Java Programming

Unit – 3

Inheritance, Polymorphism Interfaces, Exceptions

Dr. Y. J. Nagendra Kumar
Professor of IT
Dean - Technology and Innovation Cell - GRIET

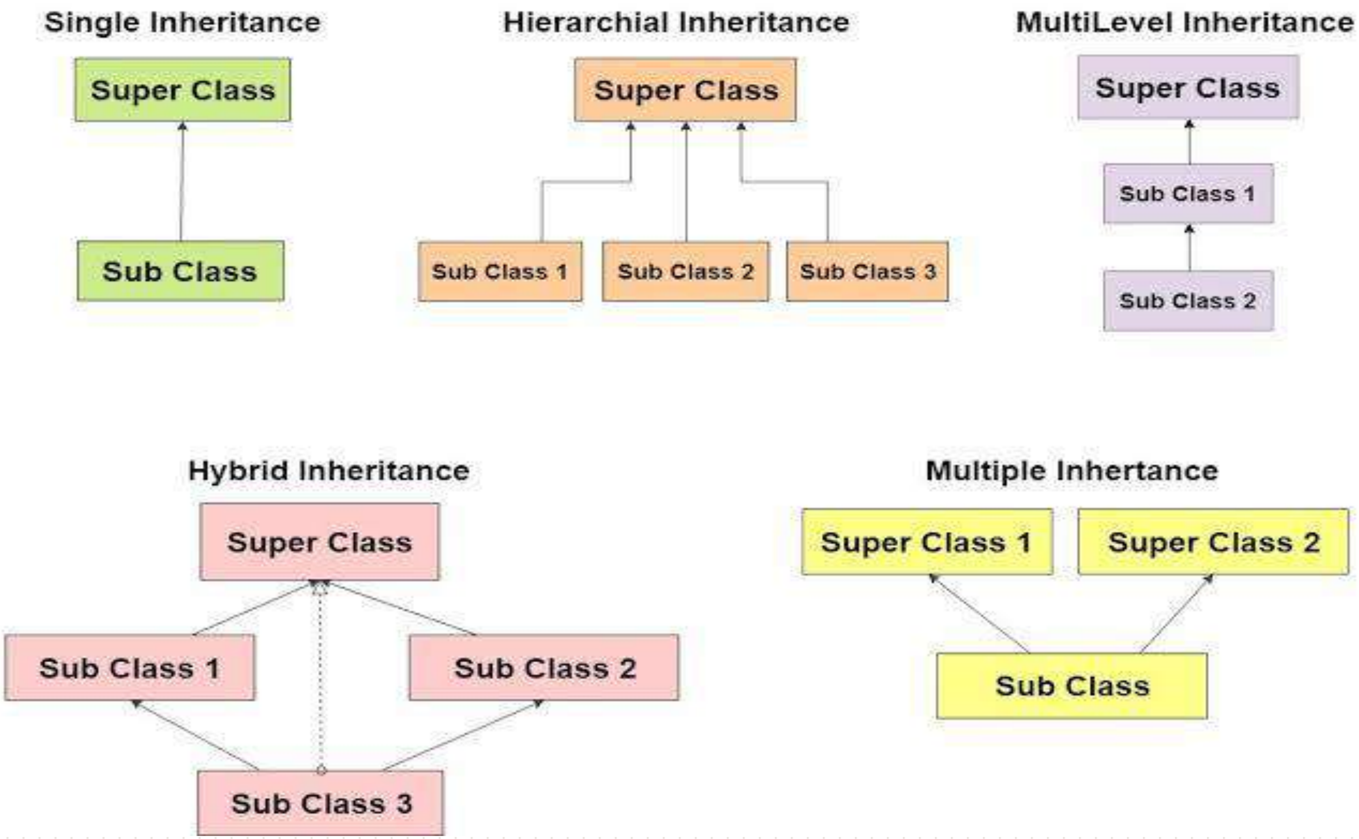
TABLE OF CONTENTS

Inheritance

Inheritance

- The mechanism of deriving a new class from an old one is called Inheritance.
- The old class is known as “base” or “super” or “parent” class.
- The new one is called the “derived” or “sub” or “child” class.
- The forms of Inheritance
  - Single
  - Multiple
  - Multi level
  - Hierarchical
  - Hybrid

Types of Inheritance



Single Inheritance

- Subclass inherits all of the instance variables and methods defined by the super class and adds its own unique elements.
- To inherit a class we simply incorporate the definition of one class into another by using the “extends” keyword.

