MODULE-2

PART-1

CLASSES, OBJECTS & METHODS

WHAT IS OOPS?

Object-Oriented Programming (OOP) is the term used to describe a programming approach based on **objects** and **classes**. Anything we wish to represent in a Java program must be encapsulated in a class that defines the state and behavior of the basic program components known as objects. The object-oriented paradigm allows us to organise software as a collection of objects that consist of both data and behaviour. This is in contrast to conventional functional programming practice that only loosely connects data and behaviour.

The object-oriented programming approach encourages:

- Modularization: where the application can be decomposed into modules.
- Software re-use: where an application can be composed from existing and new modules.

An object-oriented programming language generally supports five main features:

- Classes
- Objects
- Classification
- Polymorphism
- Inheritance

Class in Java

A class is a user-defined data type with a template that serves to define its properties. We can create variables of that type that has common properties which are objects.

A class in java can contain:

- **data member**
- **method**
- **constructor**
- **block**
- **class and interface**

A class can contain any of the following variable types.

- **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the objects are instantiated and therefore associated with the objects. They take different values for each object. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword. They are global to a class and belong to the entire set of objects that class creates.

A class can have any number of methods to access the value of various kinds of methods.

Syntax to declare a class:

```
1.  class <class_name>{
2.      data member;
3.      method;
4.  }
```

Object in Java

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of intangible object is banking system.

An object has three characteristics:

- **state :** represents data (value) of an object.
- **behavior :** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity :** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance (result) of a class.

Creating an Object

An object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class −

- **Declaration** − A variable declaration with a variable name with an object type.
- **Instantiation** − The 'new' keyword is used to create the object.
- **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Simple Example of Object and Class

In this example, a Student class has two data members, id and name. We are creating the object of the Student class by new keyword and printing the objects value.

```
class Student1
{
  int id;  //data member (also instance variable)
  String name; //data member(also instance variable)

 public static void main(String args[])
{
  Student1 s1=new Student1();          //creating an object of Student
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Output: 0 null

Constructor in Java

Constructor in java is a special type of method that is used to initialize the object. Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

There are basically two rules defined for the constructor.

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

Types of java constructors

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Java Default Constructor: A constructor that has no parameter is known as default constructor.

Syntax of default constructor:

```
<class_name>()
{       }
```

Example of default constructor

```
class Student3
{
  int id;
  String name;

 Student3()
{  }

void display()
 {
System.out.println(id+" "+name);
}

public static void main(String args[])
{
 Student3 s1=new Student3();
 Student3 s2=new Student3();
```

```
 s1.display();
 s2.display();
}
}
```
Output : 0 null

0 null

Note :

- If there is no constructor in a class, compiler automatically creates a default constructor.
- Default constructor provides the default values to the object like 0, null etc. depending on the type.

Java parameterized constructor

A constructor that has parameters is known as parameterized constructor.

Parameterized constructor is used to provide different values to the distinct objects.

```
class Student4
{
   int id;
   String name;

Student4()
{  }

   Student4(int i,String n)
  {
   id = i;
   name = n;
    }
   void display()
  {
         System.out.println(id+" "+name);
  }
    public static void main(String args[])
   {
   Student4 s1 = new Student4(111,"Karan");
```

```
    Student4 s2 = new Student4(222,"Aryan");
    s1.display();
    s2.display();
   }
}
```

Output: 111 Karan
222 Aryan

Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

```
class Student5
{
   int id;
   String name;
   int age;

Student5()
{  }

   Student5(int i,String n)
  {
    id = i;
    name = n;
   }

  Student5(int i, String n, int a)
 {
    id = i;
    name = n;
    age=a;
   }
    void display()
   {
System.out.println(id+" "+name+" "+age);
```

```
  }
   public static void main(String args[]){
   Student5 s1 = new Student5(111,"Karan");
   Student5 s2 = new Student5(222,"Aryan",25);
   s1.display();
   s2.display();
  }
}
```

Output: 111 Karan 0
222 Aryan 25

Method Overloading in Java

If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**.

Method overloading **increases the readability of the program**.

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

Example of Method Overloading by changing the no. of arguments

