UNIT 1: INHERITANCE

Unit Structure

1.0 Learning Objectives

After learning this unit, you will be able to:

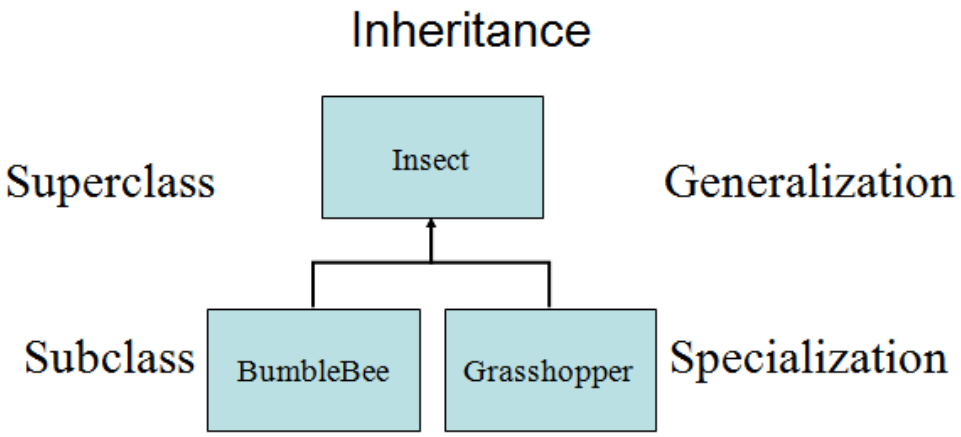
- Explain the concept of inheritance
- Define Polymorphism
- Compile time and Runtime
- Illustrate super keyword and final keyword

1.1 Introduction

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order.

Java Inheritance defines a relationship between a superclass and its subclasses. This means that an object of a subclass can be used wherever an object of the superclass can be used. Class Inheritance in java mechanism is used to build new classes from existing classes. The inheritance relationship is transitive: if class x extends class y, then a class z, which extends class x, will also inherit from class y.

1.2 Concept of Inheritance



Concept of Inheritance

"Inheritance is one of the cornerstones of OOP because it allows for the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related objects, that is, objects with common attributes and behaviors. This class may then be inherited by other, more specific classes", each adding only those attributes and behaviors that are unique to the inheriting class.

Need of Inheritance

The various needs of inheritance are given below:

1. Closer to Real-World
2. Code Reusability
3. Transitive Nature

As stated earlier, inheritance leads to the definition of generalized classes that are at the top of an inheritance hierarchy, thus it is an implementation of

generalization. This feature available in C++ makes the data and methods of a Superclass or base class available to its subclass or derived class. It has many advantages, the most important of that is the reusability of code. Once a class has been created, it can be used to create new subclasses.

Generalisation/ Specialisation

A class that is inherited is referred to as a base class. The class that does the inheriting is referred to as the derived class. Each instance of a derived class includes all the members of the base class. The derived class inherits all the properties of the base class. Therefore, the derived class has a large set of properties than its base class. However, a derived class may override some of the properties of the base class.

To inherit a class, the definition of one class can be incorporated into another by using the extends keyword. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. The syntax of inheriting class is given below:

Syntax:

<accessspecifier> class <class name(Subclass)> extends <class name(Superclass)>

For example,

Public class B extends A

Public class C extends B

Public Class D extends C

The public data members and methods (Except constructors) in the superclass are inherited by the subclass, i.e., their definitions are copied into the subclass's class definition. No members of the subclass are visible to the superclass.

