Session 12 – Inheritance

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Inheritance Basics
- ✓ Types of Inheritance
- ✓ IS A Relationship
- ✓ Overriding
- ✓ Co-Variant Return Types

Inheritance

Introduction to Inheritance & its Basics

One of the primary reasons for adopting the object oriented approach was the ability to reuse the code. Well organized classes can be used in new applications easily. However we can achieve an even greater degree of reusability and extendibility with the feature "inheritance".

Consider a case where we wish to represent several related real-world objects that have some distinguishing qualities.

- A variety of employee positions(like Manager, Programmer and Clerk etc.) with different responsibilities and privileges.
- Different type of Accounts(like SBAccount,CDAccount,RDAccount) in the bank
- A line of cars with different options.

These situations are characterized by the fact that there is a set of qualities common to all of objects in question, but that subsets of these objects have qualities not shared by the others. We can write classes for each of the different kinds of objects, but we would end up duplicating lot of code.

Instead, we can use "Inheritance". Inheritance allows us to extend the existing class whenever there is a need of extending it. So here we can write a common class representing the common qualities of all objects and then create additional class extending (inheriting) the common class. This process of starting with something general and creating a refined version of it is known as "Specialization".

Imp Note: Inheritance between classes should satisfy the basic relation "IS A", which defines the specialization from generalization. Writing inheritance without the "IS A" relation between the classes does not make any sense.

Let us understand few commonly used terms under Inheritance;

Super Class : The class from which we inherit is called super class. Also called as Base class or Parent class.

Sub Class: The class to which we inherit is called as sub class. Also can be called as child class or derived class.

Note: With Inheritance all members and methods of super class(except private) will be available to sub class. So we can make use of those methods in the sub class as if they are existing in the sub class.

Syntax:

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

Here B is the sub class and A is the super class.

```
classSBAccount extends Account{

}
```

```
booleanchqFacility;
   String nominee;
publicSBAccount(inta,Stringt,doubleb,booleancf,String n){
          super(a,t,b);
          chqFacility=cf;
          nominee=n;
   }}
```

Imp to Note:

1. Private members of the superclass are not inherited by the subclass and can only be indirectly accessed.

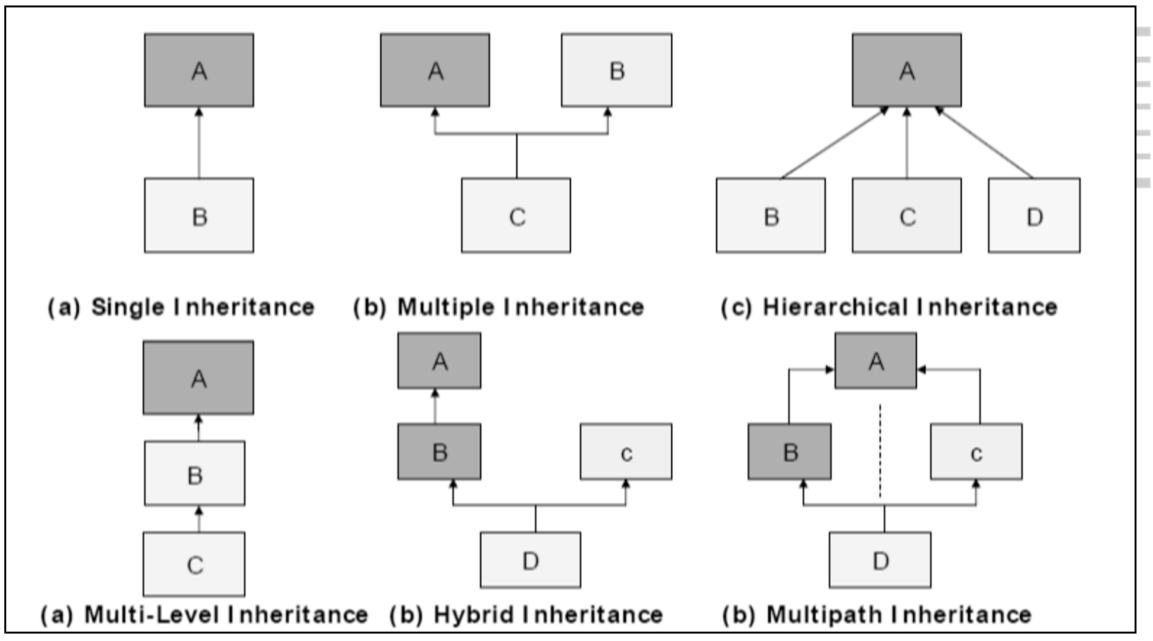
2. Members that have default accessibility in the superclass are also not inherited by subclasses in other packages, as these members are only accessible by their simple names in subclasses within the same package as the superclass.

