# UNIT-3: CLASSES, INHERITANCE, EXCEPTIONS, PACKAGES AND INTERFACES

## Syllabus :

Classes, Inheritance, Exceptions, Packages and Interfaces:

Classes: Classes fundamentals; Declaring objects; Constructors, this keyword, garbage collection.

Inheritance: inheritance basics, using super, creating multi level hierarchy, method overriding.

Exception handling: Exception handling in Java. Packages, Access Protection, Importing Packages, Interfaces.

.

**Beautiful thought**: *"You have to grow from the inside out. None can teach you, none can make you spiritual. There is no other teacher but your own soul."* — *Swami Vivekananda*

# 1. CLASSES:

**Definition**

A class is a template for an object, and defines the data fields and methods of the object. The class methods provide access to manipulate the data fields. The "data fields" of an object are often called "instance variables."

**Example Program:**

```
Program to calculate Area of Rectangle  class
Rectangle
{       int length;                  //Data Member or instance
Variables               int width;       void getdata(int x, int y)
//Method
            {
        length=x;
        width=y;
            }       int rectArea()
        //Method
            {
                    return(length*width);
            }
}

class RectangleArea
{       public static void main(String
args[])
    {
          Rectangle      rect1 =new      Rectangle();      //object      creation
              rect1.getdata(10,20); //calling methods using object with dot(.)
          int area1=rect1.rectArea();
          System.out.println("Area1="+area1);
    }
}
```

After defining a class, it can be used to create objects by instantiating the class. Each object occupies some memory to hold its instance variables (i.e. its state).

After an object is created, it can be used to get the desired functionality together with its class.

## Creating instance of a class/Declaring objects:

```
Rectangle rect1 =new Rectangle()

Rectangle rect2=new Rectangle()
```

The above two statements declares an **object rect1 and rect2 is of type Rectangle class using new operator** , this operator dynamically allocates memory for an object and returns a refernce to it.in java all class objects must be dynamically allocated.

We can also declare the object like this:

***Rectangle rect1***; // declare reference to object. ***rect1 =new***

***Rectangle()*** // allocate memory in the Rectangle object.

## The Constructors:

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

**Example:**

Here is a simple example that uses a constructor:

```
// A simple constructor. class
MyClass
{       int
x;

   // Following is the constructor
   MyClass()
  {
      x = 10;
   }
}
```

You would call constructor to initialize objects as follows:

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

## Parameterized Constructor:

Most often you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method: just declare them inside the parentheses after the constructor's name.

**Example:**

Here is a simple example that uses a constructor:

```
// A simple constructor. class
MyClass
{     int
x;

   // Following is the Parameterized constructor
   MyClass(int i )
   {
      x = 10;
   }
}
```

You would call constructor to initialize objects as follows:

```
class ConsDemo
{     public static void main(String
args[])
 {
      MyClass t1 = new MyClass( 10 );
      MyClass t2 = new MyClass( 20 );
      System.out.println(t1.x + " " + t2.x);
   }
}
```

This would produce following result:

```
10 20
```

## static keyword

The **static keyword** is used in java mainly for memory management. We may apply static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class **static variable**

**Example Program without static variable**

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at

the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each objects will have the value 1 in the count variable.

```
class Counter
{
        int count=0;//will get memory when instance is created
 Counter()
        {
                count++;
                System.out.println(count);
        }
}

Class MyPgm
{
        public static void main(String args[])
        {
                        Counter c1=new Counter();
                        Counter c2=new Counter();
                        Counter c3=new Counter();
        }
}
```

Output: 1
1
1

---

**Example Program with static variable**

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
class Counter
{
        static int count=0;//will get memory only once and retain its value

        Counter()
```

```
        {
                count++;
                System.out.println(count);
        }
 }
Class MyPgm
{
        public static void main(String args[])
        {
                Counter c1=new Counter();
                Counter c2=new Counter();
                Counter c3=new Counter();
        }
}
```

Output:1
2
3

## Static method

If you apply static keyword with any method, it is known as static method

A static method belongs to the class rather than object of a class.

A static method can be invoked without the need for creating an instance of a class.

static method can access static data member and can change the value of it.

//Program to get cube of a given number by static method **class Calculate**

```
{
        static int cube(int x)
        {
                return x*x*x;
        }
}
```

```
Class MyPgm
{
        public static void main(String args[])
        {
```

```
        //calling a method directly with class (without creation of object) int
        result=Calculate.cube(5);
        System.out.println(result);
    }
}
```

**Output:125**

## this keyword

this keyword can be used to refer current class instance variable.

If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

### *Understanding the problem without this keyword*

Let's understand the problem if we don't use this keyword by the example given below:

```
class student
{
    int id;
  String name;      student(int
id, String name)
    {
        id = id;
    name = name;
    }    void display()
    {
        System.out.println(id+" "+name);
    }
}
```

```
class MyPgm
{   public static void main(String
args[])
```

```
        {
                student s1 = new student(111,"Anoop");
                student s2 = new student(321,"Arayan");
        s1.display();
                s2.display();
        }
}
```

Output: 0 null
0 null

In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

***Solution of the above problem by this keyword***

```
//example of this keyword    class
Student
{
    int   id;            String   name;
student(int id,  String name)
        {
      this.id = id;            this.name =
name;
        }
         void display()
        {
                System.out.println(id+" "+name);
        }
}
class MyPgm
{  public static void main(String args[])
        {
                Student   s1   =   new   Student(111,"Anoop");
                Student s2 = new Student(222,"Aryan");
                s1.display();                    s2.display();
        }   }
```

Output111 Anoop
222 Aryan

## Inner class

It has access to all variables and methods of **Outer** class and may refer to them directly. But the reverse is not true, that is, **Outer** class cannot directly access members of **Inner** class.

One more important thing to notice about an **Inner** class is that it can be created only within the scope of **Outer** class. Java compiler generates an error if any code outside **Outer** class attempts to instantiate **Inner** class.

***Example of Inner class*** **class**

```
Outer
{
        public void display()
        {
                Inner in=new Inner();
in.show();
        }

        class Inner
        {
                public void show()
                {
                        System.out.println("Inside inner");
                }
        }
}
class Test
{
        public static void main(String[] args)
        {
                Outer ot=new Outer();
ot.display();
        }
}
```

**Output:**

Inside inner

## Garbage Collection

In Java destruction of object from memory is done automatically by the JVM. When there is no reference to an object, then that object is assumed to be no longer

needed and the memories occupied by the object are released. This technique is called **Garbage Collection**. This is accomplished by the JVM.

### *Can the Garbage Collection be forced explicitly?*

No, the Garbage Collection cannot be forced explicitly. We may request JVM for **garbage collection** by calling **System.gc()** method. But this does not guarantee that JVM will perform the garbage collection.

---

### *Advantages of Garbage Collection*

1. Programmer doesn't need to worry about dereferencing an object.
2. It is done automatically by JVM.
3. Increases memory efficiency and decreases the chances for memory leak.

---

### *finalize() method*

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held. To handle such situation **finalize()** method is used. **finalize()** method is called by garbage collection thread before collecting object. It's the last chance for any object to perform cleanup utility.

Signature of **finalize()** method protected

```
void finalize()
{
 //finalize-code
}
```

*gc() Method* **gc()** method is used to call garbage collector explicitly. However **gc()** method does not guarantee that JVM will perform the garbage collection. It only requests the JVM for garbage collection. This method is present in **System** and **Runtime** class.

*Example for gc() method* public class Test

```
{
   public static void main(String[] args)
   {
        Test t = new Test();
        t=null;
        System.gc();
   }
   public void finalize()
   {
        System.out.println("Garbage Collected");
 }
}
```

**Output :**

Garbage Collected

## Inheritance:

As the name suggests, inheritance means to take something that is already made. It is one of the most important features of Object Oriented Programming. It is the concept that is used for **reusability** purpose.

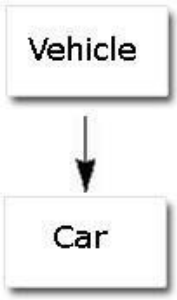
Inheritance is the mechanism through which we can derive classes from other classes.

The derived class is called as child class or the subclass or we can say the extended class and the class from which we are deriving the subclass is called the base class or the parent class.

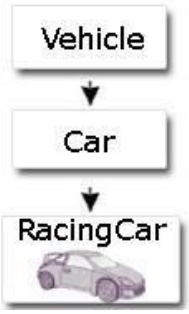
To derive a class in java the keyword **extends** is used. The following kinds of inheritance are there in java.

**Types of Inheritance** 1. **Single level/Simple Inheritance** 2. **Multilevel Inheritance** 3. **Multiple Inheritance (Java doesn't support Multiple inheritance but we can achieve this through the concept of Interface.)**

**Pictorial Representation of Simple and Multilevel Inheritance**




**Simple Inheritance**




**Multilevel Inheritance**

**Single level/Simple Inheritance**

When a subclass is derived simply from its parent class then this mechanism is known as simple inheritance. In case of simple inheritance there is only a sub class and its parent class. It is also called single inheritance or one level inheritance.

## Example

```
class A
{     int x;        int y;
      int get(int p, int q)
      {
            x=p;
            y=q;
            return(0);
      }
      void Show()
      {
           System.out.println(x);
      }
}
  class B extends A
  {
            public static void main(String args[])
            {
                  A        a        =        new        A();
  a.get(5,6);
                  a.Show();
  }
```