```
class Calculation
{
  void sum(int a,int b)
 {                  System.out.println(a+b);
 }
  void sum(int a,int b,int c)
  {       System.out.println(a+b+c);
  }

  public static void main(String args[])
 {
  Calculation obj=new Calculation();
  obj.sum(10,10,10);
  obj.sum(20,20);
  }
```

```
 }
```

Example of Method Overloading by changing data type of argument

```
class Calculation2
{
  void sum(int a,int b)
  {       System.out.println(a+b);
  }
  void sum(double a,double b)
  {       System.out.println(a+b);
  }

  public static void main(String args[])
 {
  Calculation2 obj=new Calculation2();
  obj.sum(10.5,10.5);
  obj.sum(20,20);
  }
}
```

Static Members

A Static Member is the member that belongs to the class as a whole rather than the objects created from the class. They are defined as follows:

```
static int count;                  // Static variable
static int max(int x, int y);      // Static method
```

Static variables are used when we want to have a variable common to all instances of a class. They are also available for use by other classes. Methods that are of general utility but do not directly affect an instance of that class are usually declared as class methods. Java libraries contain a large number of class methods. Static methods are called using class names. We can define our own static methods as follows:

Program to illustrate **Static Methods**

```
class MathOperation
{
        static int mul(int x, int y)
        {       return x*y;     }
```

```
        static int divide(int x, int y)
        {       return x/y;     }

}

class Operations
{
        public static void main(String args[])
        {       int a=MathOperation.mul(4,5);
                int b=MathOperation.divide(5,2);
                System.out.println(a+"   "+b);
 }
}
```

Output : 20 2

Static methods have several restrictions:

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

Program to illustrate **Default Constructor , Parameterized Constructor, Constructor Overloading and Static Members**

```
class Student
{
  int id;  //data member (also class variable)
  String name;    //data member(also instance variable)
  int age=18;    //data member(also instance variable)
static int count;
  Student()
  {count++;
   }

  Student(int x,String str)
  {id=x;
```

```
  name=str;
  count++; }

  Student(int x,String str,int y)
  {id=x;
   name=str;
   age=y;
   count++; }
     void display()
          {
          System.out.println(id+" "+name+" "+age+"    "+count);
            }

 public static void main(String args[])
{
  Student s1=new Student();   //creating an object of Student
          s1.display();

  Student s2=new Student(1,"abc");   //creating an object of Student
          s2.display();

  Student s3=new Student(2,"xyz",20);         //creating an object of Student
          s3.display();


 }
}
```

Output : 0 null 18 1
1 abc 18 2
2 xyz 20 3

Static Block

Static block is mostly used for changing the default values of static variables.This block gets executed when the class is loaded in the memory.
A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

Program to illustrate StaticBlock

```
class StaticBlock{
          int n;
  static int sn;
  static int num;
  static String mystr;

          void getdata(int y)
          {n=y;
            System.out.println("Value of n="+n);          }

  static void data(int x)
          {sn=x;
            System.out.println("Value of sn="+sn);       }

 static{
    num = 25;
    mystr = "Static keyword in Java";
  }

   public static void main(String args[])
  {  StaticBlock obj=new StaticBlock();
          obj.getdata(5);
          data(10);
    System.out.println("Value of num="+ ++num);
    System.out.println("Value of mystr="+mystr);
  }
}
```

Output: Value of n=5
Value of sn=10
Value of num=26
Value of mystr=Static keyword in Java

Program to illustrate Multiple Static Blocks

```
class MultipleStaticBlock
{
   static int num;
  static String mystr;
  //First Static block
  static{
    System.out.println("Static Block 1");
    num = 68;
    mystr = "Block1";
  }
```

```
  //Second static block
  static{
     System.out.println("Static Block 2");
     num = 98;
     mystr = "Block2";
  }
  public static void main(String args[])
  {
     System.out.println("Value of num="+num);
     System.out.println("Value of mystr="+mystr);
   }
}
```

Output:

```
Static Block 1
Static Block 2
Value of num=98
Value of mystr=Block2
```

PART-2

INHERITANCE

SUBCLASS

Java classes can be reused by creating new classes that are built upon existing ones. The mechanism of deriving a new class from an old one is called inheritance. The old class is called base class or super class or parent class and the new one is called the subclass or derived class or child class.

Subclasses inherit all the variables and methods of their parent classes.

Syntax of Defining a Subclass

class Subclassname **extends** Superclassname

