Prepared by:bishowpaudel@hotmail.com

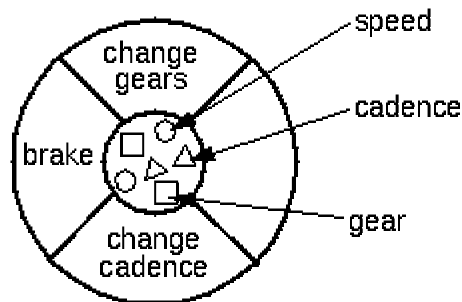
Object Oriented Concept in java

Java is a pure Object oriented Programming Language and hence the underlying structure of all java programming is class.the Object oriented approach of Java Programming can be defined through

a. Class b. Object c. Buttom Up approach d. Data Hiding /Encapsulation
e . Message Passing f. Inheritance g. Method Overloading

class :

Simply class is a collection of object .Class is prototype that defines variable and common to all object of same kind. In the real world, you often have many objects of the same kind. For example, your bicycle is just one of many bicycles in the world. Using object-oriented terminology, we say that your bicycle object is an *instance* of the class of objects known as bicycles. Bicycles have some state (current gear, current cadence, two wheels) and behavior (change gears, brake) in common. However, each bicycle's state is independent of and can be different from other bicycles.



Definition: A class is a blueprint or prototype that defines the variables and methods common to all objects of a certain kind. A class is set of attributes and behavior shared by similar object

Object

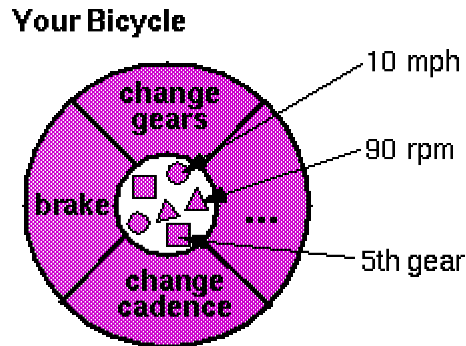
As the name *object-oriented* implies, *objects* are key to understanding object-oriented technology. You can look around you now and see many examples of real-world objects: your dog, your desk, your television set, your bicycle.

These real-world objects share two characteristics: they all have state and they all have behavior. For example, dogs have state (name, color, breed, hungry) and dogs have behavior (barking, fetching, and slobbering on your newly cleaned slacks). Bicycles have state (current gear, current pedal cadence, two wheels, number of gears) and behavior (braking, accelerating, slowing down, changing gears).

Software objects are modeled after real-world objects in that they, too, have state and behavior. A software object maintains its state in *variables* and implements its behavior with *methods*.

Definition: An object is a software bundle of variables and related methods. An Object has State ,exhibits some well defined behavior and has a unique identity. Everything that the software object knows (state) and can do (behavior) is expressed by the variables and methods within that object. A software object that modelled your real-world bicycle would have variables that indicated the bicycle's current state: its speed is 10 mph, its

pedal cadence is 90 rpm, and its current gear is the 5th gear. These variables and methods are formally known as *instance variables* and *instance methods* to distinguish them from class variables and class methods. The following figure illustrates a bicycle modeled as a software object.



Bottom up Approach
It is the Bottom Up approach as it solves easy task first and then goes to complex.
Encapsulation
Encapsulation hides the implementation details of an object and therby ,hides its complexity . The Benefits of Encapsulation
Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

- Modularity--The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else and it will still work.
- Information hiding--An object has a public interface that other objects can use to communicate with it. But the object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. You don't need to understand the gear mechanism on your bike in order to use it.

