



ሐምሌ, 2013 ዓ.ም

Object Oriented Programming



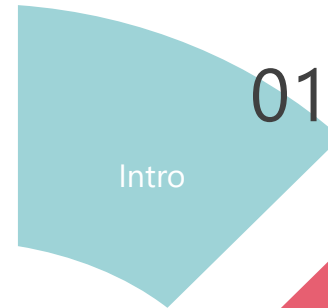
Chapter Four

Wollo University, Kombolcha Institute of Technology.
College of Informatics.
Department of Information System.

By Daniel G.

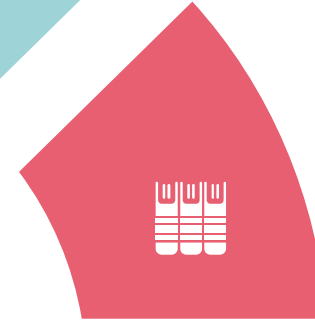
Contents

Object Oriented Programming
chapter 4 will include this topics.



01

Concept of Polymorphism
Type of polymorphism



02

Relationships among
objects in an inheritance



03

Assigning reference of
subclass to superclass-type
variable



04

Assigning a superclass reference
to subclass-type variable



05

Multiple inheritance
and interfaces

06

Subclass method calls via
superclass-type variable

Basics questions In this chapter

1. *What is Polymorphism ?*
2. *What are the type of polymorphism?*
3. *What is abstract class and methods?*
4. *What is interface in java ?*
5. *Multiple inheritance in OOP with interface ?*



What is Polymorphism ?

- ❖ **Polymorphism** is a feature of object-oriented programming languages that allows a specific routine to use variables of different types at different times
- ❖ **Polymorphism** is the ability of a programming language to present the same interface for several different underlying data types
- ❖ **Polymorphism** is the ability of different objects to respond in a unique way to the same message

- ❖ **Polymorphism** is one of the **OOPs** feature that allows us to perform a **single action** in different ways.
- ❖ **For example**, lets say we have a class **Animal** that has a method **sound()**. Since this is a generic class so we can't give it a implementation like: **Roar**, **Meow**, **Oink** etc. We had to give a generic message.

Conti...

```
public class Animal{
    public void sound() {
        System.out.println("Animal is making a sound");
    }
}
```

Now lets say we **two** subclasses of **Animal** class: **Horse** and **Cat** that extends **Animal** class. We can provide the implementation to the same method like.....

Conti...

```
class Horse extends Animal{
    @Override
    public void sound() {
        System.out.println("Neigh");
    }
}

public static void main(String args[]) {
    Animal obj = new Horse();
    obj.sound();
}
}
```

Conti...

```
class Cat extends Animal{
    @Override
    public void sound() {
        System.out.println("Meow");
    }
}

public static void main(String args[]) {
    Animal obj = new Cat();
    obj.sound();
}
}
```


❖ There are two basic concepts of *Polymorphism*

1. Static Polymorphism
2. Dynamic Polymorphism

Static Polymorphism

- ❖ In Java, **static** polymorphism is achieved through **method overloading**.
- ❖ **Method overloading** means there are several methods present in a class having the **same** name but different types/order/number of parameters.
- ❖ At compile time, Java knows which method to **invoke** by checking the method signatures. So, this is called **compile time polymorphism** or **static binding**.

Example:

```
class MOverload{  
    public int add(int x, int y){  
        //method 1  
        return x+y;  
    }  
    public int add(int x, int y, int z){  
        //method 2  
        return x+y+z;  
    }  
}
```

```
public int add(double x, int y){  
    //method 3  
    return (int)x+y;  
}
```

```
public int add(int x, double y){  
    //method 4  
    return x+(int)y;  
}
```

```
class Test{  
    public static void main(String[] args){  
        DemoOverload demo=new DemoOverload();  
        System.out.println(demo.add(2,3)); //method 1 called  
        System.out.println(demo.add(2,3,4)); //method 2 called  
        System.out.println(demo.add(2,3.4)); //method 4 called  
        System.out.println(demo.add(2.5,3)); //method 3 called }  
    }  
}
```

Dynamic Polymorphism

- ❖ Suppose a **subclass** overrides a particular method of the **superclass**.
- ❖ Let's say we create an object of the **subclass** and assign it to the **superclass** reference. Now, if we call the **overridden** method on the superclass reference then the subclass version of the method will be called.