3. Since constructors and initializer blocks are not members of a class, they are not inherited by a subclass.

4. A subclass can extend only one superclass

Different Types of Inheritances

- Simple Inheritance
- Hierarchical Inheritance
- Multiple Inheritance (Not supported directly)
- Multilevel inheritance
- Hybrid Inheritance (Not supported)



Relationships in Java

Java represents 2types of relationships –

- IS-A relationship
- HAS-A relationship

IS-A Relationship: - In object oriented programming, the concept of IS-A is a totally based on Inheritance (extends) and Interface implementation (implements).It is just like saying "A is a B type of thing".

It is key point to note that you can easily id entify the IS-A relationship .Wherever you see an extends keyword or implements keyword in a class declaration, then this class is said to be passed IS-A relationship.

HAS-A Relationship:-HAS-A relationship has nothing to do with Inheritance rather it is based on the usage of various variables and methods of other class. We can say "A HAS-A B if the code in class A has reference to an instance of B".

Overriding

This is achieved when a subclass rewrites instance method defined in the superclass. The sub class method definition must have the same method signature and return type.

Note: The sub class method definition cannot narrow the accessibility of the method, but it can widen it. i.e., the access specifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass. The sub class method definition can only specify all or none, or a subset of the exception classes (including their subclasses) specified in the throws clause of the overridden method in the super class.

Overriding Example:

```
classSBAccount extends Account{
booleanchqFacility;
   String nominee;
publicSBAccount(inta,Stringt,doubleb,booleancf,String n){
        super(a,t,b);
        chqFacility=cf;
        nominee=n;
}public void withdraw(double amt){
        if (chqFacility){
        if ((balance-amt)>=5000){
        balance-=amt;
        System.out.println("Balance after withdrawl is "+balance);
        }else
        System.out.println("Sorry chq book a/cs should maintain min of 5k");
        }else
        super.withdraw(amt);
}public static void main(String args[]){
        SBAccount sa1=new SBAccount(101,"Nimai1",100000,true,"Krsna");
        SBAccount sa2=new SBAccount(102,"Nimai2",100000,false,"Krsna");
        sa1.deposit(53000);
        sa1.withdraw(23000);
        sa1.withdraw(127000);
        sa2.deposit(53000);
        sa2.withdraw(23000);
        sa2.withdraw(127000);}}
```


Note: Though sub class overrides the super class withdraw() method,stillit is accessible to sub class. We can invoke the super class withdraw() implementation using "super" keyword as shown in the above example.

Co-Variant Return Types

A covariant return type allows to override a super class method that returns a type that sub class type of super class method's return type. It is to minimize up casting and down casting.

The following code snippet depicts the concept:

```
class Parent
{
      Parent sampleMethod()
      {
         System.out.println("Parent sampleMethod() invoked");
         return this;
      }
}
class Child extends Parent
{
```

```
    Child sampleMethod()
    {
        System.out.println("Child sampleMethod() invoked");
        return this;
    }
}
class Covariant
{
    public static void main(String args[])
    {
      Child child1 = new Child();
      Child child2 = new child1.sampleMethod();
      Parent parent1 = child1.sampleMethod();
    }
}
```

Session 13 – InheritanceContd...