```
        void display()
        {
            System.out.println("y");        //inherited "y" from class A
        }
    }
```

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the extends keyword, followed by the name of the class to inherit from:

```
class A
{
  }   class B extends A                    //B is a subclass of super
class A.
{

}
```

**Multilevel Inheritance**

When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance.

The derived class is called the subclass or child class for it's parent class and this parent class works as the child class for it's just above ( parent ) class.

Multilevel inheritance can go up to any number of level.

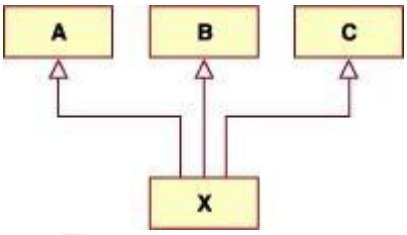
```
class A
{
      int x;
int y;        int get(int
p, int q)
      {
x=p;
y=q;
return(0);
      }
      void Show()
      {
          System.out.println(x);
      }
}

class B extends A
{       void
      Showb()
      {
           System.out.println("B");
      }
}

class C extends B
{
      void display()
      {
           System.out.println("C");
      }        public static void
main(String args[])
      {
              A a = new A();
a.get(5,6);
              a.Show();
      }
}
OUTPUT
5
```

### Multiple Inheritance

The mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritance. Java does not support multiple inheritance but the multiple inheritance can be achieved by using the interface.




Here you can derive a class from any number of base classes. Deriving a class from more than one direct base class is called multiple inheritance.

**Java does not support multiple Inheritance**

In Java Multiple Inheritance can be achieved through use of Interfaces by implementing more than one interfaces in a class.

## Super keyword

The super is java keyword. As the name suggest super is used to access the members of the super class. It is used for two purposes in java.

The first use of keyword super is to access the hidden data variables of the super class hidden by the sub class.

Example: Suppose class A is the super class that has two instance variables as int a and float b. class B is the subclass that also contains its own data members named a and b. then we can access the super class (class A) variables a and b inside the subclass class B just by calling the following command. **super.member;**

Here member can either be an instance variable or a method. This form of super most useful to handle situations where the local members of a subclass hides the members of a super class having the same name. The following example clarifies all the confusions.

**Example:**

```
 class
A
{
        int         a;
float b;         void
Show()
        {
               System.out.println("b in super class:  " + b);
        }
}
class B extends A
{         int a;
float b;
        B( int p, float q)
        {
a = p;
             super.b = q;
        }
        void Show()
        {
             super.Show();
             System.out.println("b in super class:  " + super.b);
             System.out.println("a in sub class:    " + a);
       }
}

 class Mypgm
{
          public static void main(String[] args)
          {
                        B subobj = new B(1, 5);
subobj.Show();                          }
}
OUTPUT
b in super class:  5.0 b
in super class:  5.0 a
in sub class:    1
```

**Use of super to call super class constructor:** The second use of the keyword super in java is to call super class constructor in the subclass. This functionality can be achieved just by using the following command.

**super(param-list);**

Here parameter list is the list of the parameter requires by the constructor in the super class. super must be the first statement executed inside a super class constructor. If we want to call the default constructor then we pass the empty parameter list. The following program illustrates the use of the super keyword to call a super class constructor.

**Example:**

```
class A
{          int a;
int b;
           int c;
         A(int p, int q, int r)
        {
a=p;
b=q;
c=r;
        }
}
class B extends A
{
            int d;
           B(int l, int m, int n, int o)
           {
super(l,m,n);
d=o;
           }
void Show()
```

```
        {
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                 System.out.println("c = " + c);
                 System.out.println("d = " + d);
         }
}
class Mypgm
{          public static void main(String
args[])
        {
                B b = new B(4,3,8,7);                    b.Show();
        }
 }
OUTPUT
a = 4 b
= 3 c =
8  d =
7
```

## Method Overriding

Method overriding in java means a subclass method overriding a super class method.

Superclass method should be non-static. Subclass uses extends keyword to extend the super class. In the example **class B** is the sub class and **class A** is the super class. **In overriding methods of both subclass and superclass possess same signatures**. Overriding is used in modifying the methods of the super class. In overriding return types and constructor parameters of methods should match.

Below example illustrates method overriding in java.

**Example:**

```
class A
{
            int i;
           A(int a, int b)
           {
i = a+b;
           }
void add()
          {
               System.out.println("Sum of a and b is: " + i);
           }
}
class B extends A
{
            int j;
            B(int a, int b, int c)
            {
super(a, b);
j = a+b+c;
           }
void add()
          {
              super.add();
              System.out.println("Sum of a, b and c is: " + j);
          }
}
class MethodOverriding
{           public static void main(String
args[])
         {
                 B b = new B(10, 20, 30);
b.add();
         }
}
```

**OUTPUT**