Message Passing:
Software objects interact and communicate with each other through messages. When the driver object wants the car object to accelerate, it sends the car object a message. If you want to think of messages more literally, think of two people as objects. If one person wants the other person to come closer, they send the other person a message. More accurately, they may say to the other person "Come here, please." This is a message in a very literal sense. Software messages are a little different in form, but not in theory—they tell an object what to do.Many times the receiving object needs—along with a message—more information so that it knows exactly what to do. When the driver tells the car to accelerate, the car must know by how much. This information is passed along with the message as message parameters

From this discussion, you can see that messages passing consist of three things:

I. The object to which the message is addressed (car)

II. The name of method to pass the message for the action to perform (accelerate)

III. Any parameters required by the method (15 mph)

These three components are sufficient information to fully describe a message for an object. Any interaction with an object is handled by passing a message. This means that objects anywhere in a system can communicate with other objects solely through messages

Eg car.accelerate(15);
str.charAt(8);
saving.showBalance(accountNo);

Constructor

Constructor methods are used to initialize new objects when they're created. Unlike regular methods, you can't call a constructor method by calling it directly; instead, constructor methods are called by Java automatically when you create a new object

when you use `new`, Java does three things:

- Allocates memory for the new object
- Initializes that object's instance variables, either to their initial values or to a default (`0` for numbers, `null` for objects, `false` for booleans `'\u0000'` for characters)

Calls the class's constructor method (which may be one of several methods)

Constructors look a lot like regular methods, with two basic differences:

Constructors	***Method***
1. has same name as that of class	1. has its on name
2. Constructors don't have a return type.	2. Method has a return type

3. Constructor is invoked with new Operator
`class Test{`
`Test a=new Test();}`
	3. Method is invoked by dot '.' operatror
	`str.toUpperCase("test");`

4. Used to initialized the object	4.Used to perform certain task for instance

Types Of Constructors

1) Default constructors 2)Parameterized Constructors
3)Non Parameterized Constructors 4)Copy Constructors

Default Constructors

Eg.

```
 class DefaultConstructor
{
      void printTest()
      {System.out.print("Class Bunking is a very very bad habbit");
      }
}
class  DefaultConstructorTest
{
      public static void main(String arg[])
      {DefaultConstructor  obj1=new  DefaultConstructor();    //Default
Constructor
     obj1.printTest();
      }
}
```

Parameterized Constructor

```
class ParameterizedConstructor
{String name;
  int age;
   ParameterizedConstructor(String n,int a)
      { name=n;
     age=a;
      }
    void printTest()
  {System.out.print("hi "+name+" you are "+age+" old");
  }
}
class  ParameterizedConstructorTest
{
 public static void main(String arg[])
  {ParameterizedConstructor obj1=new
  ParameterizedConstructor("RamBhorasha",21);
             //Parameterized Constructor
     obj1.printTest();
  }
}
```

NonParameterized Constructor

```
Eg.
 class NonParameterizedConstructor
 {String name;
  int age;
   NonParameterizedConstructor()
      { name="RamBhorasa";
     age=20;
      }
    void printTest()
  {System.out.print("hi "+name+" you are "+age+" old");
  }
}
class  NonParameterizedConstructorTest
{
 public static void main(String arg[])
  {NonParameterizedConstructor obj1=new
NonParameterizedConstructor();
// Non-Parameterized Constructor
     obj1.printTest();
  }
}
```

Copy Constructor

```
class CopyConstructor
{String name;
  int age;
   CopyConstructor(String n,int a)
      { name=n;
     age=a;
      }
  CopyConstructor(CopyConstructor cc)
  {name=cc.name;
     age=cc.age;
  }
    void printTest()
  {System.out.print("hi "+name+ " you are "+age+" old");
  }
}
class  CopyConstructorTest
{
 public static void main(String arg[])
  {CopyConstructor obj1=new CopyConstructor("RamBhorasha",21);
  //Parameterized  Constructor
    obj1.printTest();
```

```
       CopyConstructor obj2=new CopyConstructor(obj1);
         //Copy  Constructor
           obj2.printTest();
       }
     }
```

Method Overloading

Another powerful object-oriented technique is method overloading. Method overloading enables you to specify different types of information (parameters) to send to a method. To overload a method, you declare another version with the same name but different parameters. Method overloading is creating multiple methods with the same name but with different signatures and definitions. Java uses the number and type of arguments to choose which method definition to execute. To create an overloaded method, all you need to do is create several different method definitions in your class, all with the same name, but with different parameter lists (either in number or type of arguments). Java allows method overloading as long as each parameter list is unique for the same method name.

