Unit-3

Topics:

Inheritance, types of inheritance, super keyword, final keyword, overriding and abstract class. **Interfaces**, **Creating the packages**, using packages, importance of CLASSPATH and java.lang package. **Exception handling**, importance of try, catch, throw, throws and finally block, user- defined exceptions, **Assertions**.

Inheritance Basics

Inheritance is the process by which one class acquires the properties of another class. This is important because it supports the concept of hierarchical classification. The class that is inherited is called ***super class***. The class that is inheriting the properties is called ***subclass***. Therefore the subclass is the specialized version of the super class. The subclass inherits all the instance variable and methods using the **extends** keyword, and adds its own code. Super class is also known as Parent class, and Base class. The sub class is also known as Child class and Derived class.

Aggregation is the process of making an object combining number of other objects. The behavior of the bigger object is defined by the behavior of its component objects. For example, cars contain number of other components such as engine, clutches, breaks, starter etc.

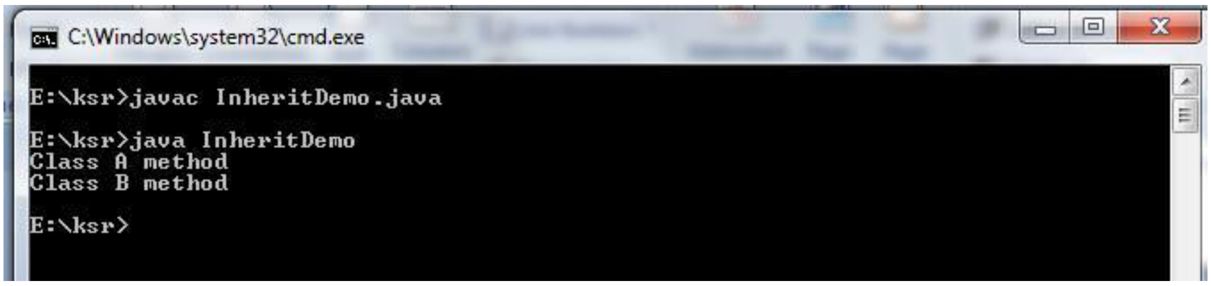
To inherit a class, we simply incorporates the definition one class into another class using the **extends** keyword.

Syntax:

```
class A
{
}
class B extends A
{
}
```

(a) Write a JAVA program to implement Single Inheritance (Lab Exercise – 5 (a))

```
class A
{
        //body of the class A
        void methodA()
        {
                System.out.println("Class A method");
        }
}
class B extends A
{
        //body of the class B
        void methodB()
        {
                System.out.println("Class B method");
        }
}
class InheritDemo
{
        public static void main(String args[])
        {
        B b=new B();
        b.methodA();
        b.methodB();
        }
}
```



```
E:\ksr>javac InheritDemo.java

E:\ksr>java InheritDemo
Class A method
Class B method

E:\ksr>
```

<u>Private member access and Inheritance</u>

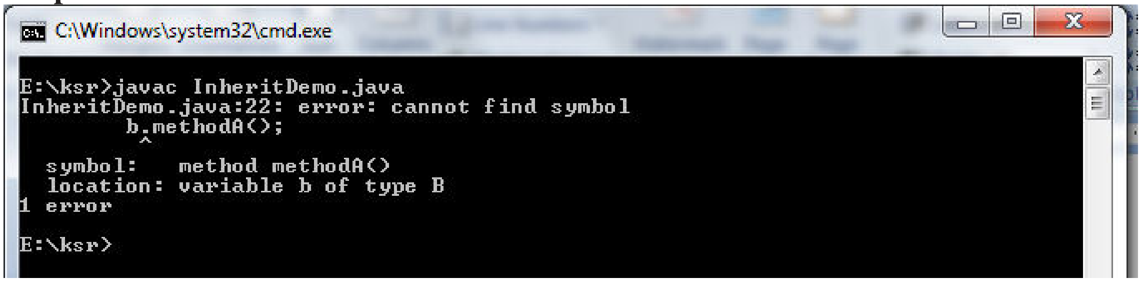
Although sub class has the right to access members of the super class, but it cannot access private members of the super class.

Example Program:

```
class A
{
        //body of the class A
        private void methodA()     //private member of the super class
        {
                System.out.println("Class A method");
        }
}
class B extends A
{
        //body of the class B
        void methodB()
        {
                System.out.println("Class B method");
        }
}
class InheritDemo
{
        public static void main(String args[])
        {
        B b=new B();
        b.methodA();   //generates error
```

```
		b.methodB();
		}
}
```

Output:



```
E:\ksr>javac InheritDemo.java
InheritDemo.java:22: error: cannot find symbol
        b.methodA();
         ^
  symbol:   method methodA()
  location: variable b of type B
1 error

E:\ksr>
```

Another example:

// Create a superclass.

```
class A
{
	int i; // public by default
	private int j; // private to A
	void setij(int x, int y)
	{
		i = x;
		j = y;
	}
}
// A's j is not accessible here.
class B extends A
 {	int total;
	void sum()
	{
		total = i + j; // ERROR, j is not accessible here
	}
}
class Access
{
public static void main(String args[])
{
	B b = new B();   // creating the object b
	b.setij(10, 12);
	b.sum();   // Error, private members are not accessed
	System.out.println("Total is " + subOb.total);
}
}
```

Note: This program will not compile because the reference to **j** inside the **sum( )** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

Types of Inheritance

There are five different types of Inheritances:

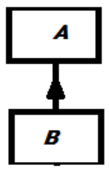
i. Single Inheritance

ii. Multiple Inheritance iii.

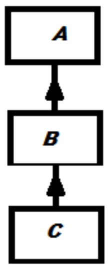
Multilevel Inheritance iv.

Hierarchical Inheritance v.

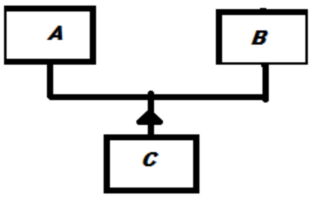
Hybrid Inheritance



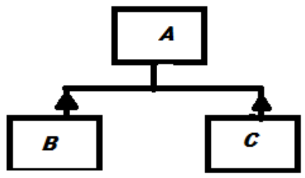
Single Inheritance



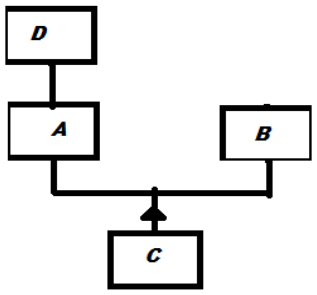
Multilevel Inheritance



Multiple Inheritance



Hierarchical Inheritance



Hybrid Inheritance

Single Inheritance:
In this one class acquires the properties from another class and adds its own code to it. Example Program is shown below.