Now let us consider an example to illustrate the same:

//Create a superclass

```
classSuperClass
{
intx,y;
voidshowXY( )
{
System.out.println (“x and y” + x + “ ” + y);
}
}
```

//Create a subclass by extending class A

```
classSubClass extends SuperClass
{
int z;
voidshowZ( )
{
System.out.println (“z is:” + z);
}

voidsum( )
{
System.out.println (“x+y+z” + (x + y+ z));
}
```

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

SuperClasssuperob = new SuperClass ( );
SubClasssubob= new SubClass ( );

//The superclass can refer itself

superob.x=10;
superob.y=7;
System.out.println (“Contents of Superclass”)
superob.showXY( );

//The subclass has access to all public members of its superclass

subob.x=14;
subob.y=16;
subob.z=20;

System.out.println (“Contents of subclass”)
subob.showXY( )
subob.showZ( );
System.out.println ( );
System.out.println (“Sum of x, y and z in subclass”);
subob.sum( );
```

```
}
}
```

The output of above program is given below:

```
Contents of superclass

X and y10 7

Contents of subclass

x and y 14 16
z is: 20

Sum of x, y and z in subclass
50
```

As it is given in the above program, the subclass SubClass includes all of the members of its superclass SuperClass. That is why subob can access x and y and call showXY( ). Also, inside add ( ), x and y can be referred to directly, as if they were part of SubClass. Although a subclass includes all of the members of its superclass, it cannot access members of the superclass that have been declared as private.

Check your progress 1

1. Write the syntax of inheriting a class.
2. Explain Implicit Subclass to Super class Conversion with the help of an example.

.....

.....

.....

.....

.....

.....

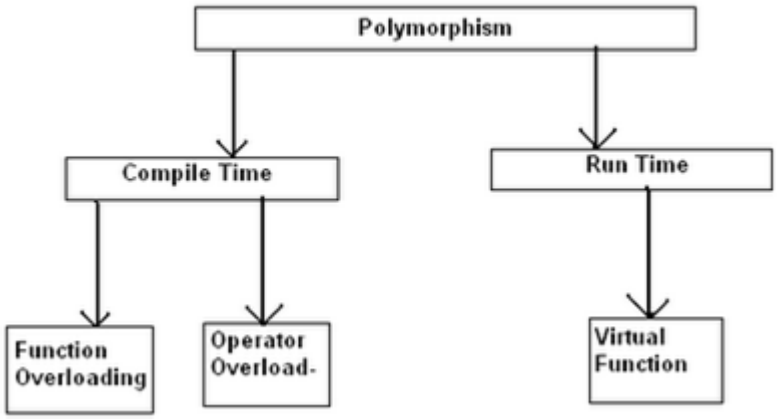
.....

.....

.....

.....

1.3 Polymorphism



"Polymorphism allows one interface to be used for a set of actions i.e. one name may refer to different functionality. Polymorphism allows an object to

accept different requests of a client (it then properly interprets the request like choosing appropriate method) and responds according to the current state of the runtime system, all without bothering the user."

There are two types of polymorphism:

1. Compile-time polymorphism
2. Runtime Polymorphism

Compile time Polymorphism

In compile time Polymorphism, method to be invoked is determined at the compile time. Compile time polymorphism is supported through the method overloading concept in java.

Method overloading means having multiple methods with same name but with different signature (number, type and order of parameters).

Here is the code of the example:

```
class One
{
public void funOne (int a)
{
System.out.println ("The value of class A is:" + a);
}
public void funOne (int a, int b)
{
System.out.println ("The value of class B is:" + a + "and" + b);
}
}
public class PolyOne
{
public static void main (String [ ] args)
{
One obj=new One ( );
```

```
//Here compiler decides that funOne (int) is to be called and “int” will be printed.
obj.funOne (20);
//Here compiler decides that funOne (int, int) is to be called and “int and int” will be printed.
obj.funOne (20, 30);
}
}
```

The output of above program is given below:

The value of class A is: 20

The value of class B is: 20 and 30

Runtime Polymorphism:

In runtime polymorphism, the method to be invoked is determined at the run time. The example of run time polymorphism is method overriding which is also called dynamic method dispatch is explained below:

Method Overriding:

If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final.

Overriding means redefining a method in an inheritance hierarchy. 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.

The benefit of overriding is: ability to define a behavior that's specific to the sub class type which means a subclass can implement a parent class method based on its requirement.

In object oriented terms, overriding means to override the functionality of any existing method.

Example:

Let us look at an example.