Eg.

```
class overload1
  {
 float area(float x,float y)          {
             return(x*y);
         }
 float area(float x)
         {
       return(3.147f*x*x);
       }
}
class overload{
  public static void main (String param[])
          {
           overload1 area1=new overload1();
           float result=area1.area(4f,5f);
    float result1=area1.area(4f);
             System.out.println("the area of the Circle is
"+result);
       System.out.println("the area of the Rectangle "+result1);
    }
}
```

Handling Errors and Exceptions

Errors are the wrong that can make a program wrong.An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash.So It is an important issue to handle and manage the possible errors .

Types of error :

Errors may broadly classified into two categories

Compile Time errors

Runtime Errors

Compile Time errors:

All types of Syntax errors detected by the java compiler at the time of the compilation of the code are called Compile times errors. Till the compiled time error are not fixed ,class file of the corresponding source code is not generated.

The most common sourse of compile time errors are:

- Missing Semicolons
- Missing (or mismatch of) brackets in class and methods
- Illegal reference to an object

- Missing of initialization of variables etc.

Run Time Errors:

Sometimes a program may compile Successfully creating the .class file but still it may not run properly of may terminate the abruptly *.Most common run time errors are:*

- Dividing by zero
- Accessing an element of an array that is out of its bounds.
- Accessing a character of a array that is out of its bounds.
- Converting invalid string to an integer.
- Trying to write a file that is *Read Only, etc..*

Exception Handling

The term exception denotes an exceptional event. It can be defined as an abnormal event that occurs during program execution and disrupts the normal flow of instruction.Exception handelling becomes a necessity when you develop application that need to take care of unexpected situations.The unexpected situation that may occurs during program execution are:

. Running Out of memory
. Resource allocation errors
. Inability to find a file
. Problem in network connectivity

Java handles exception in the Object oriented way.You can use a hierarchy of exception classes to manage runtime errors. creating an exception object and handing it to the runtime system is called ***throwing an exception.*** Those errors that can be caught by the program code is called Exception.

The common Java Exceptions are:

Arithmetic Exception:This exception is thrown when an exceptional arithmetic condition occurred. For eg. a division by zero generates such exception

ArrayIndexOutOfBoundsException :This exception is thrown when an attempt is made to access an array element beyond the index of the array. For example ,it is thrown when an array has 4 elements and you try to access the 5th element of the array.

Eg int a[] ={4,5,6,7};
int y=a[4]; generates such types of exception

StringIndexOutOfBoundsException :This exception is thrown when an attempt is made to access an element of a string beyond the index of the String. For example ,it is thrown when an String object has 4 character and you try to access the 5th character of the String object;.

Eg String a =”neap”;
char y=a.charAt(4); generates such types of exception

NullPointerException: This exceptionis thrown when an application attempts to use null where an object is required.An object that has not been allocated memeory holds a null value.The situation in which such an exception is thrown includes

Using an object without allocation memory for it.
Calling the method of a null object
Accessing or modifying the attributes of null object
Using the length of null as if it were an array

Eg: `String s;`

```
s.length();
```
generates such an exception

NumberFormatException: When we try to convert alphanumeric string into numeric String

For eg: `int n=Integer.parseInt("notalphanumericstring");`

IllegalArgumentException

This exception is thrown to indicate that an illegal argument has been passed to a method.