A more practical example to illustrate the Inheritance (Single Inheritance)

```
class Box
{
        double width,height,depth;
        Box(double w,double h,double d)
        {
                width=w;
                height=h;
                depth=d;
        }
}
class BoxVolume extends Box
{
        BoxVolume(double w,double h,double d)
        {
                super(w,h,d); //calling the super class constructor
        }
        void boxVolume()
        {
                double v=width*height*depth;
                System.out.println("The volume of the Box is "+v);
        }
}
class BoxTest
{
        public static void main(String args[])
        {
                BoxVolume bv=new BoxVolume(12.3,13.2,14.3);
                bv.boxVolume();
        }
}
```

Output:

```
E:\ksr>javac BoxTest.java
E:\ksr>java  BoxTest
The volume of the Box is 2321.7480000000005
```

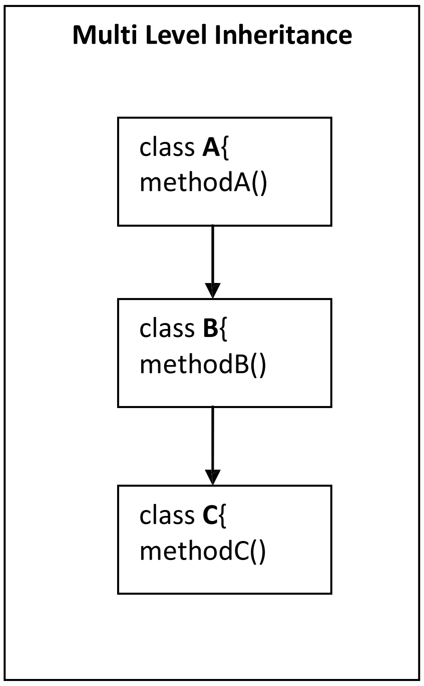
Multiple Inheritance:

In this one class acquires the properties from two or more classes at a time and adds its own code to it. This is not supported by Java among the classes, but is supported among the Interfaces. We will see this example in the Interface section.

Multilevel Inheritance:

In this one class acquires the properties from another class, which in turn has acquired the properties from another class. Hence, in this there are many levels in the process of Inheritance. The example program is below.

```
class A
        {
        public void methodA()
        {
                System.out.println("class A method");
        }
        }
class B extends A
        {
        public void methodB()
        {
                System.out.println("class B method");
        }
        }
class C extends B
        {
        public void methodC()
        {
                System.out.println("class C method");
        }
        }
class MLI
{
  public static void main(String args[])
  {
```



```
        C obj = new C();
        obj.methodA(); //calling grand parent class method
        obj.methodB(); //calling parent class method
        obj.methodC(); //calling child class method
 }
}
```

Output:

```
E:\ksr>javac MLI.java
E:\ksr>java  MLI
class A method
class B method
class C method
```

Hierarchical Inheritance:
In this two or more classes will acquire the properties from only one same class and add their own code. The example program is given below.

```
class A
        {
        public void methodA()
        {
                System.out.println("class A method");
        }
        }
class B extends A
        {
        public void methodB()
        {
                System.out.println("class B method");
        }
        }
class C extends A
        {
        public void methodC()
        {
                System.out.println("class C method");
        }
        }
class MLI
{
  public static void main(String args[])
  {
        B b=new B();
        C c = new C();
        System.out.println("calling the methodA() and methodB() with B's object");
        b.methodA();   // calling the methodA() and methodB() with B's object
        b.methodB();
        System.out.println("calling the methodA() and methodC() with C's object");
        c.methodA();  // calling the methodA() and methodC() with C's object
        c.methodC();
  }
}
```

Output:

```
E:\ksr>javac MLI.java
E:\ksr>java  MLI
calling the methodA() and methodB() with B's object
class A method
class B method
calling the methodA() and methodC() with C's object
class A method
class C method
```

Hybrid Inheritance:
In this different types of Inheritances are used to acquire the properties from number of classes to a single class. We will discuss this example in the interface section.

Note: Java Supports Single and Multilevel Inheritances between classes and Multiple Inheritance among the Interfaces.

A Superclass Reference Variable Can be assigned a Subclass Object

Any sub class reference variable can be assigned to the super class reference variable. When a reference to a subclass object is assigned to a super class reference variable, we will have access only to those parts of the object defined by the super class.
For example,

```
class Box
{
        double width,height,depth;
}
class Boxweight extends Box
{
        double weight;
        Boxweight(double x,double y,double z,double z)
        {
                width=x; height=y; depth=z; weight=a;
        }
        void volume()
        {
                System.out.println("The volume is :"+(width*height*depth));
        }
}
//main class
class BoxDemo
{
        //creating super class object
        public static void main(String args[])
        {
        Box b=new Box();
        //creating the subclass object
        Boxweight bw=new Boxweight(2,3,4,5);
        bw.volume();
        //assigning the subclass object to the superclass object
        b=bw;    // b has been created with its own data, in which weight is not a member
        System.out.println ("The weight is :"+b.weight);
        }
}
```

Super keyword

There are three uses of the ***super*** keyword.

i. super keyword is used to call the super class ***constructor***
ii. super keyword is used to access the super class ***methods***
iii. super keyword is used to access the super class ***instance variables***.

Using the super to call the super class constructor
A subclass can call the constructor of the super class by the using the super keyword in the following form:

```
super(arg_list);
```

Here, the arg_list, is the list of the arguments in the super class constructor. This must be the first statement inside the subclass constructor. For example,

```
// BoxWeight now uses super to initialize its Box attributes.
class Box
{
        double width,height,depth;
        //superclass constructor
        Box(double x,double y,double z)
        {
                width=x;height=y;depth=z;
        }
class BoxWeight extends Box
{
double weight; // weight of box
// initialize width, height, and depth using super()
        BoxWeight(double w, double h, double d, double m)
        {
                super(w, h, d); // call super class constructor
                weight = m;
        }
}
```

Using the super to access the super class members (methods or instance variable)

The second form of **super** acts somewhat like **this**, except that it always refers to the super class of the subclass in which it is used. This usage has the following general form:

super.*member;*

Here, *member* can be either a method or an instance variable.
This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the super class. Consider this simple class hierarchy:

Write a JAVA program give example for "super" keyword. (Lab Exercise - 6 (a))

```
class A
{
        int i;
        void show()
        {
                System.out.println(" i in A is: "+i);
        }
}
// Create a subclass by extending class A.
class B extends A
{
        int i; // this i hides the i in A
        B(int a, int b)
        {
                super.i = a; // i in A
                i = b; // i in B
        }
        void show()
        {
                super.show();  //calling the super class method
                System.out.println("i in A : " + super.i); // accessing the super class variable
                System.out.println("i in B : " + i);
        }
}
class UseSuper
{
public static void main(String args[])
{
B subOb = new B(1, 2);
subOb.show();
}
}
```

Output:
```
E:\ksr>javac UseSuper.java
E:\ksr>java UseSuper
i in A is: 1
i in A : 1
i in B : 2
```

Note: In the above program the super and sub classes have common names for variables and methods. When we want to execute the super class method, at run time actually the sub class method is executed, because of ***method overriding***. To overcome this problem and ***hide*** the sub class members from the super class members, the keyword super is used with help of .(dot) operator along with member.

final keyword

The "***final***" keyword is used for the following purposes.

- to declare constants
- to prevent method overriding
- to prevent inheritance.

When a variable is declared as final through the program its value should not be changed by the program statement. If any modification is done on the final variable, that can lead to error, while compiling the program.

Example program: demonstrating (1) and (2)

FinalTest.java

```
import java.io.*;
class A
{
        final int MAX=100;   // (1) constant declaration
final   void disp()   //(2) prevents overriding
        {
                // MAX++ or MAX-- operations are illegal
        System.out.println("The super class disp method MAX is:"+MAX);

        }
}
class B extends A
{
        void disp()
        {
        System.out.println("The SUB class disp method :");
        }

}
class FinalTest
{
        public static void main(String args[])
        {
                B b=new B();
                b.disp();
        }
}
```

Output:

```
E:\ksr>javac FinalTest.java
FinalTest.java:14: disp() in B cannot override disp() in A; overridden method is
 final
      void disp()
         ^
1 error
```

Explanation:

In the above program, the super class method is declared as final, and hence the sup class cannot override this. So we should not redefine the same method in the sub class. If we do so, it leads to an error. If we do the same program without disp() method in the sub class, it will produce the following output.