```
Sum of a and b is: 30
Sum of a, b and c is: 60
```

## Method Overloading

Two or more methods have the same names but different argument lists. The arguments may differ in type or number, or both. However, the return types of overloaded methods can be the same or different is called **method overloading**. An example of the method overloading is given below:

**Example:**

```
class MethodOverloading
{               int add( int a,int
b)
          {
return(a+b);
          }
          float add(float a,float b)
          {
return(a+b);
          }
          double add( int a, double b,double c)
          {
return(a+b+c);
          }
}
class MainClass
{
          public static void main( String arr[] )
          {
                MethodOverloading mobj = new MethodOverloading ();
                System.out.println(mobj.add(50,60));
                System.out.println(mobj.add(3.5f,2.5f));
                System.out.println(mobj.add(10,30.5,10.5));
          }
}
OUTPUT
110
6.0
51.0
```

## Abstract Class

**abstract keyword** is used to make a class abstract.

Abstract class can't be instantiated with new operator.

We can use abstract keyword to create an abstract method; an abstract method doesn't have body.

If classes have abstract methods, then the class also needs to be made abstract using abstract keyword, else it will not compile.

Abstract classes are used to provide common method implementation to all the subclasses or to provide default implementation. **Example Program:**

```
abstract Class AreaPgm
{
        double dim1,dim2;
        AreaPgm(double x, double y)
        {
dim1=x;
dim2=y;
        }
        abstract double area();
}
class rectangle extends AreaPgm
{
        rectangle(double a,double b)
        {
                super(a,b);
        }
        double area()
        {
                System.out.println("Rectangle Area");
        return dim1*dim2;
        }
}
class triangle extends figure
{
        triangle(double x,double y)
        {
                super(x,y);
        }
        double area()
        {
                System.out.println("Traingle Area");
        return dim1*dim2/2;
        }
}
class MyPgm
{
```

```
        public static void main(String args[])
        {

                AreaPgm a=new AreaPgm(10,10); // error, AreaPgm is a abstract class.

                rectangle r=new rectangle(10,5);
                System.out.println("Area="+r.area());

                triangle t=new triangle(10,8);
                AreaPgm          ar;
                ar=obj;
                System.out.println("Area="+ar.area());
        }
}
```

## final Keyword In Java

The **final keyword** in java is used to restrict the user. The final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

**1) final variable:** If you make any variable as final, you cannot change the value of final variable(It will be constant).

**Example:** There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike
{       final int speedlimit=90;//final variable
void run()
        {
                speedlimit=400;
         }
}
Class MyPgm
```

```
{  public static void main(String args[])
        {
                Bike obj=new  Bike();
               obj.run();
         }
}
```

**Output: Compile Time Error**

---

**2) final method:** If you make any method as final, you cannot override it.

**Example:** **class Bike**

```
{       final void run()
        {
                System.out.println("running");
        }
}    class Honda extends
Bike
{    void run()
        {
                System.out.println("running safely with 100kmph");
        }  }
Class MyPgm
{       public static void main(String args[])
        {
                Honda honda= new Honda();   honda.run();
        }
}
```

**Output: Compile Time Error**

**3) final class:** If you make any class as final, you cannot extend it.

**Example:**

```
final class Bike
{
}

class Honda extends Bike
{
          void run()
```

```
        {
                System.out.println("running safely with 50kmph");
         }
}

Class MyPgm
{

         public static void main(String args[])
        {
                 Honda honda= new Honda();
honda.run();
        }
}
```

**Output:Compile Time Error**

## Exception handling:

**Introduction**

An **Exception**, It can be defined as an abnormal event that occurs during program execution and disrupts the normal flow of instructions. The abnormal event can be an error in the program.

Errors in a java program are categorized into two groups:

1. **Compile-time errors** occur when you do not follow the syntax of a programming language.
2. **Run-time errors** occur during the execution of a program.

**Concepts of Exceptions**

An exception is a run-time error that occurs during the exception of a java program.

Example: If you divide a number by zero or open a file that does not exist, an exception is raised.

In java, exceptions can be handled either by the java run-time system or by a userdefined code. When a run-time error occurs, an exception is thrown. The unexpected situations that may occur during program execution are:

- Running out of memory
- Resource allocation errors
- Inability to find files
- Problems in network connectivity

**Exception handling techniques:**

Java exception handling is managed via five keywords they are:

1. try:
2. catch.
3. throw.
4. throws.
5. finally.