Catching and Handling Exceptions

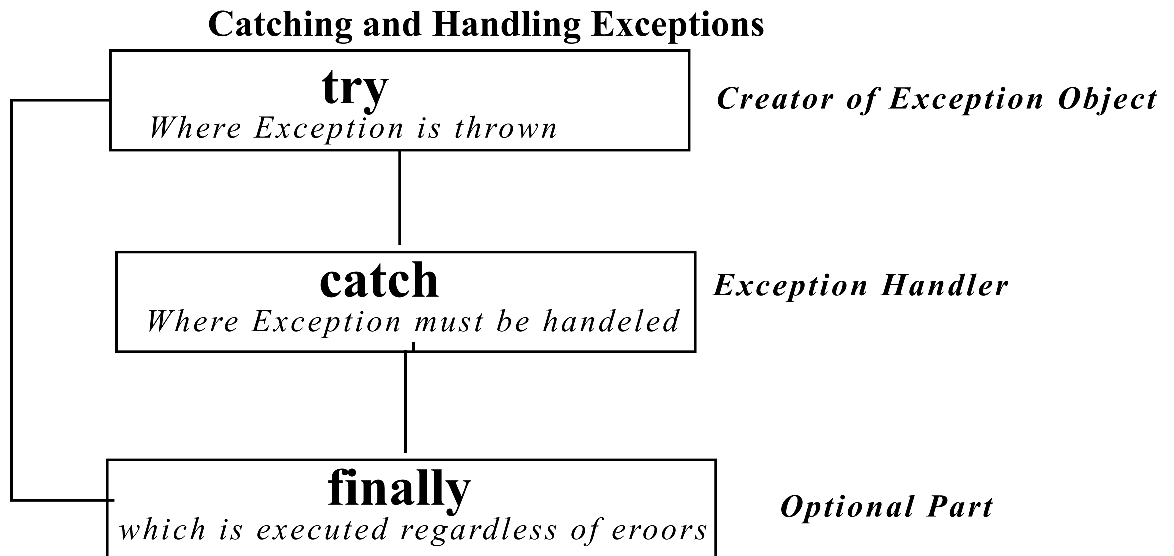


fig 2.1 Mechanism of Exceptional Handelling using try,catch ,finally

The three components of an exception handler are the try, catch, and finally blocks.

The try Block

The first step in constructing an exception handler is to enclose the statements that might throw an exceptionwithin a try block. In general, a try block looks like this: *try { Java statements }* .The segment of code labeled *Java statements* is composed of one or more legal Java statements that could throw an exception. A try statement *must* be accompanied by at least one catch block or one finally block.

The catch Block

You associate exception handlers with a try statement by providing one or more catch blocks directly after the try block:

try { . . . }
catch (. . .) { . . . }
catch (. . .) { . . . } . . .

There can be no intervening code between the end of the try statement and the beginning of the first catch statement. The general form of Java's catch statement is:

```
catch (SomeThrowableObject variableName) { Java statements }
```

The finally Block

The final step in setting up an exception handler is providing a mechanism for cleaning up the state of the method before (possibly) allowing control to be passed to a different part of the program. You do this by enclosing the cleanup code within a finally block. When an exception is raised ,the rest of the statements in the try block are ignored

.Sometimes ,it is necessary to process certain statement ,no matter whether an exception is raised or not.When a finally block is define it is guaranteed to execute regardless of whether or not an exception is thrown.
Eg

```
try{
openfile();
writefile() ;  //may cause an exception
}
catch(………)
{…..
…..
}
catch(………)
{…
…  .
}
finally
{
closefile();
}
```

The file has to be closed whether an exception is raised or not .The finally block follows the catch blocks.You can have only one finally block for an exception handler but it is mandatory to have finally block

```
try{            or            try{
     …….                         ………
     …….                         ………
   }                            }
catch(………)                    finally{
   {…..                            ………
     …..                           ………
    }                           }
catch(………)
{…………
 …………
}
..
..
..
finally
{
………
}
```

Some of the house Keeping task such as closing files and releasing system resources are done on finally block