```
{
  //methods and fields
}
```

The keyword **extends** indicates that the properties of Superclassname are extended to Subclassname. The Subclass will now contain its own variables and methods as well as those of the Superclass. This helps to add some properties to an existing class without actually modifying it.

SUPER KEYWORD

The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever an instance of subclass is created, an instance of parent class is created implicitly i.e. referred by super reference variable.

Usage of java super Keyword

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.
3. super is used to invoke immediate parent class method.

```
// super keyword in subclass is used to refer immediate parent class instance variable.
class Vehicle{
  int speed=50;
}
```

```
 class Bike4 extends Vehicle
{           int speed=100;
            void display()
            {       System.out.println(super.speed);//will print speed of Vehicle now
                    System.out.println(speed);//will print speed of Bike4 now
            }
            public static void main(String args[])
           {        Bike4 b=new Bike4();
                    b.display();
           }
}                   OUTPUT :    50
                                100
```

SUBCLASS CONSTRUCTOR

A subclass constructor is used to construct the instance variables of both the subclass and the superclass. The subclass constructor uses the keyword super to invoke the constructor method of the superclass. The call to the superclass constructor must appear as the first statement within the subclass constructor and the parameters in the **super** call must match the order and type of the instance variable declared in the superclass.

Example program

```
class Room
{           int l,b;
            Room(int x, int y)
            {           l=x;b=y;        }
            int area()
           {        return(l*b);     }
}
class Bedroom extends Room          //Inheriting Room
{          int h;
           Bedroom(int x,int y, int z)
             {      super(x,y);                             // Pass values to superclass
                    h = z;
           }
           int volume()
           {        return(l*b*h);  }
}
```

```
class    subclasscons
{ public static void main(String args[])
  {       Bedroom room1=new Bedroom(10,15,20);
          int a=room1.area();              //superclass method
          int v=room1.volume();                    //subclass method
          System.out.println("Area= "+a);
          System.out.println("Volume="+v);
   }
}
```

OUTPUT: Area=150
Volume=3000

SINGLE INHERITANCE

When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **Super class** of B and B would be a **Sub class** of A.

Example Program on Single Inheritance

```
class A
{          public void methodA()
          {         System.out.println("Base class method");
          }
}
class B extends A
{         public void methodB()
          {         System.out.println("Child class method");
          }
 }
class single
{         public static void main(String args[])
          {         B obj = new B();
                    obj.methodA(); //calling super class method
                    obj.methodB(); //calling local method
          }
}
```

OUTPUT:

Base class method

Child class method

MULTILEVEL INHERITANCE

Multilevel inheritance refers to a mechanism in Object Oriented technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A.

Example Program on Multilevel Inheritance

```
 class X
{           public void methodX()
        {           System.out.println("Class X method");       }
}
class Y extends X
{       public void methodY()
        {       System.out.println("class Y method");         }
}
class Z extends Y
{           public void methodZ()
        {       System.out.println("class Z method");                  }
}
class multilevel
{
  public static void main(String args[])
  {         Z obj = new Z();
            obj.methodX(); //calling grand parent class method
            obj.methodY(); //calling parent class method
            obj.methodZ(); //calling local method
 }
}
```

OUTPUT :

Class X method

Class Y method

Class Z method

HIERARCHICAL INHERITANCE

In this type of inheritance one class is inherited by many **sub classes**.

Example Program on Hierarchical Inheritance

```
class A
{       int a=10;       }
class B extends A
{       int b=30;
        void display()
        {       System.out.println("The values are:"+a+"   "+b);      }
}
```

```
class C extends A
{       int c=40;
        void show()
        {       System.out.println("The values are:"+a+"   "+c);      }
}
class hierarchicaldemo
{       public static void main(String args[])
        {       B obj1=new B();
                obj1.display();
                C obj2=new C();
                obj2.show();
        }
}
```

OUTPUT:

The values are:10 30

The values are:10 40

Method Overriding: Design a vehicle class hierarchy in Java, and develop a program to demonstrate Polymorphism.