**Exception handling Statement Syntax :**

Exceptions are handled using a try-catch-finally construct, which has the Syntax.

```
try
{
      <code>
}
catch (<exception type1> <parameter1>)
{
      // 0 or more<statements>
 }
finally
{
      // finally block<statements>
}
```

1. **try Block:** The java code that you think may produce an exception is placed within a try block for a suitable catch block to handle the error.

If no exception occurs the execution proceeds with the finally block else it will look for the matching catch block to handle the error.

Again if the matching catch handler is not found execution proceeds with the finally block and the default exception handler throws an exception.

2. **catch Block**: Exceptions thrown during execution of the try block can be caught and handled in a catch block. On exit from a catch block, normal execution continues and the finally block is executed (Though the catch block throws an exception).

3. **finally Block:** A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed. Generally finally block is used for freeing resources, cleaning up, closing connections etc.

**Example**:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest
{              public static void main(String
args[])
          {
                    try
                    {
                              int a[] = new int[2];
                              System.out.println("Access element three :" + a[3]);
                    }
catch(ArrayIndexOutOfBoundsException e)
                    {
                              System.out.println("Exception thrown  :" + e);
                    }
                    System.out.println("Out of the block");
         }
}
```

This would produce following result:

```
Exception thrown: java.lang.ArrayIndexOutOfBoundsException: 3 Out of the block
```

**Multiple catch Blocks:**

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
  // code
}

catch(ExceptionType1 e1)
{
  //Catch block
}


catch(ExceptionType2 e2)
{
  //Catch block
```

```
}
catch(ExceptionType3 e3)
{
  //Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try.

**Example: Here is code segment showing how to use multiple try/catch statements.**

```
class Multi_Catch
{           public static void main (String args
[])
        {

                try
                 {
                        int a=args.length;
                        System.out.println("a="+a);
int b=50/a;
int c[]={1}
                 }
                 catch (ArithmeticException e)
                 {
                       System.out.println ("Division by zero");
                 }

                 catch (ArrayIndexOutOfBoundsException e)
                 {
                        System.out.println (" array index out of bound");
                 }
     }
}
```

**OUTPUT:**

```
Division by zero array index
out of bound
```

## Nested try Statements

Just like the multiple catch blocks, we can also have multiple try blocks. These try blocks may be written independently or we can nest the try blocks within each other, i.e., keep one try-catch block within another try-block.
The program structure for nested try statement is:

**Syntax**

```
try
{
    // statements
    // statements

try
  {
     // statements
     // statements
  }
  catch (<exception_two> obj)
  {
     // statements
  }

  // statements
  // statements
}
catch (<exception_two> obj)
{
  // statements
}
```

Consider the following example in which you are accepting two numbers from the command line. After that, the command line arguments, which are in the string format, are converted to integers.

If the numbers were not received properly in a number format, then during the conversion a NumberFormatException is raised otherwise the control goes to the next try block. Inside this second try-catch block the first number is divided by the second number, and during the calculation if there is any arithmetic error, it is caught by the inner catch block.

**Example**

```
class Nested_Try
{
   public static void main (String args [ ] )
   {       try                    {
int a = Integer.parseInt (args [0]);
int b = Integer.parseInt (args [1]);
int quot = 0;
                  try
                  {
                        quot = a / b;
                        System.out.println(quot);
                  }
                  catch (ArithmeticException e)
                  {
                        System.out.println("divide by zero");
                  }
           }
           catch (NumberFormatException e)
           {
                 System.out.println ("Incorrect argument type");
           }
     }
}
```

**The output of the program is: If the arguments are entered properly in the command prompt like:**

**OUTPUT:**

```
java Nested_Try 2 4 6
 4
```

If the argument contains a string than the number:

**OUTPUT**

```
java  Nested_Try 2 4  aa
 Incorrect argument type
```

If the second argument is entered zero:

**OUTPUT**

```
java Nested_Try 2 4 0  divide
by zero
```

**throw Keyword:**

**throw keyword** is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown.

Program execution stops on encountering **throw** statement, and the closest catch statement is checked for matching type of exception.

**Syntax : throw** *ThrowableInstance*

---

**Creating Instance of Throwable class**

There are two possible ways to get an instance of class Throwable,

1. Using a parameter in catch block.
2. Creating instance with **new** operator.