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Constructor Calling Order
- ✓ Object Type Casting
- ✓ Up & down Casting
- ✓ Early & Late Binding
- ✓ Dynamic MethodDispatch

Inheritance Contd...

Constructor Calling Sequence

Let us look into the Constructor Calling Sequence and understand through the code blocks about how the Calling Sequence works.

Scenarios of "Constructor" calling sequence:

Ex 1:

```
class First{
public First(){
        System.out.println("First");
  }
}
class Second extends First{
public Second(){
        System.out.println("Second");
  }
}
```

What will be the output of the command : Second s=new Second();

Output: First
Second

Description: Here when you are not explicitly making a call to super class constructor, system automatically calls the super class constructor(only zero parameter one) to initialize the members of super class. Because of that it is displaying also "First" along with "Second".

Ex 2 :

```
class First{
public First(int x){
        System.out.println("First");
  }
}
class Second extends First{
public Second(){
        System.out.println("Second");
  }}
```

What will be the output of the command : Second s=new Second();
Output: compilation error. Second class will not be compiled.
Description: As i said for the ex1, either you call the super class constructor explicitly or let the system do it. In the "First" class you have written an explicit constructor with int argument. This cannot be called automatically by system. System can only make a call to the "No Parameter" constructor of the super class by default. You are breaking the rule "Either you call it or let system call it".

Ex 3 :

```
class First{
public First(){
        System.out.println("First");
  }
public First(int y){
        System.out.println(y);
  }
}
class Second extends First{
public Second(){
        System.out.println("Second");
        super(100);
  }
}
```


What will be the output of the command : Second s=new Second();
Output: compilation error. Second class will not be compiled.
Description: "super" can be written only as first statement. It cannot be used in 2nd or 3rd statement.

Ex 4 :

```
class First{
public First(){
        System.out.println("First");
  }
public First(int y){
        System.out.println(y);
  }
}
class Second extends First{
public Second(){
        super(100);
        this(200);
        System.out.println("Second");
}public Second(int x){
        System.out.println(x);
  }
}
```

Output: Compilation error at second class . Either "super" or "this" only can be used as first statement. Both cannot be used in on single constructor.

Ex 5:

```
class First{
public First(){
        System.out.println("First");
  }
public First(int y){
        System.out.println(y);
  }
}
class Second extends First{
public Second(){
        this(200);
        System.out.println("Second");
  }
public Second(int x){
        super(100);
        System.out.println(x);
  }
}
```

What will be the output of the command : Second s=new Second();

Output:

100
200
Second

Object Type Casting-

In java object typecasting one object reference can be type cast into another object reference. The cast can be to its own class type or to one of its subclass or superclass types or interfaces. There are compile-time rules and runtime rules for casting in java.

The casting of object references depends on the relationship of the classes involved in the same hierarchy. Any object reference can be assigned to a reference variable of the type "Object", because the "Object" class is a supermost class of every Java class.

There are two casting scenarios

1. Upcasting

When we cast a reference along the class hierarchy in a direction from the sub classes towards the root, it is an upcast. We need not use a cast operator in this case.

Implicit Casting

In general an implicit cast is done when an Object reference is assigned (cast) to:

- A reference variable whose type is the same as the class from which the object was instantiated.
- A reference variable whose type is a super class of the class from which the object was instantiated.
- A reference variable whose type is an interface that is implemented by the class from which the object was instantiated.
- A reference variable whose type is an interface that is implemented by a super class of the class from which the object was instantiated.

Upcasting code snippet

```
Account act; //super class reference
SBAccountsa=new SBAccount(....);        //Sub class object
act=sa;  //upcasting
act.withdraw(5000);      // will call sub class implementation of withdraw()
```

Why the above line is calling sub class flavour despite of being called on super classreference? Even for the same code it calls super/base class flavour of withdraw(). Whyit is so? Here we need to understand the binding concepts to get the clarity.