Throw Statement

Till now you have only been catching exception are thrown by the java run time system. However ,it is possible for your program to throw an exception explicitly ,using throw statement .For example you may want to throw an exception when a user enter a wrong id or password .you can use throw statement to do so.
The throw statement takes a single argument which is an object of the exception class.
Syntax :
throw ThrowableInstance
eg: *throw new MyException*

where MyException is a defined exception class

Example
```
class MyException extends Exception
{
      MyException(String message)
      {super(message);
      }
}
class TestMyException
{public static void main(String arg[])
{
 String name;
 try
 {
      name=arg[0];
      if(name.equals("umesh"))
        {throw new MyException("u r not a valid user");
        }
      System.out.print("\n Welcome to our programm. \n Thankyou .\n");

 }
 catch(MyException e)
 {System.out.print("Sorry"+e);

 }
 catch(ArrayIndexOutOfBoundsException error)
 {
   ………
 }

}
}
```

Throws

The throws statement is used to specify the lists of exceptions that are thrown by the method.A throws clause lists the types of exception that a method might throw. All exception that a method might throw can through must be declared in the throws clause. If they are not ,a compile time error will result.

Syntax

type methodName(parameter list) throws exception_lists
  {
//body of method
  }

here a exception list is a comma separated list of the exception that a method can through

Interface

1. Interface is a named collection pf methods definitions but does not implements them (i.e collection of unimplemented method
2. Interface are used as a super class whose properties and behavior are inherited by the subclass
3. interface are used to define a protocol of behavior that must be implemented by any class in the class hierarchies
4. An interface may contain declaration .
5. Inheritance are used to standardize the method definition
6. Inheritance provides multiple inheritances.
7. class can inherit a single super class but can implement multiple interfaces.

8. Once defined ,any number of classes can implement an interface and a class can implement any number of interfaces
9. The keyword *interface* is used to define an interface and the keyword *implement* is used to implement interface
10. interface can be extended

Syntax:

access interface   name
    {
        return_type method_name1(Parameter list);
        return_type method_name1(Parameter list);
        type variable="intitalization";
        type variable;
    }

Example

```
interface A
        {
             float compute(float m,float n);
             final static float pi=3.14f;
         }
 class Disp
        {
                 void display(float value)
                 {
                  System.out.print("Area is"+value);
                 }
        }

class Rect extends Disp implements A
          {
             public float  compute(float x,float y)
                 {
                   return(x*y);
                 }

          }
 class Circle extends Disp implements A
           {
              public float  compute(float x,float y)
                 {
                    return(pi*x*x);
                 }
     }
  class MyMainclass
           {
```

```
        public static void main(String arg[])
    {
                Rect rect=new Rect();
      float area;
      area=rect.compute(2.0f,2.0f);
      rect.display(area);
      Circle circ=new Circle();
      area=circ.compute(1.0f,0f);
      circ.display(area);
    }
}
```

Type Conversion
Every expression has a type that is determined by the components of the expression. Consider the following statements.

```
 class VariavleTest
{ public static void main(String arg[])
{
        double x;
   int y=2;
   float z=2.2f;
   x=y+z;
         System.out.println("The sum is"+x);
}
}
```

Here java Automatically promotes value to a higher data type,to prevent any loss of information
Similarly in the following example

int x= 5.5/2;

The expression on the right evaluates to a decimal value and the expression on the left is an integer value, hence cannot hold a fraction. When you compile this code ,the compiler will give you error, it is because data can be lost when it is converted from a higher data types to a lower data types. The compiler requires you to typecast the assignment. To accomplish this ,type the following:

int x=(int) 5.5/2;

Type Casting refers to the conversion of one data types to different data type

Two types casting

1. Implicit Type casting: (Automatic Type Casting);Java Allows automatic type casting from lower data types to higher data types to prevent any loss of data .This is clled implicit type casting

Example *double x;*
int y=2;
float z=2.2f;
x=y+z;

An implicit casts occurs in the following order:
byte→short→ int →long →float →double
char→ int →long →float →double

2. **Explicit Type Conversion(Narrowing)**:Explicit type casting has to be done when there is a possibility of loss of data during conversion. Type conversion from higher data types to lower data type is called explicit type casting or narrow casting. This can cause loss of information.