```
new NullPointerException("test");
```

This constructs an instance of NullPointerException with name test.

---

**Example demonstrating throw Keyword class**

```
Test
{       static void avg()
          {
                  try
                   {
                           throw new ArithmeticException("demo");
                    }
                    catch(ArithmeticException e)
                    {
                           System.out.println("Exception caught");
```

```
                }
        }
        public static void main(String args[])
        {
                avg();
        }
}
```

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement.

---

## throws Keyword

Any method capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions to handle. A method can do so by using the **throws** keyword.

**Syntax :** *type method_name(parameter_list)* **throws** *exception_list*

```
        {
                //definition of method
        }
```

**NOTE :** It is necessary for all exceptions, except the exceptions of type **Error** and **RuntimeException**, or any of their subclass.

---

**Example demonstrating throws Keyword**

```
class Test
{
        static void check() throws ArithmeticException
        {
    System.out.println("Inside check function");          throw
new ArithmeticException("demo");
        }
        public static void main(String args[])
        {       try
                {
```

```
                check();
            }
            catch(ArithmeticException e)
            {
                System.out.println("caught" + e);
            }
        }
}
```

## finally

The finally clause is written with the try-catch statement. It is guaranteed to be executed after a catch block or before the method quits.

**Syntax**

```
try
{
   // statements
}
```

```
catch (<exception> obj)
{
   // statements
} finally
{
   //statements
}
```

Take a look at the following example which has a catch and a finally block. The catch block catches the ArithmeticException which occurs for arithmetic error like divide-by-zero. After executing the catch block the finally is also executed and you get the output for both the blocks.

**Example:**

```
class Finally_Block
{     static void division
( )
   {
       try
       {                              int
num = 34, den = 0;
int quot = num / den;
       }
       catch(ArithmeticException e)
       {
             System.out.println ("Divide by zero");
       }
finally
       {
             System.out.println ("In the finally block");
      }
   }
}
class Mypgm
{
   public static void main(String args[])
   {
          Finally_Block   f=new   Finally_Block();
f.division();
    }
}
```

**OUTPUT**

```
Divide by zero
In the finally block
```

## Java's Built in Exceptions

Java defines several exception classes inside the standard package **java.lang**.

The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java **Unchecked RuntimeException**.

| Exception | Description |
|---|---|
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast. |

| | |
|---|---|
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalMonitorStateException | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException | Environment or application is in incorrect state. |
| IllegalThreadStateException | Requested operation not compatible with current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. |
| NumberFormatException | Invalid conversion of a string to a numeric format. |

| | |
|---|---|
| SecurityException | Attempt to violate security. |
| StringIndexOutOfBounds | Attempt to index outside the bounds of a string. |
| UnsupportedOperationException | An unsupported operation was encountered. |

Following is the list of **Java Checked Exceptions** Defined in java.lang.

| Exception | Description |
|---|---|
| ClassNotFoundException | Class not found. |
| CloneNotSupportedException | Attempt to clone an object that does not implement the Cloneable interface. |
| IllegalAccessException | Access to a class is denied. |
| InstantiationException | Attempt to create an object of an abstract class or interface. |
| InterruptedException | One thread has been interrupted by another thread. |
| NoSuchFieldException | A requested field does not exist. |

| | |
|---|---|
| NoSuchMethodException | A requested method does not exist. |

## Creating your own Exception Subclasses

Here you can also define your own exception classes by extending **Exception**. These exception can represents specific runtime condition of course you will have to throw them yourself, but once thrown they will behave just like ordinary exceptions.

When you define your own exception classes, choose the ancestor carefully. Most custom exception will be part of the official design and thus checked, meaning that they extend Exception but not RuntimeException.

**Example: Throwing User defined Exception**

```
public class MyException extends Exception
{
    String msg = "";
int marks=50;
     public MyException()
    {
    }
    public MyException(String str)
    {
super(str);
    }
    public String
    {          if(marks <= 40)
msg = "You have failed";
if(marks > 40)
            msg = "You have Passed";
        return msg;

    }
}
```

```
class test
{          public static void main(String args[])
      {              test t = new test();              t.dd();
}
      public void add()
      {
          try              {
             int i=0;              if( i<40)
             throw new MyException();
          }
          catch(MyException ee1)
          {
              System.out.println("Result:"+ee1);
          }
     }
}
OUTPUT
Result: You have Passed
```

## Chained Exception

Chained exceptions are the exceptions which occur one after another i.e. most of the time to response to an exception are given by an application by throwing another exception.

Whenever in a program the first exception causes an another exception, that is termed as **Chained Exception**. Java provides new functionality for chaining exceptions.

Exception chaining (also known as "nesting exception") is a technique for handling the exception, which occur one after another i.e. most of the time

is given by an application to response to an exception by throwing another exception.

Typically the second exception is caused by the first exception. Therefore chained exceptions help the programmer to know when one exception causes another.

The **constructors** that support chained exceptions in **Throwable** class are:

Throwable initCause(Throwable)
Throwable(Throwable)
Throwable(String, Throwable)
Throwable getCause()

## Packages in JAVA

A **java package** is a group of similar types of classes, interfaces and subpackages.

Package in java can be categorized in two form, ○ built-in package and ○ userdefined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

### Advantage of Java Package

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

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple
{
    public static void main(String args[])
    {
        System.out.println("Welcome to package");
    }
}
```

### How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*; 2. import package.classname;
3. fully qualified name.

### *1) Using packagename.**

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

**Example of package that import the packagename.***

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

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

Output:Hello

---

### *2) Using packagename.classname*

If you import package.classname then only declared class of this package will be accessible.