Connecting a method call to a method body is called binding. When binding is performed before the program is run (by the compiler and linker, if there is one), it's called early binding. You might not have heard theterm before because it has never been an option with procedural languages.
C compilers have only one kind of method call, and that's early binding. The confusing part of the above program revolves around early bindingbecause the compiler cannot know the correct method to call when it has only an Instrument handle.

The solution is called **late binding**, which means that the binding occurs at run-time based on the type of object. Late binding is also called dynamic binding or run-time binding.
When a language implements late binding, there must be some mechanism to determine the type of the object at run-time and to call the appropriate method. That is, the compiler still doesn't know the object type, but the method-call mechanism finds out and calls the correct method body.
The late-binding mechanism varies from language to language, but you can imagine that some sort of type information must be installed in the objects.
All method binding in Java uses late binding unless a method has been declared final. This means that you ordinarily don't need to make any decisions about whether late binding will occur – it happens automatically.
By construction C++ follows early binding and java follows late binding default(Dynamic method dispatch). However we have late binding in C++ with virtual methods. And we have early binding for static and final methods in java.

Downcasting
When we cast a reference along the class hierarchy in a direction from the root class towards the children or subclasses, it is a downcast.
The compile-time rules are there to catch attempted casts in cases that are simply not possible. This happens when we try to attempt casts on objects that are totally unrelated (that is not subclass super class relationship or a class-interface relationship) At runtime a
ClassCastException is thrown if the object being cast is not compatible with the new type it is being cast to.
Explicit Casting syntax is required when we are doing down casting.
instanceof Operator
The instanceof operator is called the type comparison operator, lets you determine if an object belongs to a specific class, or implements a specific interface. It returns true if an object is an instance of the class or if the object implements the interface, otherwise it returns false.
Downcasting and "instanceof" operator is explained with the followingExample.
There are three classes 1.SBAccount 2.CDAccount and 3. RDAccount inheriting "Account" class. All these classes have some special membersand special methods in addition to the methods and members inherited fromits super class.

GenericTransaction.java

```
public class GenericTransaction{
public void action(Account a){
        a.deposit(5000);
        if (a instanceofSBAccount ){
        SBAccountsa=(SBAccount)a;
          //assuming callnterest() is the method of SBAccount
        sa.callnterest(5000);
        }else if (a instanceofCDAccount){
        CDAccountca=(CDAccount)a;
          //assuming calODIerest() is the method of CDAccount
```

```
            ca.calODInterest();
            }
  }
public static void main(Strig[] args){
            GenericTransactiongt=new GenericTransaction();
            SBAccountsa=new SBAccount();
            gt.action(sa);
            CDAccountca=new CDAccount();
            gt.action(ca);
  }
}
```

Dynamic Method Dispatch

In dynamic method dispatch,super class refers to subclass object and implements method overriding.Dynamic dispatch is a mechanism by which a call to Overridden function is resolved at runtime rather than at Compile-time, and this is how Java implements Run time Polymorphism.

The following code block helps in understanding better –

```
class Student {
void which() {
System.out.println("A Brilliant Student.");
}
}
class Avg extends Student {
void which() {
System.out.println("Avg");
}
}
class Excel extends Student {
void which() {
System.out.println("Excel.");
}
}
class Test {
public static void main(String[] args) {
Student ref1 = new Student();
Student ref2 = new Avg();
Student ref3 = new Excel();
ref1.which();
ref2.which();
ref3.which();
}
}
```


Session 14 – Abstract Classes & Interfaces

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Hierarchical Abstraction
- ✓ Final Classes & Methods
- ✓ Writing Abstract Classes
- ✓ Interfaces
- ✓ Why Interfaces

Abstract Classes & Interfaces

What do we understand by Hierarchical Abstraction?

When thinking about an abstraction hierarchy, we mentally step up and down the hierarchy, automatically zeroing in on only the single layer or subset of the hierarchy (known as a **subtree**) that is important to us at a given point in time.The simpler an abstraction - that is, the fewer features it presents - the more general it is, and the more versatile it is in describing a variety of real-world situations. The more complex an abstraction, the more restrictive it is, and thus the fewer situations it is useful in describing.