- Syn:

```
class subclassname extends superclassname
{
      //body
}
```

- Subclass cannot access those members of super class that have been declared as “private”

Single Inheritance Example

```
class rect
{       double length;
        double breadth;
        rect()
        {  length=-1; breadth=-1;           }
        rect(double l,double b)
        {
                length=l; breadth=b;
        }
        double area()
        {
                return length*breadth;
        }
}
```

```
class box extends rect
{
      double height;
        box(double l,double b,double h)
        {           length=l;
                    breadth=b;
                    height=h;
        }
        void volume()
        {
                    System.out.println("Volume :
        "+length*breadth*height);
        }
}
```

Single Inheritance Example Contd.,

```
class inhert1
{
    public static void main(String ar[])
    {
        rect r=new rect(20,10);
        System.out.println("Area : "+r.area());
        box b=new box(20,10,5);
        b.volume();
    }
}
```

- In the previous example, the constructor for **box** explicitly initializes the length and breadth fields of rectangle. This duplicate code makes the program inefficient.

- Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword “**super**”.

‘super’ forms

- super has two general forms.
  - The first calls the superclass’ constructor.
  - The second is used to access a member of the superclass

Using super to Call Superclass Constructors

- A subclass can call a constructor method defined by its superclass by using the following form of super:

super(*parameter-list*);

- Here, *parameter-list* specifies any parameters needed by the constructor in the superclass.
- super( ) must always be the first statement executed inside a subclass’ constructor.

Super First form Example

```
class rect
{       double length;
        double breadth;
        rect()
        {  length=-1; breadth=-1;           }
        rect(double l,double b)
        {
                length=l; breadth=b;
        }
        double area()
        {
                return length*breadth;
        }
}
```

```
class box extends rect
{
      double height;
         box()
        {           super();
                    height=-1;      }
        box(double l,double b,double h)
        {           super(l,b);
                    height=h;       }
        void volume()
        {
System.out.println("Volume : “+length*breadth*height);
        }
}
```

Super First form Example Contd.,

```
class inhert2
{
        public static void main(String ar[])
        {
                rect r=new rect(20,10);
                System.out.println("Area : "+r.area());

                box b=new box(30,40,5);
                b.volume();
        }
}
```

A Second Use for super

- The second form of super acts like this, except that it always refers to the superclass of the subclass in which it is used.

  Syn: 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 superclass.

Super Second form Example

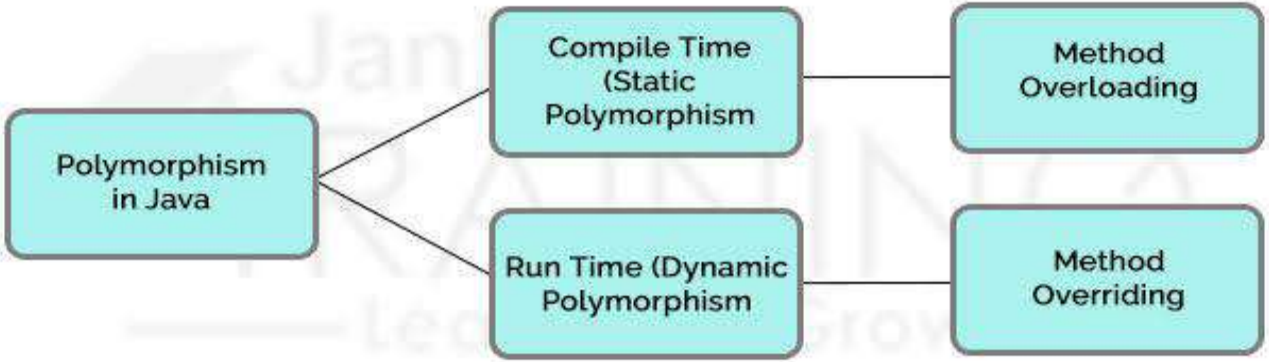
```
class A
{
        int i;
}

class B extends A
{
        int i;

        B(int a,int b)
        {
                super.i=a;
                i=b;
        }
```

```
void display()
        {
                System.out.println(" Super i : "+super.i);
                System.out.println(" Sub i : "+i);
        }
}
class inhert3
{
        public static void main(String ar[])
        {
                B sub=new B(20,30);
                sub.display();
        }
}
```

Polymorphism

Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- The word "poly" means many and "morphs" means forms. So polymorphism means many forms.