**Example of package by import package.classname**

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

//save by B.java
package  mypack;
import pack.A;

class B
{
          public static void main(String args[])
          {
                    A obj = new A();
obj.msg();
          }
}
Output:Hello
```

### *3) Using fully qualified name*

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

**Example of package by import fully qualified name**

```
//save by A.java   package
pack;
```

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

//save by B.java
package mypack;
class B
{
          public static void main(String args[])
         {
                  pack.A obj = new pack.A();//using fully qualified name
obj.msg();
         }
}
```

Output:Hello

## Access Modifiers/Specifiers

The access modifiers in java specify accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

**1) private access modifier**

The private access modifier is accessible only within class.

**2) default access modifier**

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package.

**3) protected access modifier**

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

**4) public access modifier**

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

**Understanding all java access modifiers by a simple table.**

| Access Modifier | within class | within package | outside package by subclass only | outside package |
|---|---|---|---|---|
| Private | Y | N | N | N |
| Default | Y | Y | N | N |
| Protected | Y | Y | Y | N |
| Public | Y | Y | Y | Y |

## Interface in java

An **interface in java** is a blueprint of a class. It has static final variables and abstract methods.

The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface does not contain method body.

It is used to achieve abstraction and multiple inheritance in Java.

It cannot be instantiated just like abstract class.

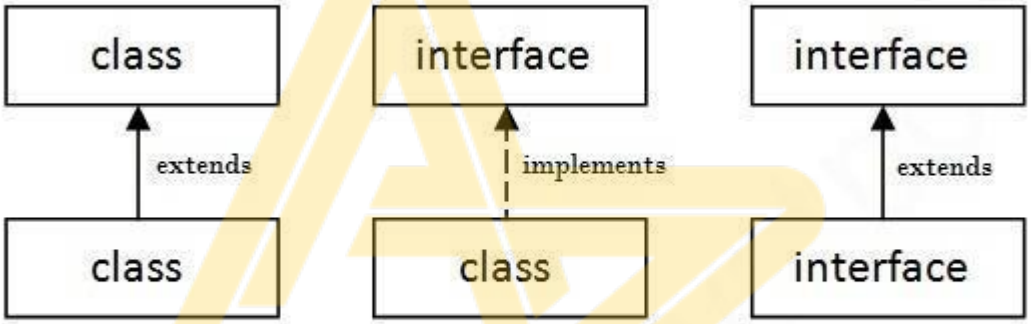
Interface fields are public, static and final by default, and methods are public and abstract.

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.

***Understanding relationship between classes and interfaces***

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




**Example 1**

In this example, Printable interface has only one method, its implementation is provided in the Pgm1 class.

```
interface printable
{    void print();
}

class Pgm1 implements printable
{
        public void print()
        {
                System.out.println("Hello");
```

```
    }  }
class IntefacePgm1
{
    public static void main(String args[])
    {
        Pgm1 obj = new Pgm1 ();
        obj.print();
    }  }
```

**Output:**

Hello

**Example 2**

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In real scenario, interface is defined by someone but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.

```
//Interface declaration: by first user    interface
Drawable
{
    void draw();
}
//Implementation: by second user    class
Rectangle implements Drawable
{
    public void draw()
    {
        System.out.println("drawing rectangle");
    }
}

class Circle implements  Drawable
{
    public void draw()
    {
        System.out.println("drawing circle");
    }
}
```

```
//Using interface: by third user  class
TestInterface1
{
        public static void main(String args[])
        {
                //In real scenario, object is provided by method e.g. getDrawable()
                Drawable d=new Circle();

                d.draw();
        }
}
```

Output: drawing circle

## Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.


**Multiple Inheritance in Java**

### Example

```
interface Printable
{   void print();
}

interface Showable
{   void show();
}
class Pgm2 implements Printable,Showable
```

```
{
        public void print()
        {
                System.out.println("Hello");
        }

        public void show()
        {
                System.out.println("Welcome");
        }
}

Class InterfaceDemo
{ public static void main(String args[])
        {
                Pgm2 obj = new Pgm2 ();   obj.print();    obj.show();
        }
}
```

Output: Hello Welcome

**Multiple inheritance is not supported through class in java but it is possible by interface, why?**

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity.