   Eg. *float f=4.5f;*
   int i;
   i=(int)f;
   short s=(short)f;

Inner Class

Class defined within another class is inner class(nested class).Inner class has access to the members and methods of the outer class in which it is defined

Scope of the inner class is the enclosing braces of the outer class. The enclosing class doesn't have access to the member of the inner class. Inner class can be used to simplify the code nested to handle certain types of events and useful when handling event in Applet

e.g

```
    class OuterClass
      {
         int x=100;
             void test()
             {System.out.print("access by inner class");
             }

         class InnerClass
           {
                  void view()
                  { System.out.print("value of outer xis"+x);
                    test();
                  }
           }
         void innerTest()
         {     InnerClass inner=new InnerClass();
                   inner.view();
        }

      }
class InnerExample
{
      public static void main(String arg[])
      {
            OuterClass outer=new OuterClass();
            outer.innerTest();
      }
}
```

Abstract Class

It is a general class consider the following hierarchies

```
                  Shape
                    |
  ------------------------------------------------------
```

| | |
Rectangle   Circle   Hexagon

Here Shape is an Abstract class.

1. Abstact class defines the common properties and behaviors of other class.It is used as a based class to derive classes of same kinds.
2. *abstract* modifier is used to specify the abstaract class
3. Abstarct class has atleast one abstract method(with no implementation)
4. The subclass of the abstract class must implement the abstaract method
5. Abstact method have public scope.

Example

```
abstract class Shape
{  abstract float calculateArea();
}
//The abstract method Calculate area given above is inherited by the
//sub classof the shape class
 class Circle extends Shape
{
  float radious;
  Circle(float x)
  {radious=x;
  }


   float calculateArea()
   { return((radious*radious)*(22/7));
   }
}
 class Rectangle extends Shape
{
  float length,breadth;
   Rectangle(float l,float b)
   {
         length=l;
   breadth=b;
   }
   float calculateArea()
   { return(length*breadth);
   }
}
class AbstractTest
{
   public static void main(String arg[])
   {
         Circle c= new Circle(2.0f);
         Rectangle r=new Rectangle(2f,2f);
         System.out.print("the area of the circle
is"+c.calculateArea());
         System.out.print("the area of the Rectangle
is"+r.calculateArea());
   }
}
```

Reflection

Reflection is the ability of a software to analyze itself. Reflection allows us to analyze a software component and describe its capability dynamically at run time rather then

compile time. By using Reflection ,we can determine what methods ,fields and constructor a class supports. This features is provided with *java.lang.reflect* Package. The package contain an interface called Member which defines method that allow you to get information about a field ,constructor or method of that class

Example

```
import java.lang.reflect.*;

class ReflectionDemo{
  public static void main(String arg[]) throws Exception
  { int i;
       Class c=Class.forName(arg[0]);
       System.out.println("constructors");
       Constructor cons[]=c.getConstructors();
        for (i=0;i<cons.length;i++)
           System.out.println("  "+cons[i]);
          System.out.println("Methods");
       Method method[]=c.getMethods();
        for (i=0;i<method.length;i++)
           System.out.println("  "+method[i]);
         System.out.println("fields");
       Field fields[]=c.getFields();
        for (i=0;i<fields.length;i++)
           System.out.println("  "+fields[i]);
      }
}
```

Packages

Package is a collection of related classes. Package name implies directory structure.

Two types of package

1. Java Api Package or Pre-define package
2. User Defined Package

Package acts as a container for class.

Package allows access protection and name space management .

Use 'package keyword followed by package name to create user defined package .