Polymorphism in Java

Compile Time (Static Polymorphism

Run Time (Dynamic Polymorphism

Method Overloading

Method Overriding

Method Overloading

- In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called ***“Method Overloading”.***

- Method Overloading is used when objects are required to perform similar tasks but using different input parameters. This process is known as ***Polymorphism.***

Method Overloading Example

```
class A
{       int i,j;
        A(int a, int b)
        {           i=a;j=b;        }
        void display()
        {
        System.out.println(" i and j "+i+" "+j);
        }
}
class B extends A
{       int k;
        B(int a,int b,int c)
        {           super(a,b);
                    k=c;            }
        void display(String s)
        {           System.out.println(s+k);
        }
}
class overload
{
        public static void main(String ar[])
        {
                    B sub=new B(10,20,30);

                    sub.display(" this is :");
                    sub.display();
        }
}
```

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to ***override*** the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Method Overriding Example

```
class A
{       int i,j;
        A(int a, int b)
        {           i=a;j=b;        }
        void display()
        {
        System.out.println(" i and j "+i+" "+j);
        }
}
class B extends A
{       int k;
        B(int a,int b,int c)
        {           super(a,b);
                    k=c;            }
        void display()
        {
                    System.out.println("k: " + k);
        }
}
class Override
{
        public static void main(String args[])
        {
                    B subOb = new B(1, 2, 3);
        subOb.display();   // this calls show() in B
        }
}
```

Dynamic Method Dispatch

- 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 also known as **run-time polymorphism.**

Dynamic Method Dispatch

- When an overridden method is called through a superclass 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. When different types of objects are referred to, different versions of an overridden method will be called.

Dynamic Method Dispatch Example

```
class draw
{
        double d1,d2;

        draw(double a, double b)
        {
                d1=a;d2=b;
        }
        double area()
        {
System.out.println(" Area is undefined");
                return 0;
        }
}
```

```
class triangle extends draw
{
        triangle(double a,double b)
        {
                super(a,b);
        }

        double area()
        {
System.out.println(" Inside Area for Triangle : ");
        return d1*d2/2;  // base * height / 2
        }
}
```

Dynamic Method Dispatch Example contd.,

```
class rectangle extends draw
{
        rectangle(double a,double b)
        {
                super(a,b);
        }

        double area()
        {
        System.out.println(" Inside Area for Rectangle : ");
        return d1*d2;  // length * breadth
        }
}
```

```
class dynamic
{       public static void main(String ar[])
        {               draw d=new draw(10,20);
                        triangle t=new triangle(12,8);
                        rectangle r=new rectangle(6,4);
                        draw ref;
                        ref=d;
        System.out.println("Area is :"+ref.area());
                        ref=t;
        System.out.println("Area is :"+ref.area());
                        ref=r;
        System.out.println("Area is :"+ref.area());
        }
}
```

Using “final” to Prevent Overriding

- To disallow a method from being overridden, specify final as a modifier at the start of its declaration.
- Methods declared as final **cannot be overridden.**

Final method Example

```
class A
{
        final void display()
        {             System.out.println(" Inside A  ");
        }
}
class B extends A
{
      // Error cannot override becoz of final
        void display()
        {
                      System.out.println(" Inside B  ");
        }
}
```

```
class finalkeyword
{       public static void main(String ar[])
        {             A a=new A();
                      a.display();
                      B b=new B();
                      b.display();    }
}
```

Output:

```
D:\>javac finalkeyword.java
finalkeyword.java:12: display() in B cannot override
        display() in A; overridden method is final
      void display() // Error cannot override becoz of
        final
            ^  1 error
```

Using final to Prevent Inheritance

- Sometimes we 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.
- 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.

Final Class Example

```
final class A
{
        void display()
        {   System.out.println(" Inside A  ");
        }
}
class B extends A
// Error cannot Inherit becoz of final
{
        void display()
        {
        System.out.println(" Inside B  ");
        }
}
```

```
class finalclass
{       public static void main(String ar[])
        {           A a=new A();
                    a.display();
                    B b=new B();
                    b.display();
        }
}
```

Output:

```
D:\>javac finalclass.java
finalclass.java:10: cannot inherit from final A
class B extends A // Error cannot Inherit becoz of final
              ^   1 error
```

Abstract Classes

Abstract Classes

- A superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- A superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- This situation can occur is when a superclass is unable to create a meaningful implementation for a method.

Abstract Methods

- Java’s solution to this problem is the *abstract method.*
- To declare an abstract method, use this general form:

abstract *type methodname(parameter-list)*;

*** no method body is present.

Abstract Classes

- Any class that contains one or more abstract methods must be declared abstract.
- To declare a class abstract, simply use the **abstract** keyword in front of the **class** keyword at 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 would be useless, because an abstract class is not fully defined.