Take, for example, the rules we might define for what constitutes a bird: namely, something which:

- Has feathers
- Has wings
- Lays eggs
- Is capable of flying

Given these rules, neither an ostrich nor a penguin could be classified as a bird, because neither can fly.

If we attempt to make the rule set less restrictive by eliminating the 'flight' rule, we are left with:

- Has feathers
- Has wings
- Lays eggs

According to this rule set, we now may properly classify both the ostrich and the penguin as birds.

This rule set is still unnecessarily complicated, because as it turns out, the 'lays eggs' rule is redundant: whether we keep it or eliminate it, it doesn't change our decision of what constitutes a bird versus a non-bird. Therefore, we simplify the rule set once again:

- Has feathers
- Has wings

We try to take our simplification process one step further, by eliminating yet another rule, defining a bird as something which:

- Has wings

Final Classes & Methods

Restricting Inheritance

The "final" before class makes the class non-inheritable.i.e once classis defined final we cannot inherit that class, but we can create objects andwork with that methods.

Restricting Overriding

The "final" before method makes the method non-overridable. Final methods follow the early binding. Final methods work faster thannormal methods.

Constants

The "final" before member makes the member constant.

Ex:

```
public static final int red=1;
public static final int blue=2;
```

As constants value will not be changed, we dont need to keep separatecopies for every instance-So we often define the constants with static.From 1.5 version java supports "enum" concept. Enum can be used to defineenumerated constants.

Writing Abstract Classes

Let us see how we write the Abstract Classes& understand what they are all about;

Abstract classes are used to declare common characteristics of subclasses. Abstract classes are used to provide a template for concrete subclasses down the inheritance tree.

Abstraction brings greater generality and conceptual clarity to class design. We use abstract classes to define broad types of behaviors at the top of object oriented programming class hierarchy, and use its subclasses to provide implementation details of abstract class.

In java classes are two types

1. Concrete Classes **2. Abstract Classes**

Abstract Classes

Simple Example:

```
abstract class Account{
privateintano;
private String title;
private double balance;
public Account(..){
        ....
  }
public void deposit(double amt){
        balance+=amt;
        System.out.println("Balance after deposit is "+balance);
  }
  // writing abstract method
public abstract void withdraw(double amt);
}
```


Important Points:

- Abstract class cannot be instantiated. They enforce a protocol on their sub classes.
- Like concrete class abstract class can encapsulate i.e. can have members, methods with code and along with it can have set of abstract methods.
    - Abstract class is only meant for inheritance.
- Once Abstract class is inherited to a class, the sub class must implement/override the methods. However methods can be left without the code if the class is abstract again.

Another standard example what we see in most of the books is also good to understand the abstract classes. Here

```
public class Shape {
doublegetArea ()
{ return 0.0;} }
public class Rectangle extends Shape {
doubleht = 0.0;
doublewd = 0.0;
public double getArea ()
{ return (ht*wd);
```

```
public void setHeight(double ht)
{ this.ht = ht; }
public void setWidth (double wd)
{ this.wd = wd; }
}
public class Circle extends Shape {
double r = 0.0;
public double getArea ()
{ return (3.14 * r * r); }
public void setRadius (double r)
{ this.r = r; }
}
```

Interfaces

It defines a protocol/contract. Interface defines the specification of a behavior. It will not contain implementation.Interfaces define the one of the public interface(face) of an objectwithout revealing its implementation.

- The implementation can change without affecting the caller of the interface.
- The caller does not need the implementation at the compile tim
  - Caller needs only interface at the compilation time.
  - During runtime,actual object instance is associated with the interface type.

Syntax:

```
public interface <interface>{
        ---constants---
        ---Method definitions---
 }
```


Ex:

```
public interface Transaction{
        public void deposit(double amt);
        public void withdraw(double amt);
  }
```

Note: Unlike class interface cannot encapsulate. It cannot have instancemembers or instance methods. It can only have constants and abstract methods.

- By default the interface methods are abstract.