To make that package accessible do one of the following

- Make the package directory structure relative to the current working directory
- Or Add the path of the package parent to CLASSPATH environment variable (Eg. set CLASSPATH=c:\packages)

Example:

```
 package nec.edu;
  public class First
    {
  public void view()
                {System.out.print("this is the demo of First class from nec.edu package");
                }
    }
```

Save the above package as First.java and compile it by

javac -d . First.java

```
package nec.edu.np;
public  class Second
 {
   String string;
    public Second(String str)
    {  string=str;
```

```
  }
  public void view()
        {
        System.out.print("u entered "+string);
        }
}
```

Save the above package as Second.java and compile it by

javac -d . Second.java

The following programs show the use of the package developed above

```
import nec.edu.np.Second;
import nec.edu.First;

class TestPackage
{
        public static void main(String arg[])
        {
          First fst=new First();
          fst.view();
          Second sec=new Second("\nhello i am suraj");
          sec.view();
        }
}
```

Output:

```
this is the demo of First class from nec.edu package
u entered hello i am suraj
```

Access Protection Mechanism

Java Provides four types of visibility of members variable and methods and there types of access specifier (*public,protected,private*)

- Same class
- Same Package Subclass
- Same package non-subclass
- Different package subclass
- Different package non-cubclass

	Private	No Modifiesrs	Protected	Public
Same class	Y	Y	Y	Y
Same package subclass	N	Y	Y	Y
Same package non-subclass	N	Y	Y	Y
Different package subclass	N	N	Y	Y
Different package non-subclass	N	N	N	Y

- Any Thing Declared as public can be access from any where
- Anything declared as private can be access from the member of the same class
- Any thing declared protected can be accessed either in the same class or subclass or any other class in the same package

- When a member variable or method has no access specifier it is visible to subclass and all other class in the same package.

Protection.java

```
package p1;
 public  class Protection
     {
     int n=1;
              public int pub=2;
              private int pri=3;
              protected int pro=4;
             public Protection()
              {
               System.out.print("n="+n);
               System.out.print("pubn="+pub);
             System.out.print("Pri n="+pri);
              System.out.print("protn="+pro);
         }
     }
```

Derived .java

```
package p1;
   class Derived extends Protection
                 {
                        Derived()
              {
               System.out.println("derived constructor");
              System.out.println("n="+n);
               System.out.print("pubn="+pub);
          //System.out.print("Pri n="+pri);
                                      System.out.print("protn="+pro);
     }
     }
```

SamePackage.java

```
package p1;
   class SamePackage
                         {
                        SamePackage()
              {
                        Protection p=new Protection();
               System.out.println("Same Package Constructor");
               System.out.println("n="+p.n);
               System.out.print("pubn="+p.pub);
      //System.out.print("Pri n="+p.pri);
               System.out.print("protn="+p.pro);
     }
     }
```

Protection.Java

```
package p2;
 import p1.*;
   class Protection2 extends Protection
                         {
                        Protection2()
              {
```

```
            System.out.println("Derived other Package constructor");
            //System.out.println("n="+n);
            System.out.print("pubn="+pub);
  // System.out.print("Pri n="+pri);
            System.out.print("protn="+pro);
  }
  }
```

OtherClass.java

```
package p2;
 import p1.*;
   class OtherPackage
                    {
                   OtherPackage()
           {
             Protection p=new Protection();
            System.out.println("Other Package constructor");
            //System.out.println("n="+p.n);
            System.out.print("pubn="+p.pub);
  // System.out.print("Pri n="+p.pri);
 //         System.out.print("protn="+p.pro);
  }
  }
```

The this Keyword

Sometime a method need to refer to the object that invoked it.To allow this java use this keyword .This can be used inside any method to refer to the current object.That is ,this is always a reference to the object on which the method was invoked.

```
class ThisDemo
{
  String name;
   int age;
   int height;
   ThisDemo(String name,int age,int height)
    {
     this.name=name;
     this.age=age;
     this.height=height;
   }
 void printTest()
 {
   System.out.print("hi "+name+" you are "+age+" years with
"+height+"inch height");
 }
}

class ThisDemoTest
{
   public static void main(String arg[])
   {
      ThisDemo obj=new ThisDemo("Aakash",25,5);
       obj.printTest();
   }
}
```