Abstract Classes

```
abstract class draw
{
    double d1,d2;
    draw(double a, double b)
    {
        d1=a;d2=b;
    }
    abstract double area();
}
```

```
class triangle extends draw
{
    triangle(double a,double b)
    {
        super(a,b);
    }
    double area()
    {
    System.out.println(" Inside Area for Triangle : ");
    return d1*d2/2;  // base * height / 2
    }
}
```

Abstract Classes

```
class rectangle extends draw
{
        rectangle(double a,double b)
        {
                super(a,b);
        }

        double area()
        {
        System.out.println(" Inside Area for
        Rectangle : ");
        return d1*d2;  // length * breadth
        }
}
```

```
class abstractclasses
{       public static void main(String ar[])
        {
//      draw d=new draw(10,20);     is illegal
                triangle t=new triangle(12,8);
                rectangle r=new rectangle(6,4);
                draw ref;

                ref=t;
        System.out.println("Area is :"+ref.area());
                ref=r;
        System.out.println("Area is :"+ref.area());
        }
}
```

Object Class

- There is one special class, Object, defined by Java.
- All other classes are subclasses of Object. That is, Object is a **superclass of all other classes.**

Interfaces

Interfaces

- Java does not support multiple inheritance. That is, classes in java cannot have more than one superclass.
- Java provides an alternate approach known as interfaces to support the concept of multiple inheritance.
- Interfaces are syntactically similar to classes, but they define only abstract methods and final fields.
- This means that interfaces do not specify any code to implement these methods and data fields contain only constants.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

Defining an Interface

- An interface is defined much like a class. This is the general form of an interface:

```
access interface name
{
type final-varname1 = value;
type final-varname2 = value;
// ...
type final-varnameN = value;
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
// ...
return-type method-nameN(parameter-list);
}
```

Here, *access* is either **public** or not used

Interfaces

- Methods are, essentially, abstract methods
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class.
- All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

Ex:

```
interface shape
{
void area(int param);
}
```

Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.

Syntax:

```
access class classname [extends superclass]
[implements interface [,interface...]]
{
                    // class-body
}
```

Interfaces contd.,

- Here, *access* is either **public** or not used. If a class implements more than one interface, the interfaces are separated with a comma.
- The methods that implement an interface must be declared **public**.
- Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Interfaces

```
 class circle implements shape
{
                    // Implement shape's interface
                    public void area(int p)
                    {
                    System.out.println(“Area of Circle “+3.14*p*p);
                    }
}
```

Interfaces First Example

```
interface draw
{               final static double PI=3.14;
                double area(double d1,double d2);
}
class triangle implements draw
{
                public double area(double d1,double d2)
                {               return d1*d2/2;          }
}
class circle implements draw
{
                public double area(double d1,double d2)
                {               return PI*d1*d1;         }
}
```

```
class interface1
{      public static void main(String ar[])
       {          triangle t=new triangle();
                  circle c=new circle();
                  draw d;

                  d=t;
System.out.println("Area of Triangle“ +d.area(22,10));

                  d=c;
System.out.println("Area of Circle “+d.area(10,0));
       }
}
```

Interfaces Second Example

```
class student
{
        int rollno;

        void getno(int a)
        {
                rollno=a;
        }
        void putno()
        {
System.out.println(" Roll Number : "+rollno);
        }
}
```

```
class test extends student
{
    double m1,m2;

    void getmarks(double a,double b)
    {
          m1=a; m2=b;
    }
    void putmarks()
    {
    System.out.println("M1 : "+m1+" M2 : "+m2);
    }
}
```

Interfaces Second Example contd.,

```
interface sports
{
        final static double spwt=10;
        void putspwt();
}
class result extends test implements sports
{
        double total;
        public void putspwt()
        {
        System.out.println("Sports Marks Weightage : "+spwt);
        }
```

```
        void show()
        {           total=m1+m2+spwt;
                    putno();
                    putmarks();
                    putspwt();
        System.out.println("Total Marks :"+total);
        }
}
class interface2
{       public static void main(String ar[])
        {           result r=new result(); r.getno(786);
                    r.getmarks(78.5,65.25);  r.show();
        }
}
```

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword extends.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain

Interfaces Can be extended Example contd.,

```
interface A
{               void meth1();
                void meth2();
}
interface B extends A
{               void meth3();
}
class MyClass implements B
{               public void meth1()
                {
                System.out.println("Implement meth1().");
                }
```

```
                public void meth2()
                {
System.out.println("Implement meth2().");
                }
                public void meth3()
                {
      System.out.println("Implement meth3().");
                }
}
class interface3
{     public static void main(String arg[])
      {         MyClass ob = new MyClass();
                ob.meth1(); ob.meth2(); ob.meth3();
      }
}
```

Difference between Interfaces and Abstract Classes

Feature	Interface	Abstract class
Multiple inheritance	A class may implement several interfaces.	A class may extend only one abstract class.
Default implementation	An interface cannot provide any code at all, much less default code.	An abstract class can provide complete code, default code.
Constants	Static final constants only	Both instance and static constants are possible.

Packages

Packages

- Java provides a powerful means of grouping related classes and interfaces together in a single unit called “***packages*** “
- Include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- If we omit the **package** statement, the class names are put into the default package.

Package Syntax

- This is the general form of the **package** statement:

  package pkg;

  Here, *pkg* is the name of the package.