```
import java.io.*;
class Vehicle
{       String regno;
        int model;
        Vehicle(String r, int m)
        {       regno=r;
                model=m;
        }
        void display()
        {       System.out.println("registration no:"+regno);
                System.out.println("model no:"+model);
         }
}

class Twowheeler extends Vehicle
{       int noofwheel;
        Twowheeler(String r,int m,int n)
        {       super(r,m);
                noofwheel=n;
        }
        void display()
        {       System.out.println("Two wheeler tvs");
                super.display();
```

```
                System.out.println("no of wheel" +noofwheel);
        }
}

class Threewheeler extends Vehicle
{       int noofleaf;
        Threewheeler(String r,int m,int n)
        {       super(r,m);
                noofleaf=n;
        }
        void display()
        {       System.out.println("three wheeler auto");
                super.display();
                System.out.println("no of leaf" +noofleaf);
        }
}

class Fourwheeler extends Vehicle
{       int noofleaf;
        Fourwheeler(String r,int m,int n)
        {       super(r,m);
                noofleaf=n;
        }
        void display()
        {       System.out.println("four wheeler car");
                super.display();
                System.out.println("no of leaf" +noofleaf);
        }
}

public class Vehicledemo
{       public static void main(String arg[])
        {       Twowheeler t1;
                Threewheeler th1;
                Fourwheeler f1;
                t1=new Twowheeler("TN74 12345", 1,2);
                th1=new Threewheeler("TN74 54321", 4,3);
                f1=new Fourwheeler("TN34 45677",5,4);
                t1.display();
                th1.display();
                f1.display();
        }
}
```

OUTPUT :

```
Two wheeler tvs
registration no:TN74 12345
```

model no:1
no of wheel2
three wheeler auto
registration no:TN74 54321
model no:4
no of leaf3
four wheeler car
registration no:TN34 45677
model no:5
no of leaf4

Final Keyword In Java

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

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that has no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

1) Java final variable

A variable declared as final, cannot change its value(It will be constant).

Example of final variable

There is a final variable speedlimit, trying to change the value of this variable is not possible because final variable once assigned a value can never be changed.

```
class FinalVariable
{       final int speedlimit=90;//final variable
        void run()
        {       speedlimit=400;     }
}
class Bike1 extends FinalVariable
{       int speedlimit=80;
        public static void main(String args[])
        {       Bike1 obj=new  Bike1();
```

```
                obj.run();
                System.out.println("obj"+ obj.speedlimit);
        }
}
```

OUTPUT:

```
FinalVariable.java:5: error: cannot assign a value to final variable speedlimit
  speedlimit=400;
  ^
1 error
```

2) Java final method

Final method is inherited but cannot be overridden.

```
class finalmethod
{       final void demo()
        {       System.out.println("finalmethod Class Method");       }
}

class finalmethoddemo extends finalmethod
{       void demo()
          {     System.out.println("finalmethoddemo Class Method");          }
          public static void main(String args[])
          {           finalmethoddemo obj= new finalmethoddemo();
                      obj.demo();
          }
}
```

OUTPUT:

```
finalmethoddemo.java:11: error: demo() in finalmethoddemo cannot override demo()
 in finalmethod
   void demo()
        ^
  overridden method is final
1 error
```

3) Java final class

Any class declared as final, cannot be extended.

```
final class XYZ
{       void demo()
          {     System.out.println("My Method");       }
}
class finalclassdemo extends XYZ
{     public static void main(String args[])
        {             finalclassdemo obj= new finalclassdemo();
                      obj.demo();
          }
}
```

OUTPUT:

```
finalclassdemo.java:8: error: cannot inherit from final XYZ
class finalclassdemo extends XYZ
                     ^
1 error
```

ABSTRACT METHODS AND CLASSES

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, then the class itself must be declared abstract, as in:

```
public abstract class GraphicObject {
  // declare fields
  // declare nonabstract methods
  abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Example of Abstract class and Abstract Methods

```
abstract class Bike
{       Bike()
          {   System.out.println("bike is created");     }
          abstract void run();
          void changeGear()
          {         System.out.println("gear changed");          }
}

class Honda extends Bike
{       void run()
        {       System.out.println("running safely..");       }
}
class abstractdemo1
{       public static void main(String args[])
        {       Honda obj = new Honda();
                obj.run();
                obj.changeGear();
        }
}
```

OUTPUT:

bike is created

running safely..

gear changed

VISIBILITY CONTROL

It is possible to inherit all the members of a class by a subclass using the keyword extends. The variables and methods of a class are visible everywhere in the program. However, it may be necessary in some situations where we may want them to be not accessible outside. We can achieve this in Java by applying visibility modifiers to instance variables and methods. The visibility modifiers are also known as access modifiers. Access modifiers determine the accessibility of the members of a class.

Java provides the following visibility/access modifiers:

- Public
- Friendly/Package(default)
- Private
- Protected

They provide different levels of protection as described below.

Public Access: Any variable or method is visible to the entire class in which it is defined. But, to make a member accessible outside with objects, we simply declare the variable or method as public. A variable or method declared as public has the widest possible visibility and accessible everywhere.

Friendly Access (Default): When no access modifier is specified, the member defaults to a limited version of public accessibility known as "friendly" level of access. The difference between the "public" access and the "friendly" access is that the public modifier makes fields visible in all classes, regardless of their packages while the friendly access makes fields visible only in the same package, but not in other packages.

Protected Access: The visibility level of a "protected" field lies in between the public access and friendly access. That is, the protected modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages

Private Access: Private fields have the highest degree of protection. They are accessible only with their own class. They cannot be inherited by subclasses and therefore not accessible in subclasses. In the case of overriding public methods cannot be redefined as private type.

Private protected Access: A field can be declared with two keywords Private and Protected together. This gives a visibility level in between the "protected" access and "private" access. This modifier makes the fields visible in all subclasses regardless of what package they are in. Remember, these fields are not accessible by other classes in the same package.

The following table summarizes the visibility provided by various access modifiers.

Access modifier →	public	protected	friendly	private protected	private
Own class	✓	✓	✓	✓	✓
Sub classin same package	✓	✓	✓	✓	×
Other classesIn same package	✓	✓	✓	×	×
Sub classin other package	✓	✓	×	✓	×
Other classesIn other package	✓	×	×	×	×

PART-3

MANAGING ERRORS AND EXCEPTIONS

INTRODUCTION

The exception handling in java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained. An exception (or exceptional event) is a problem that arises during the execution of a program. When an "Exception" occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

EXCEPTION TYPES

Every exception type is a subclass of the built-in class **Throwable.** The top of Exception class hierarchy is **Throwable.** The two subclasses that partition throwable in to two various parts are Exception and Error.

Exception class is used for handling exceptional conditions that user programs should catch. **Error** class defines exceptions that are not expected to be caught under normal circumstances by the user program. This chapter will not be dealing with exceptions of type **Error.**

TYPES OF ERRORS: COMPILE TIME AND RUN TIME ERRORS

Errors are broadly classified into Compile time and Run time Errors.

Compile time Errors: All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, .class file is not generated. So, it is necessary to fix all the errors for successful compilation and running of the program.

Examples of Compile time errors

1. Missing semicolons
2. Missing brackets in classes and methods
3. Use of undeclared variables
4. Incompatible types in assignments / initialization
5. Missing of double quotes in strings
6. Use of = in place of == operator

Run time Errors: Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow.

Examples of Runtime errors

1 Dividing an integer by zero
2 Accessing an element that is out of the bounds of an array
3 Trying to store a value into an array of an incompatible class or type
4 Passing a parameter that is incompatible type
5 Trying to illegally change the state of a thread
6 Converting invalid string to a number

When such errors are encountered, Java typically generates an error message and aborts the program.

EXCEPTIONS

An exception is a condition that is caused by a run time error in a program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it. If the exception object is not caught and handled properly, the interpreter will display an error message and terminate the program.