Why do we need Interfaces?

- Interfaces are mainly used to provide polymorphic behavior.
- Interfaces function to break up the complex designs and clear the dependencies between objects.

Session 15 –Interfaces

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Interface Implementations
- ✓ Interface Features
- ✓ Interface Casting
- ✓ Interface Inheritance

Interfaces

Interface Implementations

- Interface reference can point to any object of the class where interface is implemented.
- Through interface we can only call methods which are defined in interface i.e. we cannot refer any method of the class object to which interface is pointing unless that method is defined in the interface.
- Once interface is implemented on class we should implement all the methods of the interface. If the class is abstract then we can leave methods without implementation.
- A Class can implement any number of interfaces and an interface can be implemented by any number of classes.

Often there is a confusion in choosing between abstract classes and interfaces for a scenario. Following table gives you the clarity to draw a line between interfaces and abstract classes.

Abstract Classes vs Interfaces

- Interfaces are often used to describe the peripheral abilities of a class, and not its central identity, E.g. an Account class might implement the "**Transaction**" interface, which could apply to many otherwise totally unrelated objects.
- Abstract classes are useful in a situation when some general methods should be implemented and specialization behavior should be implemented by subclasses. Interfaces are useful in a situation when all its properties need to be implemented by subclasses
- If the various objects are all of-a-kind, and share a common state and behavior, then tend towards a common base class. If all they share is a set of method signatures, then tend towards an interface.

We can also consider the following points

An interface is also used in situations when a class needs to extend another class apart from the abstract class. In such situations it is not possible to have multiple inheritances of classes. An interface on the other hand can be used when it is required to implement one or more interfaces. Abstract class does not support Multiple Inheritance whereas an Interface supports multiple inheritances.

The problem with an interface is, if you want to add a new feature (method) in its contract, then you **MUST** implement those methods in all of the classes which implement that interface. However, in the case of an abstract class, the method can be simply implemented in the abstract class and the same can be called by its subclass

Note: There is no difference between a fully abstract class (all methods declared as abstract and all fields are public static final) and an interface.

Note: Interfaces are slow as it requires extra indirection to find corresponding method in the actual class. Abstract classes are fast.

Interfaces - Miscellaneous

"Multiple Inheritance" in Java

The interface isn't simply a "more pure" form of abstract class. It has a higher purpose than that. Because an interface has no implementation at all—that is, there is no storage associated with an interface—there's nothing to prevent many interfaces from being combined. The act of combining multiple class interfaces is called multiple inheritance, and it carries some rather sticky baggage because each class can have an implementation. In Java, you can perform the same act, but only one of the classes can have an implementation, so the problems seen in C++ do not occur with Java when combining multiple interfaces:

In a derived class, you aren't forced to have a base class that is either an abstract or "**concrete**" (one with no abstract methods). If you do inherit from a non-interface, you can inherit from only one. All the rest of the base elements must be interfaces. You place all the interface names after the implements keyword and separate them with commas. You can have as many interfaces as you want; each one becomes an independent type that you can upcast to.

Grouping constants

Because any fields you put into an interface are automatically static and final, the interface is a convenient tool for creating groups of constant values, much as you would with an enum in C or C++. For example:

```
public interface Weeks{
int
   MONDAY = 1, TUESDAY = 2, WEDNESDAY = 3,
   THURSDAY = 4, FRIDAY = 5, SATURDAY = 6, SUNDAY = 7;
}
```

The fields in an interface are automatically public, so it's unnecessary to specify that.

Initializing fields in interfaces

Fields defined in interfaces are automatically static and final. These cannot be "blank finals," but they can be initialized with nonconstant expressions. For example:

```
importjava.util.*;
public interface RandVals {
  Random rand = new Random();
intrandomInt = rand.nextInt(10);
}
```

Since the fields are static, they are initialized when the class is first loaded, which happens when any of the fields are accessed for the first time. The fields, of course, are not part of the interface but instead are stored in the static storage area for that interface.

Note: Interfaces may be nested within classes and within other interfaces.