  Ex: *package MyPackage*;

- The **.class** files for any classes we declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.

Packages Example

```
package infotech;
class balance
{               String name;
                double bal;
                balance(String n,double b)
                {               name=n; bal=b;
                }
                void show()
                {               if(bal<0)
                                System.out.print(" --> ");
                System.out.println(name+" "+bal);
                }
}
```

```
class pack1
{
      public static void main(String arg[])
      {
      balance cur=new balance("Jeevan",22222.45);
                cur.show();
      }
}
```

Packages

- More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong.
- We can create a hierarchy of packages.
- Separate each package name from the one above it by use of a period.
- The general form of a multileveled package statement is shown here:

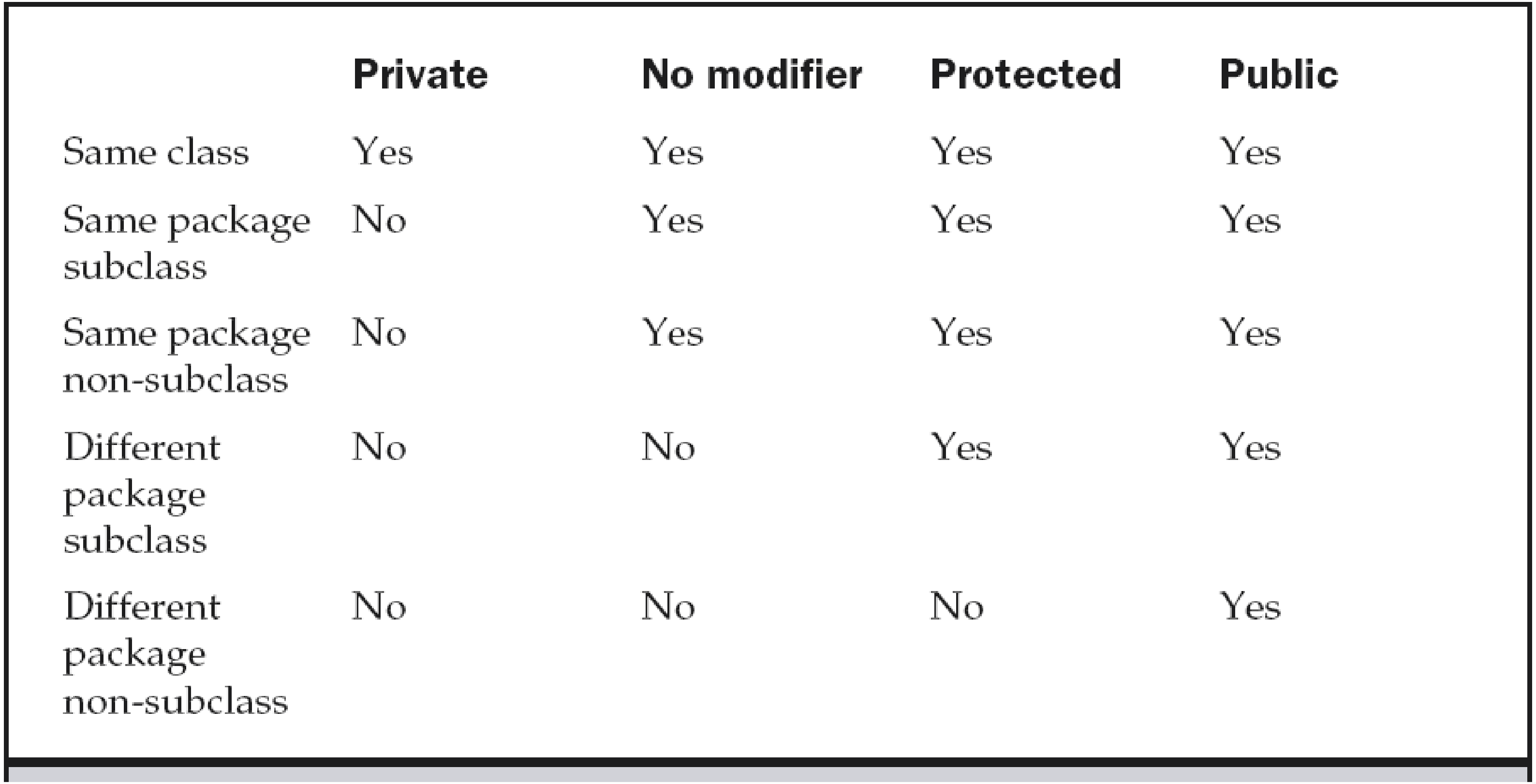
package pkg1[.pkg2[.pkg3]];

Ex: *package griet.it.oops;*

Access Protection

- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code.