The exception handling in java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained. Its purpose is to provide a means to detect and report an exceptional circumstance to take an appropriate action. The mechanism incorporates a separate error handling code that performs the following tasks:

1. Find the problem(Hit the exception)
2. Inform that an error has occurred(Throw the exception)
3. Receive the error information(Catch the exception)
4. Take corrective actions(Handle the exception)

The error handling code consists of two segments, one to detect errors and throw exceptions and the other to catch exceptions and to take appropriate actions.

TYPES OF EXCEPTIONS

Exceptions are categorized into two types:

1. Checked Exceptions: These exceptions are explicitly handled in the code itself with the help of try-catch blocks. Checked Exceptions are extended from the java.lang.Exception class.
2. Unchecked Exceptions: These exceptions are not essentially handled in the program code, instead the JVM handles such exceptions. Unchecked Exceptions are extended form the java.lang.RuntimeException class.

Functionality of Checked and Unchecked Exceptions is same, the difference lies only in the way they are handled.

SYNTAX OF EXCEPTION HANDLING CODE

```
try
{
//code that may throw exception
}
catch(Exception_class_Name ref)
{       }
```

Example Program:

```
public class Testtrycatch
{   public static void main(String args[])
  {    try
        {                int data=50/0;            }
         catch(ArithmeticException e)
        {        System.out.println(e); }
 System.out.println("rest of the code...");
 }
}
```

OUTPUT:
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...

MULTIPLE CATCH STATEMENTS

Multiple Catch statements are used to handle different exceptions when different tasks are to be performed. When an exception in a try block is generated, the Java treats the multiple catch statements like cases in a Switch statement. The first statement whose parameter matches the exception object will be executed, and the remaining statements will be skipped.

Example Program:

```
public class TestMultipleCatchBlock
{  public static void main(String args[])
  {    try
        {     int a[]=new int[5];
                a[5]=30/0;
          }
  catch(ArithmeticException e)
        {        System.out.println("task1 is completed");     }
```

```
 catch(ArrayIndexOutOfBoundsException e)
        {       System.out.println("task 2 completed");       }
  catch(Exception e)
        {       System.out.println("common task completed");       }
  System.out.println("rest of the code...");
 }
}
```

Output:task1 completed
rest of the code...

USING FINALLY STATEMENT

Java finally block is a block that can handle an exception that is not caught by any of the previous catch statements. When a finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown. So, it can be used *to execute important code* such as closing connection, stream etc. Java finally block follows try or catch block.

Example Program:

```
class TestFinallyBlock
{   public static void main(String args[])
        {       try
                {       int data=25/5;
                        System.out.println(data);
                }
                catch(NullPointerException e)
                {       System.out.println(e); }
                Finally
                {       System.out.println("finally block is always executed");       }
          System.out.println("rest of the code...");
  }
}
```

Output:5
finally block is always executed
rest of the code...

THROWING OUR OWN EXCEPTIONS

The Java throw keyword is used to explicitly throw an exception. We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

Syntax: throw exception;
Eg: throw new IOException("sorry device error);
Throw new ArithmeticException();

Example Program: Common Java Exception

```
class TestCommonException
{       public static void main(String args[])
        {       int c=10,d=0;
        try
        {       if(d==0)
                {       throw new ArithmeticException("Division by zero is not possible");
                }
        }
        catch(ArithmeticException h)
        {       System.out.println(h);          }
        finally
        {       System.out.println("exception cleared");      }
        }
}
```

Output:

```
java.lang.ArithmeticException: Division by zero is not possible
exception cleared
```

Example Program: User defined Exception

```
class myexception extends Exception
{       myexception(String s)
        {       super(s);       }
}
class TestUserException
{       public static void main(String args[])
        {       int age=Integer.parseInt(args[0]);
          try
          {     if(age>20)
                {       throw new myexception("your age exceeds 20");     }
                if(age<20)
                {       throw new myexception("your age is under 20");     }
```

```
                else
                {       System.out.println("eligible for job");          }   }
            catch(myexception e)
            {       System.out.println("you are not eligble for job");
                System.out.println(e);
            }
          }
}
```

Output :

```
C:\Users\admin>java TestUserException      25
you are not eligble for job
myexception: your age exceeds 20

C:\Users\admin>java TestUserException      20
eligible for job

C:\Users\admin>java TestUserException      15
you are not eligble for job
myexception: your age is under 20
```