Example Program: removing the disp() method in the sub class

```
import java.io.*;
class A
{
        final int MAX=100;   // (1) constant declaration
        final    void disp()   //(2) prevents overriding
        {
                // MAX++ or MAX-- operations are illegal
        System.out.println("The super class disp method MAX is:"+MAX);

        }
}
class B extends A
{       /*void disp()    multiple comments
        {
        System.out.println("The SUB class disp method :");
        }*/
}
class FinalTest
{       public static void main(String args[])
        {       B b=new B();
                b.disp();
        }
}
```

Output:

```
E:\ksr>javac FinalTest.java
E:\ksr>java  FinalTest
The super class disp method MAX is:100
```

<u>Example program: Demonstrating "final" to Prevent Inheritance</u>

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its

methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.
Here is an example of a **final** class:

```
final class A
{
        // ...
}
// The following class is illegal.

class B extends A
{
        // ERROR! Can't subclass A
        // ...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

Method overriding

In the Inheritance, when a method in the sub class has the same name as the super class method name and signature, then the method in the sub class is executed. The method in the sub class is said to be override the method in the super class. The super class version of the method is hidden. This is called "***method overriding***".

Example Program on method overriding

```
class A
{
        void disp()
        {
                System.out.println("Method of class A");
        }
}
class B extends A
{
        void disp()
        {
                System.out.println("Method of class B");
        }
}
class MOTest
{
        public static void main(String args[])
        {
                B b=new B();
                b.disp();
        }
}
```

Output:

```
E:\ksr>javac MOTest.java
E:\ksr>java  MOTest
Method of class B
```

Using Abstract Classes

Sometimes it may be need by the super class to define the structure of the every method without implementing it. The subclass can fill or implement the method according to its requirements. This kind of situation can come into picture whenever the super class unable to implement the meaningful implementation of the method. For example, if we want to find the area of the Figure given, which can be Circle, Rectangle, and Triangle. The **class Figure** defines the method area(), when subclass implements its code, it implements its own version of the method. The Java's solution to this problem is **abstract method.**

To declare anabstract method, use this general form:

```
abstract type name(parameter-list);
```

As you can see, no method body is present.

To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keywordat the beginning of the class declaration. There can be no objects of an abstract class. That is,an abstract class cannot be directly instantiated with the **new** operator. Such objects wouldbe useless, because an abstract class is not fully defined.

Example Program:

FigureDemo.java

c). Write a java program for abstract class to find areas of different shapes (Exercise 5 (c))

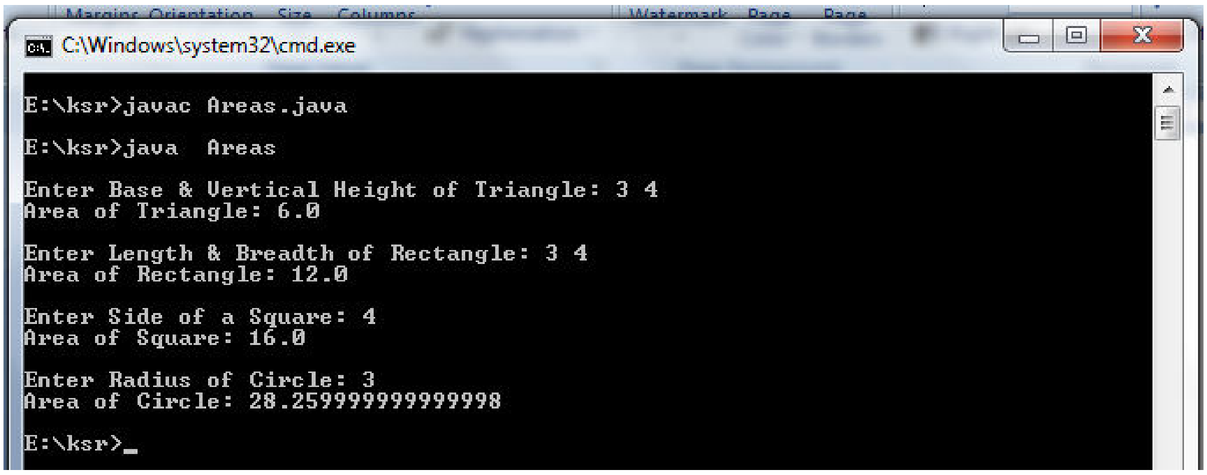
```
import java.util.Scanner;

abstract class CalcAreas
{
   abstract void findTriangle(double b, double h);
   abstract void findRectangle(double l, double b);
   abstract void findSquare(double s);
   abstract void findCircle(double r);
}

class FindArea extends CalcAreas
{
    void findTriangle(double b, double h)
   {
      double area = (b*h)/2;
      System.out.println("Area of Triangle: "+area);
   }
```

```
     void findRectangle(double l, double b)
    {
       double area = l*b;
       System.out.println("Area of Rectangle: "+area);
    }
     void findSquare(double s)
    {
       double area = s*s;
       System.out.println("Area of Square: "+area);
    }
     void findCircle(double r)
    {
       double area = 3.14*r*r;
       System.out.println("Area of Circle: "+area);
    }
}
class Areas
{
    public static void main(String args[])
    {
       double l, b, h, r, s;
       FindArea area = new FindArea();
       Scanner get = new Scanner(System.in);
       System.out.print("\nEnter Base & Vertical Height of Triangle: ");
       b = get.nextDouble(); h
       = get.nextDouble();
       area.findTriangle(b, h);
       System.out.print("\nEnter Length & Breadth of Rectangle: ");
       l = get.nextDouble(); b
       = get.nextDouble();
       area.findRectangle(l, b);
       System.out.print("\nEnter Side of a Square: ");
       s = get.nextDouble();
       area.findSquare(s);
       System.out.print("\nEnter Radius of Circle: ");
       r = get.nextDouble();
       area.findCircle(r);
    }
}
```

Output:


C:\Windows\system32\cmd.exe

```
E:\ksr>javac Areas.java

E:\ksr>java  Areas

Enter Base & Vertical Height of Triangle: 3 4
Area of Triangle: 6.0

Enter Length & Breadth of Rectangle: 3 4
Area of Rectangle: 12.0

Enter Side of a Square: 4
Area of Square: 16.0

Enter Radius of Circle: 3
Area of Circle: 28.259999999999998

E:\ksr>_
```

Dynamic Method Dispatch (Runtime Polymorphism) (Topic Beyond Syllabus)

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements **run-time polymorphism.**

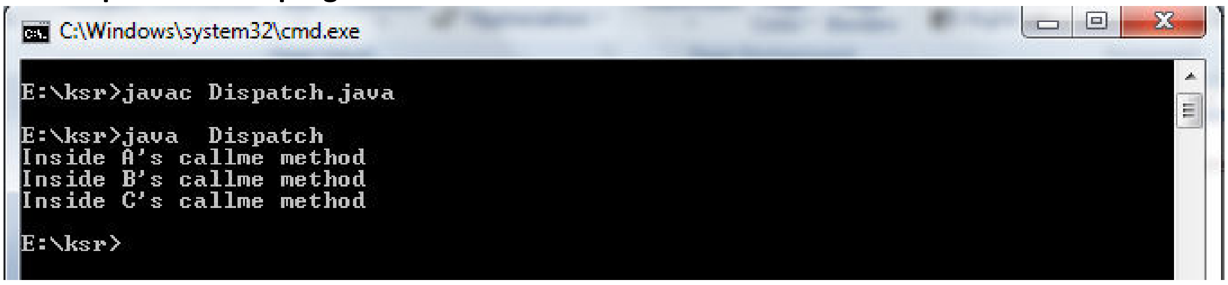
Let's begin by restating an important principle: a super class reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a super class reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
Here is an example that illustrates dynamic method dispatch:

Write a JAVA program that implements Runtime polymorphism class A (Exercise 8 (a))