Access Protection



	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Access Protection Example

```
package p1;
public class base
{       private int a=10;   int b=20;
        protected int c=30;
        public int d=40;
        public base()
        {
         System.out.println("Base Constructor ...");
                    System.out.println("a = "+a);
                    System.out.println("b = "+b);
                    System.out.println("c = "+c);
                    System.out.println("d = "+d);
        }
}
```

```
package p1;

class derived1 extends base
{
        derived1()
        {
        System.out.println("Derived 1 Constructor ...");
        //          System.out.println("a = "+a);
                    System.out.println("b = "+b);
                    System.out.println("c = "+c);
                    System.out.println("d = "+d);
        }
}
```

Access Protection Example

```
package p1;
class other1
{
        other1()
        {
                    base k=new base();
                    System.out.println("Other 1 Constructor ...");
        //          System.out.println("a = "+k.a);
                    System.out.println("b = "+k.b);
                    System.out.println("c = "+k.c);
                    System.out.println("d = "+k.d);
        }
}
```

```
package p1;

public class demo1
{
        public static void main(String ar[])
        {
                    base n=new base();
                    derived1 d1=new derived1();
                    other1 o1=new other1();
        }
}
```

Access Protection Example

```
package p2;
class derived2 extends p1.base
{
        derived2()
        {
        System.out.println("Derived 2 Constructor ...");
        //      System.out.println("a = "+a);
        //      System.out.println("b = "+b);
                System.out.println("c = "+c);
                System.out.println("d = "+d);
        }
}
```

```
package p2;
class other2
{
        other2()
        {
                p1.base k=new p1.base();
        System.out.println("Other 2 Constructor ...");
        //      System.out.println("a = "+k.a);
        //      System.out.println("b = "+k.b);
        //      System.out.println("c = "+k.c);
                System.out.println("d = "+k.d);
        }
}
```

Access Protection Example

```
package p2;
public class demo2
{
        public static void main(String ar[])
        {
                derived2 d2=new derived2();
                other2 o2=new other2();
        }
}
```

Importing Packages

- Java includes the ***import*** statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- In a Java source file, **import** statements occur immediately following the **package** statement and before any class definitions.

Importing Packages

- This is the general form of the **import** statement:

  import pkg1[.pkg2].(classname|);*

  Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**). There is no practical limit on the depth of a package hierarchy.

- Finally, we specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package.

  import java.util.Date; *import java.io.*;*

Importing Packages Example Contd.,

```
package mypack;
public class import1
{               String name;
                double bal;
                public import1(String n,double b)
                {               name=n; bal=b;
                }
                public void show()
                {
                                if(bal<0)
                                System.out.print(" --> ");
                System.out.println(name+" "+bal);
                }
}
```

```
import mypack.*;

class import2
{
      public static void main(String arg[])
      {
      import1 cur=new import1("Jeevan",33333.45);
                cur.show();
      }
}
```

Exceptions

Uncaught Exceptions

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

Uncaught Exceptions

- Any exception that is not caught by your program will ultimately be processed by the default handler.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.
- Here is the output generated when this example is executed.

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

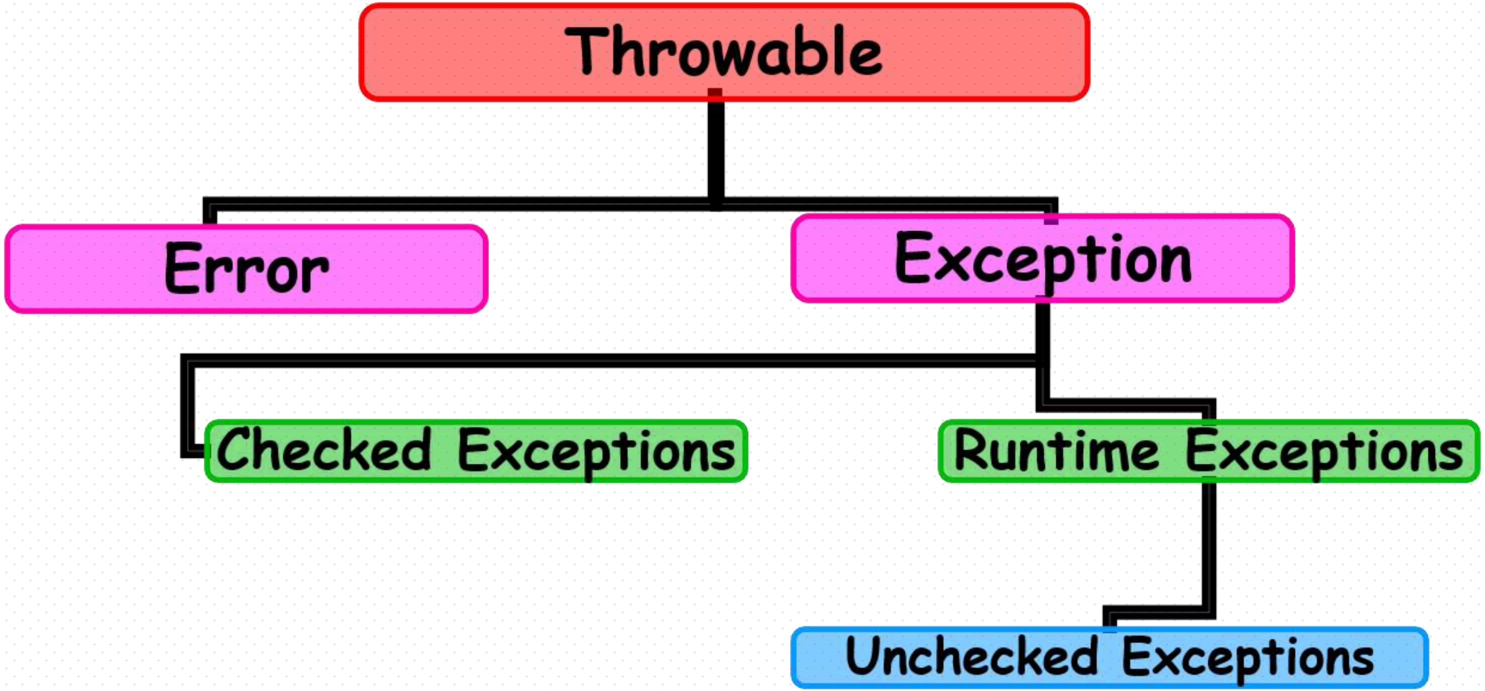
Uncaught Exceptions

```
class Exc1
{
        static void subroutine()
        {
                int d = 0;
                int a = 10 / d;
        }
        public static void main(String args[ ])
        {
                Exc1.subroutine();
        }
}
```

