterClass outer = new OuterClass();

OuterClass.lnnerClass inner = outer.new InnerClass();
```

Program:

```
class  Outer
{
   int  outdata = 22;
   void  display()
   {
      Inner  inobj = new  Inner();
      System.out.println("Accessing from outer class");
```

```
        System.out.println("The value of outdata is " +outdata);
        System.out.println("The value of indata is "   +inobj.indata);
    }
    class  Inner
    {
        int  indata = 20;
        void  inmethod()
        {
            System.out.println("Accessing from inner class");
            System.out.println("The sum of indata & outdata is " +(outdata + indata));
        }
    }
}

 class InnerclassDemo {
    public  static  void  main(String  args[])
    {
        Outer  outobj  = new Outer();
        outobj.display();
        Outer.Inner  inobj1 = outobj.new  Inner();
        inobj1.inmethod();
    }
}
```