```
{
        void callme()
        {
                System.out.println("Inside A's callme method");
        }
}
class B extends A
{
        // override callme()
        void callme()
        {
                System.out.println("Inside B's callme method");
        }
}
class C extends A
{
        // override callme()
        void callme()
        {
                System.out.println("Inside C's callme method");
        }
}
```

```
class Dispatch
{
	public static void main(String args[])
	{
	A a = new A(); // object of type A
	B b = new B(); // object of type B
	C c = new C(); // object of type C
	A r; // obtain a reference of type A r =
	a; // r refers to an A object r.callme();
	// calls A's version of callme r = b; // r
	refers to a B object
	r.callme(); // calls B's version of callme
	r = c; // r refers to a C object
	r.callme(); // calls C's version of callme
	}
}
```

The output from the program is shown here:



```
E:\ksr>javac Dispatch.java

E:\ksr>java  Dispatch
Inside A's callme method
Inside B's callme method
Inside C's callme method

E:\ksr>
```

Introduction to Interfaces

Java supports the concept of Inheritance, that is acquiring the properties from one class to other class. The class that acquires properties is called **"subclass"**, and the class from which it acquire is called "**super class**". Here, one class can acquire properties from other class using the following statement:

```
class A extends  B
{
---------
---------
}
```

But, Java does not allow to acquire properties from more than one class, which we call it as "**multiple inheritance**". We know that large number of real-life applications require the use of multiple inheritance. Java provides an alternative approach known as "***interface***" to support the concept of multiple inheritance.

Defining Interface

An interface is basically a kind of class. Like classes, interfaces contain the methods and variables but with major difference. The difference is that interface define only abstract methods and final fields. This means that interface do not specify any code to implement these methods and data fields contain only constants. Therefore, it is the responsibility of the class that implements an interface to define the code for these methods.

The syntax of defining an interface is very similar to that of class. The general form of an interface will be as follows:

```
        interface  Intyerface_name
{
        //variables inside interface are by default final, publicand static
        type id=value;
        //by default abstract methods
        return_type method_name(paprameter_list);
        return_type method_name(paprameter_list);
        return_type method_name(paprameter_list);

}
```

// Here, ***interface*** is the keyword and the Calculator is name for interface. The variables are declared as follows:

Note: 1) ***variables inside interface are by default final, public and static***
2) by default methods are public and abstract
***These methods must be implemented* any class that want to acquire the properties.**

Here is an example of an interface definition that contain two variable and one method

```
interface Calculator
{
        //variables inside interface are by default final, public and static
        double PI=3.14;
        //by default abstract methods
        int add(int a,int b);
        int sub(int a,int b);
        int mul(int a,int b);
        int div(int a,int b);
        double area(int r);
}
```

Implementing the interface

Interfaces are used as "**superclasses**" whose properties are inherited by the classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

```
        class Class_Name      implements     Interface_Name
        {
                //Body of the class
        }
```

Here, Class_Name class, implements the interface "Interface_Name". A more general form of implementation may look like this:

```
class    Class_Name    extends   Superclass_Name   implements interface1,
interface2,interface3..
{
//        body of the class
}
```

This shows that a class can extend another class while implementing interfaces. When a class implements more than one interface they are separated by a comma.

Example program using interface

InterfaceTest.Java

```
interface Calculator
{
        //variables inside interface are by default final, publi and static
        double PI=3.14;
        //by default abstract methods
        int add(int a,int b);
        int sub(int a,int b);
        int mul(int a,int b);
        int div(int a,int b);
        double area(int r);
}
class NormCal implements Calculator
{
        public int add(int x,int y)
        {
        return(x+y);
        }
        public int sub(int x,int y)
        {
        return(x-y);
        }
        public int mul(int x,int y)
        {
        return(x*y);
        }
        public int div(int x,int y)
        {
        return(x/y);
        }
        public double area(int r)
        {
                return(PI*r*r);
        }
        public static void main(String args[])
        {
        NormCal nc=new NormCal();
```

```
		System.out.println("The sum is:"+nc.add(2,3));
		System.out.println("The sum is:"+nc.sub(2,3));
		System.out.println("The sum is:"+nc.mul(2,3));
		System.out.println("The sum is:"+nc.div(4,2));
		System.out.println("area of Circle is:"+nc.area(5));
		}
}
```

OutPut:

```
E:\ksr>javac NormCal.java
E:\ksr>java  NormCal
The sum is:5
The sum is:-1
The sum is:6
The sum is:2
area of Circle is:78.5
```

Extending the Interfaces

Like classes interfaces also can be extended. That is, an interface can be sub interfaced from other interface. The new sub interface will inherit all the members from the super interface in the manner similar to the subclass. This is achieved using the keyword extends as shown here:

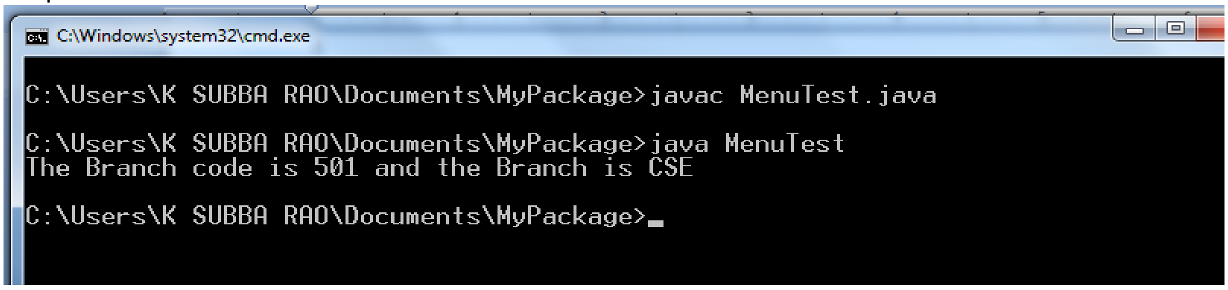
```
		interface	Interface_Name1		extendsInterface_Name2
		{
			//Body of the Interface_name1
		}
```

For example, **MenuTest.java**

```
interface Const
{
		static final int code=501;
		static final String branch="CSE";
}
interface Item extends Const
{
		void display();
}
class Menu implements Item
{
		public void display()
		{
			System.out.println("The Branch code is "+code+" and the Branch is "+branch);
		}
}
class MenuTest{
		public static void main(String args[])
		{
```

```
                Menu m=new Menu(); // this contains the interfaces Item and Const
                m.display();
        }
}
```

Output:



```
C:\Users\K SUBBA RAO\Documents\MyPackage>javac MenuTest.java

C:\Users\K SUBBA RAO\Documents\MyPackage>java MenuTest
The Branch code is 501 and the Branch is CSE