Uncaught Exceptions

- The resulting stack trace from the default exception handler shows how the entire call stack is displayed:
    - java.lang.ArithmeticException: / by zero
        - at Exc1.subroutine(Exc1.java:4)
            - at Exc1.main(Exc1.java:7)
- As you can see, the bottom of the stack is **main's line 7, which is the call to subroutine( ), which caused the exception at line 4.**

Exception Class Hierarchy



Exception Class Hierarchy

-> All exception types are subclasses of the built-in class **Throwable.**

-> **Throwable** is at the top of the exception class hierarchy.

-> Immediately below **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.

-> There is an important subclass of Exception, called **Runtime Exception.**

Error

- Error: When a dynamic linkage failure or other hard failures occurs then it throws an error.
- Simple programs can not catch or throw an error.
- Ex: Internal Error, Linkage error, Out of memory error, Stack Overflow error etc.

Exception

- Exception indicates that a problem occurred but it is not a serious problem. An exception is an abnormal condition that arises in a code sequence at run time.
- Runtime Exception: It is reserved for exceptions that indicate incorrect use of API.
- Unchecked Exceptions: Most exceptions are derived from Runtime Exception are automatically available and they need not be included in try – catch block or in any methods throws list.

Ex:

1. ArithmeticException
2. ArrayIndexOutOfBoundsException
3. ArrayStoreException
4. NegativeArraySizeException
5. NumberFormatException
6. NullPointerException

Exception-Handling Fundamentals

- A Java exception is an object that describes an exceptional 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.*

Exception-Handling Fundamentals

- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
- Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Manually generated exceptions are typically used to report some error condition to the caller of a method.

Try, Catch, throw, throws & Finally

- Java exception handling is managed via five keywords: **try, catch, throw, throws** and **finally.**
- Program statements that you want to monitor for exceptions are contained within a **try block.**
- 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.

Try, Catch, throw, throws & Finally

- System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw.**
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed before a method returns is put in a **finally block.**

Exception Handling Mechanism Syntax

This is the general form of an exception-handling block:

```
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 before try block ends
}
```

Unchecked Exceptions

```
class excep1
{               public static void main(String ar[])
                {
                                int a=10;
                                int b=0;
                                try
                                {               a=a/b;
                                System.out.println("This will not be printed");
                                }
                                catch(ArithmeticException ae)
                                {
                                System.out.println("Division by 0 error... change the value");
                                }
                                System.out.println("Quittting");
                }
}
```

Checked Exceptions

CheckedExceptions: CheckedExceptions must be included in try-catch or in a methods throws list.

Ex:

1. ClassNotFoundException
2. IOException
3. SQLException
4. NoSuchMethodException etc.,

Checked Exceptions

```
public class excep3
{
        public static void main(String ar[])
        {
                        try
                        {
                         Class c=Class.forName(ar[0].trim());
                         String name=c.getName();
                         Class sc=c.getSuperclass();
                         String sname=sc.getName();
                    System.out.println("Name is : "+name+" and SuperClass name is :"+sname);
                }
                catch(ClassNotFoundException cnf)
                {                   System.out.println("No such Class");
                }
        }
}
```

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, we 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.