But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

**Example**

```
interface Printable
{  void print();
}
 interface Showable
{  void print();
}
 class InterfacePgm1 implements Printable, Showable
{
        public void print()
```

```
        {
                System.out.println("Hello");
        }
}
class InterfaceDemo
{
        public static void main(String args[])
        {
                InterfacePgm1  obj = new  InterfacePgm1 ();   obj.print();
         }
}
```

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

## Interface inheritance

A class implements interface but one interface extends another interface .

```
interface Printable
{  void print();
}
interface  Showable  extends  Printable
{
        void show();
}
class InterfacePgm2  implements  Showable
{
        public void print()
        {
                System.out.println("Hello");
        }

        public void show()
        {
                System.out.println("Welcome");
```

```
        }
Class InterfaceDemo2
{
        public static void main(String args[])
        {
                InterfacePgm2  obj = new  InterfacePgm2 ();   obj.print();
                obj.show();
         }
}
```

Output:

Hello
Welcome

## Program to implement Stack

```
public class StackDemo
{   private static final int capacity = 3;
int arr[] = new int[capacity];
    int top = -1;
    public void push(int pushedElement)
   {   if (top < capacity - 1)
            {
            top++;
                arr[top] = pushedElement;
                System.out.println("Element " + pushedElement  + " is pushed to Stack !");
            printElements();
             }
            else
             {
                    System.out.println("Stack Overflow !");
            }
     }      public
void pop()
   {
            if (top >= 0)
             {   top--;
                    System.out.println("Pop operation done !");
            }
            else


            {
```

```
                System.out.println("Stack Underflow !");
            }
        }       public void
printElements()     {
                if (top >= 0)
                {
                    System.out.println("Elements in stack :");
                    for (int i = 0; i <= top; i++)
                    {
                        System.out.println(arr[i]);
                    }
                }
        }
    }
     class MyPgm
    {
            public static void main(String[] args)
            {
                StackDemo stackDemo = new StackDemo();
                stackDemo.pop();
                stackDemo.push(23);
                stackDemo.push(2);
                stackDemo.push(73);
                stackDemo.push(21);
                stackDemo.pop();    stackDemo.pop();
```

**Output:**

```
Stack Underflow !
Element 23 is pushed to Stack !
Elements in stack :
23
Element 2 is pushed to Stack !
Elements in stack :
23
2
Element 73 is pushed to Stack !
Elements in stack :
23
2
73
Stack Overflow !
Pop operation done !
Pop operation done !
Pop operation done !
Stack Underflow !
```

```
                stackDemo.pop();
                stackDemo.pop();
            }
    }
```

## Questions

1. Distinguish between Method overloading and Method overriding in JAVA, with suitable examples. (Jan 2014) 6marks
2. What is super? Explain the use of super with suitable example (Jan 2014) 6marks
3. Write a JAVA program to implement stack operations. (Jan 2014) 6marks
4. What is an Exception? Give an example for nested try statements? (Jan 2013) 6 Marks
5. WAP in java to implement a stack that can hold 10 integers values (Jan 2013) 6 Marks
6. What is mean by instance variable hiding ?how to overcome it? (Jan 2013) 04 Marks
7. Define exception .demonstrate the working of nested try blocks with suitable example? (Dec 2011)08Marks
8. Write short notes on (Dec 2011)04Marks

   i) Final class ii) abstract class

9. Write a java program to find the area and volume of a room. Use a base class rectangle with a constructor and a method for finding the area. Use its subclass room with a constructor that gets the value of length and breadth from the base class and has a method to find the volume. Create an object of the class room and obtain the area and volume. (Jan-2006) 8Marks
10. Explain i) Instance variables ii) Class Variables iii) Local variables (Jan-2009) 06 marks
11. Distinguish between method overloading and method overriding? How does java decide the method to call? (Jan-2008-8Marks)

12. Explain the following with example.

i) Method overloading ii) Method overriding (jun-2006) 8Marks

13. Write a java program to find the distance between two points whose coordinates are given. The coordinates can be 2dimensional or 3dimensional (for comparing the distance between 2D and a 3D point, the 3D point, the 3D x and y components must be divided by z).

    Demonstrate method overriding in this program.

    (May-2007)10 marks

14. What is an interface? Write a program to illustrate multiple inheritance using interfaces. (Jan-2010) 8Marks
15. Explain packages in java.
16. What are access specifiers? Explain with an example.
17. With an example explain static keyword in java.
18. Why java is not support concept of multiple inheritance? Justify with an example program.
19. Write a short note on:
    1. this keyword
    2. super keyword
    3. final keyword
    4. abstract
20. Illustrate constructors with an example program