C:\Users\K SUBBA RAO\Documents\MyPackage>_
```

Interfaces Vs Abstract classes

Sl	Interface	Abstract
1	Multiple Inheritance possible	Multiple Inheritance not possible
2	***implements*** keyword is used	***extends*** keyword is used
3	By default all the methods are public, and abstract. No need to tag as public and abstract	Methods have to be tagged as public and abstract.
4	All methods of interface need to be overridden	Only abstract methods need to be overridden
5	All variable declared in interface are By default public, final and static	Variable if required, need to be declared in interface as public, final and static
6	Methods cannot be static	Non-abstract methods can be static

Packages

Introduction to Packages

One of the main features of the Object Oriented Programming language is the ability to ***reuse the code*** that is already created. One way for achieving this is by extending the classes and implementing the interfaces. Java provides a mechanism to partition the classes into smaller ***chunks***. This mechanism is the ***Package***. The ***Package is container of classes***. The class is the container of the data and code. The package also addresses the problem of name ***space collision*** that occurs when we use same name for multiple classes. Java provides convenient way to keep the same name for classes as long as their subdirectories are different.

Benefits of packages:

1. The classes in the packages can be easily reused.
2. In the packages, classes are unique compared with other classes in the other packages.
3. Packages provide a way to hide classes thus prevent classes from accessing by other programs.
4. Packages also provide way to separate "design" from "coding".

Categories of Packages

The Java Packages are categorized into two categories (i) Java API Packages (ii) User Defined Packages.

1. **Java API Packages** –Java API provides large number of classes grouped into different packages according to the functionality. Most of the time we use the package available with Java API.

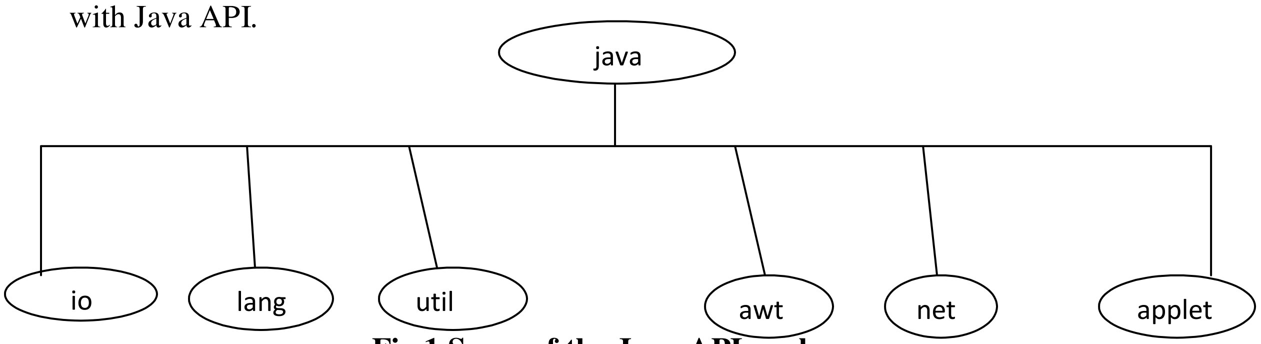


Fig 1 Some of the Java API packages

Table 1 Java System Packages and their Classes

Sl No	Package Name	Contents
1	java.lang	Contains language support classes. The java compiler automatically uses this package. This includes the classes such as primitive data types, String, StringBuffer, StringBuilde etc;
2	java.util	Contains the language utility classes such asa Vectors, Hash Table , Date, StringTokenizer etc;
3	java.io	Contains the classes that support input and output classes.
4	java.awt	Contains the classes for implementing the graphical user interfaces
5	Java.net	Contains the classes for networking
6	Java.applet	Contains the classes for creating and implementing the applets.

Java.lang Package

The java.lang package includes the following classes.

Boolean	Enum	Package	StackTraceElement	ThreadGroup
Byte	Float	Process	StrictMath	ThreadLocal
Character	Integer	ProcessBuilder	String	Throwable
Class	Long	Runtime	StringBuffer	Void
ClassLoader	Math	RuntimePermission	StringBuilder	
Compiler	Number	SecurityManager	System	
Double	Object	Short	Thread	

Using the System Packages

Packages are organized in hierarchical structure, that means a package can contain another package, which in turn contains several classes for performing different tasks. There are two ways to access the classes of the packages.

i. **fully qualified class name**- this is done specifying package name containing the class and appending the class to it with dot (.) operator.
Example: java.util.StringTokenizer(); Here, "java.util" is the package and "StringTokenizer()" is the class in that package. This is used when we want to refer to only one class in a package.

ii. **import** statement –this is used when we want to use a class or many classes in many places in our program. **Example: (1) import java.util.*;**
(2) import java.util.StringTokenizer;

<u>**Naming Conventions**</u>

Packages can be named using the Java Standard naming rules. Packages begin with "**lower case**" letters. It is easy for the user to distinguish it from the class names. All the class Name by convention begin with "**upper case**" letters. Example:

double d= java.lang.Math.sqrt(3);

Here, "java.lang" is the package, and "Math" is the class name, and "sqrt()" is the method name.

2. <u>User Define Packages</u>

To create a package is quite easy: simply include a **package** command in the first line of the source file. A class specified in that file belongs to that package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put in the default package, which has no name.

The general form of the package statement will be as followed: **package pkg;**
Here, "pkg" is the package name. Example: **package MyPackage;**

Java Uses file system directories to store packages. For example, the " **.class"** files for any classes you declare to be part of the "MyPackage" must be store in the "MyPackage" directory. The directory name "MyPackage" is different from "mypackage".
Notes: 1. The case for the package name is significant.
2. The directory name must match with package name.

We can create hierarchy of packages. To do so, simply separate package name from the above one with it by use of the dot (.) operator. The general form of multileveled package statement is shown here: **package pkg1.pkg2.pkg3;**
Example: package iicse.asection.java

<u>Finding the Packages and CLASSPATH</u>

Packages are mirrored by the directories. This raises an import question: how does Java-Run time system look for the packages that you create? The answer has three parts: (1) By default, Java-Run time system uses the current working directory as its starting point. Thus your package is in a subdirectory of your directory, it will be found. (2) You can specify a directory path by setting the CLASSPATH environment variable. (3) You can use - **classpath** option with **java** and **iavac** to specify the path for the classes.

A Short Example Package: **Calc.java**

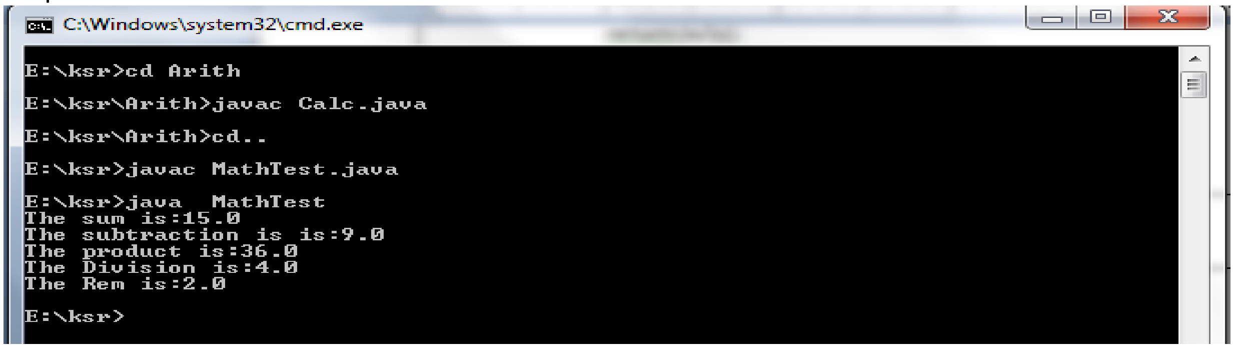
Write a JAVA program that import and use the defined your package in the previous Problem (Exercise – 12 (c))