Multiple Catch

```
public class excep4
{
        public static void main(String ar[])
        {       try
                {
                        int n=ar.length;
                        int x[]=new int[-3];
                         if(n>0)
                         {
                                        int m=Integer.parseInt(ar[0]);
                                        System.out.println("Given number is : "+m/(m-n));
                         }
                }
                catch(NumberFormatException nfe)
                {
                        System.out.println("Number format exceptionnnn: "+nfe);
                }
```

Multiple Catch (contd.,)

```
        catch(ArrayIndexOutOfBoundsException ai)
        {
                        System.out.println("Array index is out of range ");
        }
        catch(ArithmeticException ae)
        {
            System.out.println("Divide by 0 errorrrr");
        }
        catch(NegativeArraySizeException nase)
        {
            System.out.println("Negative size........");
        }
        catch(Exception e)
        {
            System.out.println("Exceptionnnn");
        }
    }
}
```

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.

Nested Try

```
class NestTry
{       public static void main(String args[])
        {       try
                {       int a = args.length;
                        int b = 12 / a;
                        System.out.println("a = " + a);
                        try
                        {       if(a==1) a = a/(a-a);
                                String s=args[4];
                        }
                        catch(ArrayIndexOutOfBoundsException e)
                        {  System.out.println("Array index out-of-bounds: " + e);
                        }
                }
                catch(ArithmeticException e)
                {       System.out.println("Divide by 0: " + e);
                }
        }
}
```

throw

- We have only been catching exceptions that are thrown by the Java run-time system.
- However, it is possible for our program to throw an exception explicitly, using the “throw” statement.
- Syntax: throw ThrowableInstance;
- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

throw

- Simple types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.
- There are two ways we can obtain a Throwable object: using a parameter into a catch clause, or creating one with the new operator.
- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.

Throw (Predefined Exceptions)

```
class ThrowDemo
{               static void throwfunction()
                {               try
                                {               throw new ArithmeticException();
                                }
                                catch(ArithmeticException e)
                                { System.out.println("Caught inside throwfunction.");
                                throw e; // rethrow the exception
                                }
                }
                public static void main(String args[])
                {               try
                                { throwfunction();
                                }
                                catch(ArithmeticException e)
                                {               System.out.println("Recaught: " + e);
                                }
                }
}
```

Throw (User-defined Exceptions)

```
class userdefined
{     static void throwfun(int a) throws UserException
      {  if(a<0)
                  throw new UserException(a);
                  System.out.println("Normal Exit");
       }
                  public static void main(String args[])
                  {               try
                                  {               throwfun(10);
                                                  throwfun(-5);
                                  }
                                  catch(UserException ue)
                                  {
System.out.println("Exception caught: " + ue);
                                  }
                  }
}
```

```
class UserException extends Exception
{
                  private int x;

                  UserException(int a)
                  {
                                  x=a;
                  }

                  public String toString()
                  {
return "IT students raised this exception "+x;
                  }
}
```

Throws clause

- 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.
- We 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.

Throws clause

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.*

Throws Clauses

```
class throwsdemo
{
        static void proc() throws ClassNotFoundException
        {
                Class c=Class.forName("java.lang.Math");
                if(c==null)
                        throw new ClassNotFoundException();
                System.out.println("Class Found" +c);
        }
        public static void main(String ar[])
        {       try
                {       proc();
                }
                catch(ClassNotFoundException cnf)
                {       System.out.println("Class not foundddd");
                }
        }
}
```

Finally Clause

- finally creates a block of code that will be executed after a try/catch block has completed
- The finally block will execute whether or not an exception is thrown.
- If an exception is thrown, the finally block will execute even if no catch statement matches the exception.

Finally Clause

- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method.
- The **finally clause is optional. However, each try** statement requires at least one **catch or a finally clause.**

Finally

```
class FinallyDemo
{
                static void procA()
                {
                                try
                                {
System.out.println("inside procA");
throw new RuntimeException("demo");
                                }
                                finally
                                {
System.out.println("procA's finally");
                                }
                }
```

```
static void procB()
{
                try
                {
System.out.println("inside procB");
                                return;
                }
                finally
                {
System.out.println("procB's finally");
                }
}
```

Finally

```
static void procC()
{
                    try
                    {
          System.out.println("inside procC");
                    }
                    finally
                    {
          System.out.println("procC's finally");
                    }
}
```

```
    public static void main(String args[])
    {
           try
           {
                     procA();
           }
           catch (Exception e)
           {
       System.out.println("Exception caught");
           }
                     procB();
                     procC();
     }
}
```

End of Unit III