```
class Animal
{
```

```
public void eat ( )
{
System.out.println (“Animals can eat”);
}
}
Class Dog extends Animal
{
public void eat ( )
{
System.out.println (“Dog can eat and drink”);
}
}
public class TestCat
{
public static void main (String args [ ])
{
Animal a = new Animal ( );    //Animal reference and object
Animal b= new Dog ( );        //Animal reference but Rat object
a.eat ( );          //runs the method in Animal class
b.eat ( );          //runs the method in Dog class
}
}
```

This will produce the given output:

Animals can eat

Dog can eat and drink

In the example given above, you can see that even though Dog is a type of Animal it runs the eat method in the Dog class. The reason for this is: In compile time the check is made on the reference type. However, in the runtime, JVM

figures out the object type and would run the method that belongs to that particular object.

Therefore, in the above example, the program will compile properly since Animal class has the method eat. Then at the runtime it runs the method specific for that object.

Rules for method overriding:

- "The return type should be the same or a subtype of the return type declared in the original overridden method in the super class.
- A method declared final cannot be overridden.
- The access level cannot be more restrictive than the overridden method's access level. For example: if the super class method is declared public then the overriding method in the sub class cannot be either private or public. However the access level can be less restrictive than the overridden method's access level.
- The argument list should be exactly the same as that of the overridden method.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited then it cannot be overridden.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overridden method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- Constructors cannot be overridden."

Using the super keyword

When invoking a superclass version of an overridden method the super keyword is used.

```
class Animal
{
public void move ( )
{System.out.println (“Animals can move”);}
}//Animal
Class cow extends Animal
{
public void move ( )
{
super.move ( );          //invokes the super class method
System.out.println (“Cow can walk and run”):
}
public class TestCow
{
public static void main (String args [ ])
{
Animal b = new Cow ( );       //Animal reference but Cow object
b.move ( );      //runs the method in Cow class
}
}
```

This would produce following result:

```
Animals can move
Cow can walk and run
```

Check your progress 2

1. Write the rules for method overriding.
2. What do you mean by the term polymorphism

.....

.....

.....

.....

.....

.....

.....

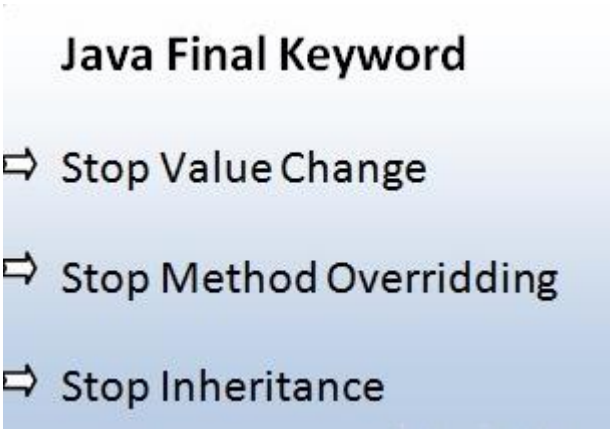
.....

.....

.....

.....

1.4 Final Keyword



The final keyword has mainly three uses:

1. Creating constants

2. Preventing method overriding
3. Preventing inheritance

Let us discuss these uses in order to understand the concept of final keyword in detail:

1. **Creating Constants**

Declaring a variable as final makes it constant, doing so prevents the contents from being modified. This means that you must initialize a final variable when it is declared.

For example:

finalint FILE_NEW=1

finalint FILE_OPEN=2

Subsequent parts of your program can now use FILE_OPEN, FILE_NEW etc. as if they were constants. It is a common naming convention to choose all uppercase identifiers for final variables. Thus, a final variable is essentially a constant.

2. **Preventing method overriding**

The keyword final can also be applied to methods but its meaning is substantially different than when it is applied to variables. Methods declared as final cannot be overridden, that is, a method in the superclass cannot be overridden in the subclass.

3. **Preventing inheritance**

In some cases, you may want to prevent a class form being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods a final, too. It is illegal to declare a class both abstract and final since an abstract class is incomplete by itself and relies upon its subclass to provide complete implementations.

Check your progress 3

1. List the uses of final keyword.
2. Write the syntax of final keyword

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

1.5 Let Us Sum Up

This Unit No.1 of this Block we have understood “Inheritance is one of the cornerstones of OOP because it allows for the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related objects”, that is, objects with common attributes and behaviors. The various needs of inheritance are 1.Closer to Real-World, 2. Code Reusability ad 3. Transitive Nature.

The study of Generalisation/Specialisation has made us understand Syntax, Create a superclass then Create a subclass by extending class A and also the subclass has access to all public members of its superclass. Then further studied that "Polymorphism allows one interface to be used for a set of actions i.e. one name may refer to different functionality. Polymorphism allows an object to accept different requests of a client (it then properly interprets the request like choosing appropriate method) and responds according to the current state of the runtime system", all without bothering the user. There are two types of polymorphism, 1. Compile-time polymorphism, 2. Runtime Polymorphism.

We understood Method Overriding, if a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final. Overriding means redefining a method in an inheritance hierarchy. 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. We have also understood in detail all Rules for method overriding. There is also good understanding about the final keyword has mainly three uses 1. Creating constants, 2.Preventing method overriding, and 3. Preventing inheritance

1.6 Suggested Answer for Check Your Progress

Check your progress 1

Answers: See Section 1.2

Check your progress 2

Answers: See Section 1.3

Check your progress 3

Answers: See Section 1.4

1.7 Glossary

3. **Inheritance -** Inheritance is one of the cornerstones of OOP because it allows for the creation of hierarchical classifications.
4. **Overriding -** Overriding means redefining a method in an inheritance hierarchy. 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.

1.8 Assignment

Explain how a subclass has access to all public members of its superclass.

Write a program to create a superclass.

1.9 Activities

Write a program to explain inheritance

1.10 Case Study

Create a class Medicine to represent a drug manufactured by a pharmaceutical company. Provide a function display Label() in this class to print Name and address of the company.

Derive Tablet, Syrup and Ointment classes from the Medicine class. Override the display Label() function in each of these classes to print additional information suitable to the type of medicine. For example, in case of tablets, it could be “store in a cool dry place”, in case of ointments it could be “for external use only” etc.

Create a class Test Medicine. Write main function to do the following:

1 Declare an array of Medicine references of size 10
2 Create a medicine object of the type as decided by a randomly generated integer in the range 1 to 3.
3 Refer Java API Documentation to find out random generation feature.
4 Check the polymorphic behavior of the display Label() method.

1.11 Further Reading

1. Core Java 2, 2 volumes, Cay S. Horstmann, Gary Cornell, The Sun Microsystems Press, 1999, Indian reprint 2000
2. Java 2, The Complete Reference, Patrick Naughton and Herbert Schildt, Tata McGraw Hill, 1999
3. Programming with Java, Ed. 2,E. Balagurusamy, Tata McGraw Hill, 1998, reprint,2000
4. The Java Tutorial, Ed. 2, 2 volumes,MaryCampione and Kathy Walrath, Addison Wesley Longmans, 1998
5. The Java Language Specification, Ed. 2, James Gosling, Bill Joy, Guy Steele &GiladBracha, (Ed. 1 1996, Ed. 2 2000), Sun Microsystems, 2000
6. Using Java 2, Joseph L. Weber, Prentice Hall, Eastern Economy Edition, 2000