```
package Arith;   //package name
public class Calc
{
        public float add(float x,float y)
        {
                return (x+y);
        }
        public float sub(float x,float y)
        {
                return (x-y);
        }
        public float mul(float x,float y)
        {
                return (x*y);
        }
        public   float div(float x,float y)
        {
                return (x/y);
        }
        public   float rem(float x,float y)
        {
                return (x%y);
        }
}
```

Using the package in another class called "MathTest.java"

```
import Arith.*;   //accessing the package
class MathTest
{
        public static void main(String args[])
        {
                Calc c=new Calc();
                System.out.println("The sum is:"+c.add(12,3));
                System.out.println("The subtraction is is:"+c.sub(12,3));
                System.out.println("The product is:"+c.mul(12,3));
                System.out.println("The Division is:"+c.div(12,3));
                System.out.println("The Rem is:"+c.rem(12,5));
        }
}
```

Output:



```
E:\ksr>cd Arith

E:\ksr\Arith>javac Calc.java

E:\ksr\Arith>cd..

E:\ksr>javac MathTest.java

E:\ksr>java  MathTest
The sum is:15.0
The subtraction is is:9.0
The product is:36.0
The Division is:4.0
The Rem is:2.0

E:\ksr>
```

Procedure for Creating and using the package:

- Open a text editor or notepad and type the above code of “Calc.java” and writing the “package Arith“ as the first statement in it as shown in the program.
- Create a Folder with name “Arith” same as package name. Save the “Calc.java” in this file and compile it with help of command prompt with directory path “ E:/ksr/Arith> javac Calc.java
- Open another text editor and type the code “MathTest.java” in it and save it in the directory E:/ksr.
- Compile this file and run it, then we see the output.

Package and Member Accessing

The visibility of an element is specified by the access specifiers: public, private, and protected and also the package in which it resides. The visibility of an element is determined by its visibility within class, and visibility within the package.

- If any members explicitly declared as "**public**", they are visible everywhere, including in different classes and packages.
- "**private**" members are accessed by only other members of its class. A private member is unaffected by its membership in a package.
- A member specified as "**protected**" is accessed within its package and to all subclasses.

Class Member Access:

Sl No	**Class member**	**Private Member**	**Default Member**	**Protected Member**	**Public Member**
1	Visible within same class	YES	YES	YES	Yes
2	Visible within the same package by subclasses	No	YES	YES	YES
3	Visible within same package by non-subclass	No	YES	YES	YES
4	Visible within different packages by subclasses	NO	NO	YES	YES
5	Visible within different packages by Non-subclasses	NO	NO	NO	YES

Adding a class to a Package:

It is simple to add a class to already existing package. Consider the following Package:

A.java

```
package p1;
public class A
{
                // body of the class
}
```

Here, the package p1 contains one public class by the name A. Suppose if we want to add another class B to this package. This can be done as follows:

1. Define the class and make it public
2. Place the package statement in the begin of class definition
   ```
   package p1;
   public class B
   {
   //body of the class B
   }
   ```
3. store this as "**B.java**" file under the directory p1.
4. Compile "**B.java**" file by switching to the subdirectory. This will create "**B.class**" file and place it in the directory **p1**.
   Now the package p1 will contain both A.class and B.class files

Exceptions and Assertions:

Introduction to Exception

An Exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is run-time error. A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.

Importance of try, catch, throw, throws and finally block

Exception handling is managed by Java by **via five keywords:**

- **try**
- **catch**
- **throw**
- **throws**
- **finally**.

Working Exception handling Techniques:

1. Briefly, here is how they work. Program statements that you want to monitor for exceptions are placed within a **try** block.

2. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner.
3. System-generated exceptions are **automatically** thrown by the Java run-time system. To **manually** throw an exception, use the keyword **throw**.
4. Any exception that is thrown out of a method must be specified as such by a **throws** clause.
5. Any code that absolutely must beexecuted after a **try** block completes is put in a **finally** block.

Syntax:

```
try
{
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
        // exception handler for ExceptionType2
}
// ...
finally
{
        // block of code to be executed after try block ends
}
```

Exception Types

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below (Fig 2) **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **Runtime Exception**. Exceptions of this type are **automatically** defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. **Stack overflow is an example of such an error**. This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

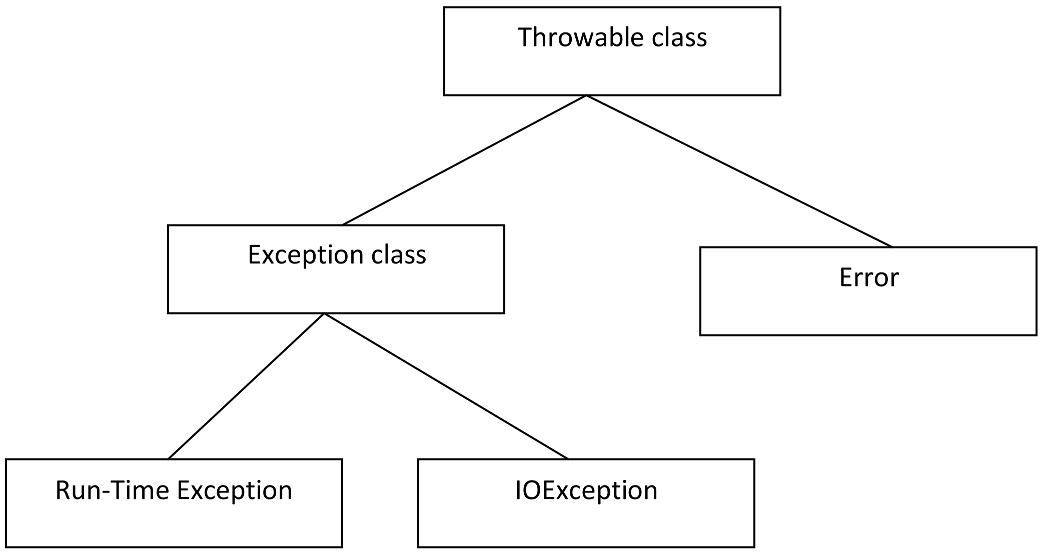


Fig 2: Exception Types

The Java Built-in Exceptions are broadly classified into two categories: Checked and Unchecked exceptions. The Checked Exceptions are those for which the compiler checks to see whether they have been handled in your program or not. Unchecked or Run-Time Exceptions are not checked by the compiler. These Unchecked Exceptions are handled by the Java Run-Time System automatically.

Checked Exceptions	**Unchecked Exceptions**
ClassNotFoundException	Arithmetic Exception
NoSuchFieldException	ArrayIndexOutOfBoundsException
NoSuchMethodException	NullPointerException
InerruptedException	ClassCastException
IOException	BufferOverFlowException
IllegalAccessException	BufferUnderFlowException

Uncaught Exceptions:

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0
{
public static void main(String args[])
        {
                int d = 0;
                int a = 42 / d;
        }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs anew exception object and then ***throws*** this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.

In the above program we have not provided any exception handler, in this context the exception is caught by the **default handler**. The **default handler** displays string describing the exception.

Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**;and the line number, **4**, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened.

Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides **two benefits**.

- First, it allows you to fix the error.
- Second, it prevents the program from automatically terminating.

Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause that processes the **ArithmeticException** generated by the division-by-zero error:

Exc2.java