Interface Inheritance

- Interfaces allow us to implement callback mechanism.
- It allows us to have unrelated classes implement similar methods.we can model multiple inheritance.
- Interfaces are useful to implement dynamic polymorphism.

Session 16 –Polymorphism

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Polymorphism Features
- ✓ Polymorphism Implementation
- ✓ Polymorphism with Inheritance
- ✓ Polymorphism with Interfaces

Polymorphism

Polymorphism means one name, many forms.That means, Polymorphism is the ability of one object to be treated and used like other object. Polymorphism gives us the ultimate flexibility in extensibility.

There are 3 distinct forms of Java Polymorphism;

- Method overloading (Compile time polymorphism)
- Method overriding through inheritance (Run time polymorphism)
- Method overriding through the Java interface (Run time polymorphism)

Run time Polymorphism allows a reference to point objects of different types at different times during execution. A super type reference exhibits polymorphic behavior, since it can denote objects of its subtypes.

Polymorphism Implementation

Polymorphism generally means the ability to appear in many forms. However, in object-oriented programming, polymorphism refers to a programming language's ability to process objects differently

depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes. For example, given a base class shape, polymorphism enables the programmer to define different circumference methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the circumference method to it will return the correct results. Polymorphism is considered to be a requirement of any true object-oriented programming language (OOPL). The Shape class can be implemented in the below manner:-

```
abstract class Shape {
abstract double calcCircumference ();
}
class Rectangle extends Shape {
doublex,y ;
public Rectangle (double x, double y) {
this.x = x;
this. y = y;
}
doublecalcCircumference  ( ) {
doublecircum = 2.0 * x + 2.0 * y ;
return (circum);
  }
}
class Circle extends Shape {
double radius;
public Circle (double radius ) {
this.radius  = radius ;
}
doublecalcCircumference ( ) {
doublecircum = 2.0 * 3.1415 * radius;
return (circum);
  }
}
```


Given the Inheritance Hierarchy, Polymorphism allows to hold a reference to a Circle object in a Variable of type Shape, as in the below class:

```
class Test   {
public static void main  (String args[]  ) {
Shape shape1 = new Rectangle (5.0, 10.0);
Shape shape2 =  new Circle (20.0);
GiveCircum.printCircum (shape1);
GiveCircum .printCircum  (shape2);
}
}
classGiveCircum    {
public static void  printCircum (Shape s) {
double result =  s. calcCircumference ()  ;
System.out.println ("hello - circumference = " + result ) ;
}
}
```

Polymorphism with Interface

How Interfaces Increase Polymorphism

In the event that you want to have a method or set of methods be common between several families of classes, rather than only among one family, using an interface can stretch the ability of polymorphism over many families of classes.

For example, remember the abstract class "Account" which we have discussed in previous chapter, it contains two abstract methods called deposit() and withdraw(). Imagine that you want these two methods to be available to more than just the Account family of classes(SBAccount/CDAccount..etc). You can create an interface that contains the the above two methods and tell the class to implement that interface.

```
public interface Shape {
public double area();
public double volume();
}
class Cube implements Shape {
        int x = 10;
        public double area( ) {
        return (6 * x * x);
        }
        public double volume() {
                return (x * x * x);
        }
}
class Circle implements Shape {
        int radius = 10;
        public double area() {
                return (Math.PI * radius * radius);
        }
        public double volume() {
                return 0;
        }
}
public class PolymorphismTest {
        public static void main(String args[]) {
        Shape[] s = { new Cube(), new Circle() };
        for (int i = 0; i <s.length; i++) {
                System.out.println("The area and volume of " + s[i].getClass() + " is " + s[i].area() + " , " +
s[i].volume());
          }
        }
}
```


Output

The area and volume of class Cube is 600.0 , 1000.0
The area and volume of class Circle is 314.1592653589793 , 0.0

The methods area() and volume() are overridden in the implementing classes. The invocation of the both methods area and volume is determined based on run time polymorphism of the current object as shown in the output.