```
class Exc2
{
public static void main(String args[])
{
        int d, a;
        try {
                // monitor a block of code.
                d = 0;
                a = 42 / d;
                System.out.println("This will not be printed.");
           }
        catch (ArithmeticException e)
           { // catch divide-by-zero error
                System.out.println("Division by zero.");
        }
System.out.println("After catch statement.");
}
}
```

Note: println() statement inside the "**try**" will never execute, because once the exception is raised the control is transferred to the "**catch**" block. Here is catch is not called, hence it will not return the control back to the "try" block. The "try" and "catch" will form like a unit. A catch statement cannot handle the exception thrown by another "try" block.

Displaying a Description of an Exception

Throwable overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println( )**statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e)
        {
                System.out.println("Exception: " + e);
                a = 0; // set a to zero and continue
        }
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

Exception: java.lang.ArithmeticException: / by zero

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block. The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultiCatch
{
public static void main(String args[])
{
try
{
        int a = args.length;
        System.out.println("a = " + a);
        int b = 42 / a;
        int c[] = { 1 };
        c[42] = 99;
}
catch(ArithmeticException e)
{
        System.out.println("Divide by 0: " + e);
}
 catch(ArrayIndexOutOfBoundsException e)
{
        System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

Nested try Statements

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception.

throw:

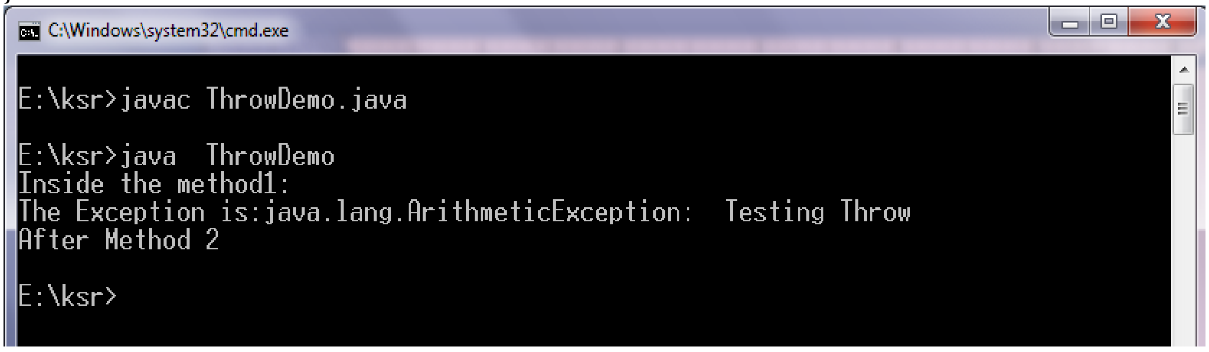
So far, you have only been catching exceptions that are thrown by the Java run-time system **implicitly**. However, it is possible for your program to throw an exception **explicitly**, using the **throw** statement. The general form of **throw** is shown here:

throw*ThrowableInstance***;**

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.
Example Program:

ThrowDemo.java

```
class ThrowDemo
{
        public static void main(String args[])
        {
         method1();
        }
        static void method1()
        {
                System.out.println("Inside the method1:");
                try
                {
                        method2();
                }
                catch(Exception e)
                {
                        System.out.println("The Exception is:"+e);
                }
                System.out.println("After Method 2");
        }
        static void method2()
        {
                throw new ArithmeticException(" Testing Throw");
        }
}
```



```
E:\ksr>javac ThrowDemo.java

E:\ksr>java  ThrowDemo
Inside the method1:
The Exception is:java.lang.ArithmeticException:  Testing Throw
After Method 2

E:\ksr>
```

throws:

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
        // body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

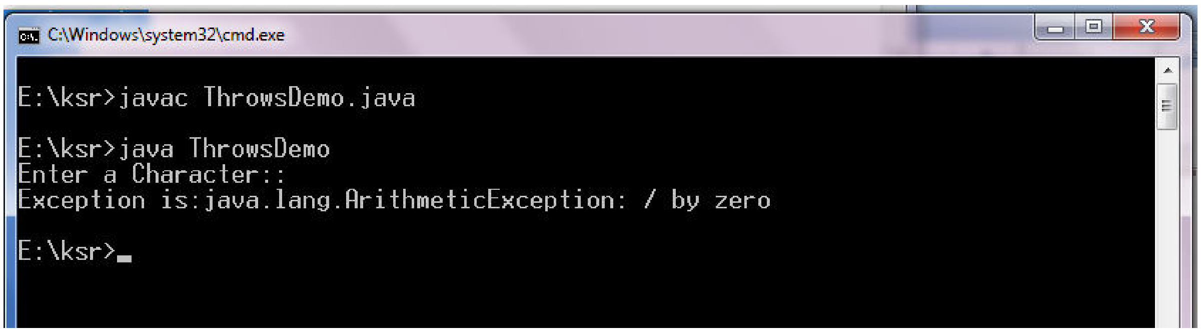
Example Program:

ThrowsDemo.java

```
import java.io.*;
class ThrowsDemo
{
public static char prompt(String str) throws IOException,ArithmeticException
{
//called method throws two exceptions
System.out.println(str+":");
int x=20,y=0;
int z=x/y;
return (char) System.in.read();
}
public static void main(String args[])
{
char ch;
try
{
```

```
                    //calling a method- here main is caller
                    ch=prompt("Enter a Character:");
                    //it is the responsibility of the caller to handle it
}
catch(IOExceptione)
{
                        System.out.println("Exception is:"+e);
}
catch(ArithmeticExceptionae)
{
                     System.out.println("Exception is:"+ae);
}
}
}
```

Output:



```
E:\ksr>javac ThrowsDemo.java

E:\ksr>java ThrowsDemo
Enter a Character::
Exception is:java.lang.ArithmeticException: / by zero

E:\ksr>_
```

finally:

The finally block is always is executed. It is always used to perform house keeping operations such as releasing the resources, closing the files that already opened etc,. That means the statement that put inside the finally block are executed compulsorily.It is always followed by the try-catch statements.

Syntax:

```
          try
          {
                    // statements
                    }
          catch(Exception e)
          {
                    //Handlers
          }
          finally
          {
                    //statements
          }
```

Exception encapsulation and enrichment

The process of wrapping the caught exception in a different exception is called "Exception Encapsulation". The Throwable super class has added one parameter in its constructor for the wrapped exception and a "getCause()" method to return the wrapped exception. Wrapping is also used to hide the details of implementation.

Syntax:

```
         try
        {
                throw new ArithmeticException();
        }
        catch(AritmeticExceptions ae)
        {
        //wrapping exception
                throw new ExcepDemo("Testing User Exception",ae);
        }
```

Disadvantages of wrapping:

- It leads to the long Stack traces.
- It is a difficult task to figure out where the exception is

Solution:

The possible solution is Exception enrichment. Here we don't wrap exception but we add some information to the already thrown exception and rethrow it.

Example program:

```
class ExcepDempo extends Exception
{
        String message;
        ExcepDemo(String msg)
        {
                message=msg;
        }
        public void addInformation(String msg)
        {
                message=message+msg;
        }
}
class ExcepEnrich
{
        static void testException()
        {
                try{
                        throw new ExcepDemo("Testing user Exceptio:");
                }
                catch(ExcepDemo e)
                {
                        e.addInformation("Example Exception");
                }
        }
        public static void main(String args[])
        {
                try
        {
                testException();
        }
        catch(Exception e)
        {
```

```
                        System.out.println(e);
        }
        }
}
```

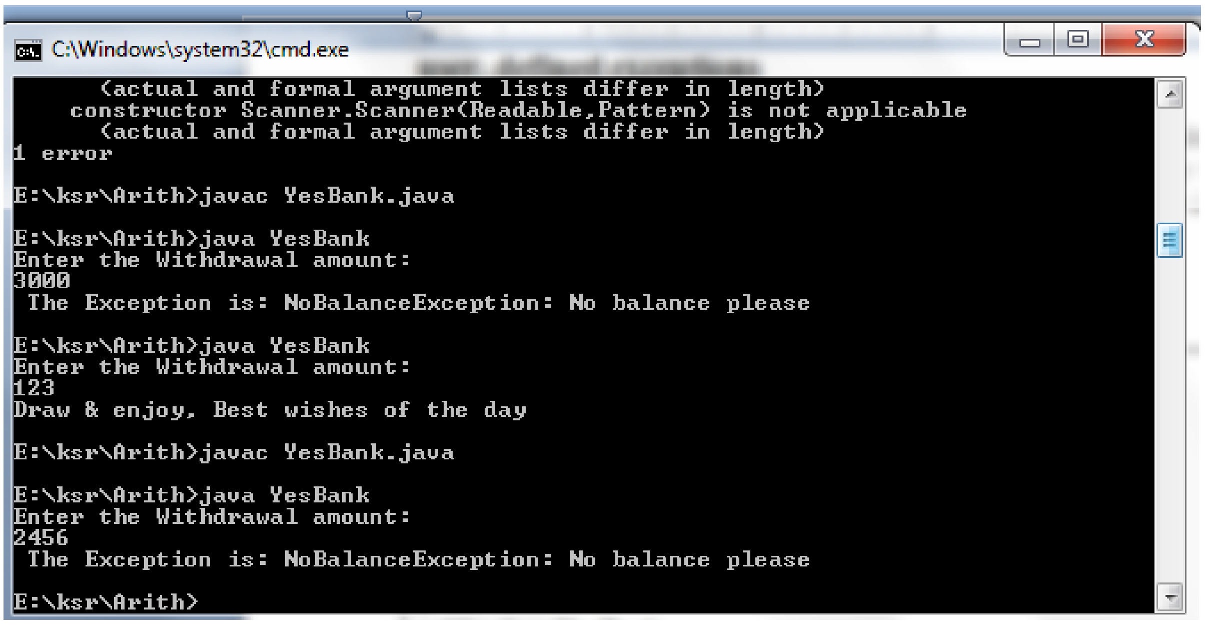
user- defined exceptions

Java has rich set of Built-in Exceptions which can handle almost all the exceptions. If we want to create our own exceptions, java provides a simple way to do so. This is possible by defining a sub class of **Exception** super class. The Exception is the sub class of Throwable class from which it acquires the properties.

Example program on User defined Exception

```
import java.util.*;
class NoBalanceException extends Exception
{
        public NoBalanceException(String problem)
        {
           super(problem);
        }
}
public class YesBank
{
         public static void main( String args[ ] )
        {
                 int balance=2000, withdraw ;
                 Scanner in=new Scanner(System.in);
                System.out.println("Enter the Withdrawal amount:");
                withdraw=in.nextInt();
                try
                {
                if (balance < withdraw)
                   {
                    NoBalanceException e = new NoBalanceException("No balance please");
                    throw e;
                   }
                else
                   {
                    System.out.println("Draw & enjoy, Best wishes of the day");
                   }
                }
            catch(Exception exp)
        {
                System.out.println(" The Exception is: "+exp);
        }
   }
}
```

Output:



```
        (actual and formal argument lists differ in length)
    constructor Scanner.Scanner(Readable,Pattern) is not applicable
        (actual and formal argument lists differ in length)
1 error

E:\ksr\Arith>javac YesBank.java

E:\ksr\Arith>java YesBank
Enter the Withdrawal amount:
3000
 The Exception is: NoBalanceException: No balance please

E:\ksr\Arith>java YesBank
Enter the Withdrawal amount:
123
Draw & enjoy, Best wishes of the day

E:\ksr\Arith>javac YesBank.java

E:\ksr\Arith>java YesBank
Enter the Withdrawal amount:
2456
 The Exception is: NoBalanceException: No balance please

E:\ksr\Arith>
```

Assertions

Assertions are added after java 1.4 to always create reliable programs that are Correct and robust programs. The assertions are Boolean expressions. Conditions such as positive number or negative number are examples.

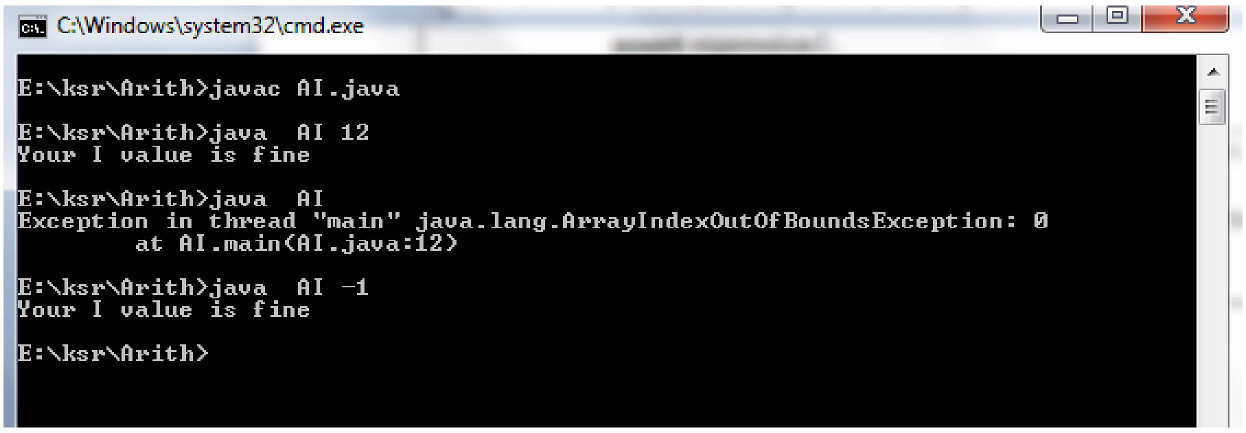
Syntax:

```
assert expression1;
        or
assert expression1:expression2;
```

Where **assert** is the keyword. Expression 1 is the Boolean expression, expression2 is the string that describes the Exceptions.
Note: assertions must be explicitly enabled. The –ea option is used to enable the exception and –da is used to disable the exception.

AI.java

```
import java.io.*;
class AI
{
 void check(int i)
 {
        assert i>0:" I must be positive:";
        System.out.println("Your I value is fine");
 }
 public static void main(String args[]) throws IOException
 {
        AI a=new AI();
        a.check(Integer.parseInt(args[0]));
 }
}}
```



```
E:\ksr\Arith>javac AI.java

E:\ksr\Arith>java  AI 12
Your I value is fine

E:\ksr\Arith>java  AI
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at AI.main(AI.java:12)

E:\ksr\Arith>java  AI -1
Your I value is fine

E:\ksr\Arith>
```