Example:

```
class Vehicle{
    public void move(){
        System.out.println("Vehicles can move!!");
    }
}

class MotorBike extends Vehicle{
    public void move(){
        System.out.println("MotorBike can move and accelerate too!!");
    }
}
```

```
class Test{  
    public static void main(String[] args){  
        Vehicle vh=new MotorBike();  
        vh.move();// prints MotorBike can move and accelerate too!!  
        vh=new Vehicle();  
        vh.move();    // prints Vehicles can move!!  
    }  
}
```


- ❖ The **relationship** between **objects** defines how these objects will interact or collaborate to perform an operation in an application.
- ❖ **Relationships** among **objects** in an **inheritance** hierarchy In public inheritance an object of a derived class can **be treated** as an object of its base class. Each derived class object is an object of its base class

Example:

```
class Person {  
}  
  
class Wucommunity extends person{  
}  
  
clas Student extends Wucommunity{  
}  
  
class KiotStudent extends Student{  
}
```

Example

ሐምሌ, 2013 ዓ.ም

```
public class test{  
    person p = new person ();  
    person wu = new Wucommunity ();  
    person s = new Student ();  
    person ks = new KiotStudent ();  
    Student ks1 = new KiotStudent ();  
}
```

Assigning reference of subclass to superclass-type variable

ሐምሌ, 2013 ዓ.ም

```
Student S = new Student ();
```

```
KiotStudent k = new KiotStudent ();
```

```
K=(Student) S
```

We can access a super class methods via 'k'

Assigning a super class reference to subclass-type variable

ሐምሌ, 2013 ዓ.ም

```
Student S = new Student ();
```

```
Student k = new KiotStudent ();
```

```
S=(KiotStudent ) k;
```

❖ We can access a sub class methods via 'S'

Subclass method calls via superclass-type variable

ሐምሌ, 2013 ዓ.ም

```
Student k = new KiotStudent ();
```

❖ We can access a sub class methods via 'k'

- ❖ Data **abstraction** is the process of **hiding** certain details and showing only essential information to the user.
- ❖ **Abstraction** can be achieved with either **abstract** classes or **interfaces**
- ❖ **Abstract class**: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- ❖ It represents any entity in the real world which is generic in nature
- ❖ **Example** : **Vehicle, fruit, shapes**

- ❖ Syntax < **abstract** **class** <class name> >
- ❖ Abstract class contains **abstract** methods
- ❖ Abstract class can not be instantiated and can't create an object
- ❖ If a **class** contain an abstract method then the class must be defined as **abstract**
- ❖ **Abstract** class may or may not contain **abstract** method

- ❖ **Abstract** class can be sub classed by other **concrete** class to be implemented
- ❖ A concrete class which implements an interface and abstract class must implement all abstract methods of the interface and abstract class
- ❖ A **concrete** class is a class can't be defined as **abstract** and it is important for **implementing** interface and abstract class.

- ❖ **Abstract method**: can only be used in an **abstract** class, and it does not have a **body** or **implementation**. The body is provided by the subclass (inherited from).
- ❖ It ends with **semicolon (;)**
- ❖ Syntax
 - ✓ `abstract <data type> method name(parameter)`

Example 1:

```
abstract class Animal {  
    public abstract void animalSound();  
    public void sleep() {  
        System.out.println("Zzz");  
    }  
}
```

```
Animal myObj = new Animal(); // will generate an error
```

Conti...

ሐምሌ, 2013 ዓ.ም

```
Example 2: abstract class Animal {  
    public abstract void animalSound();  
    public void sleep() {  
        System.out.println("Zzz");  
    }  
}
```

```
class Pig extends Animal {  
    public void animalSound() {  
        System.out.println("The pig says: wee wee");  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        Pig myPig = new Pig(); // Create a Pig object  
        myPig.animalSound();  
        myPig.sleep();  
    }  
}
```

Multiple inheritance and interfaces

ሐምሌ, 2013 ዓ.ም

- ❖ An **interface** contains variables and methods like a class but the methods in an interface are **abstract** by default **unlike** a class.
- ❖ **Multiple inheritance** by interface occurs if a class implements multiple interfaces or also if an interface itself extends multiple interfaces.
- ❖ A **program** that demonstrates multiple inheritance by **interface** in Java is given as follows:

Conti...

ከምሌ, 2013 ዓ.ም

```
interface AnimalEat {  
    void eat();  
}  
  
interface AnimalTravel {  
    void travel();  
}  
  
class Animal implements AnimalEat, AnimalTravel {  
    public void eat() {  
        System.out.println("Animal is eating");  
    }  
    public void travel() {  
        System.out.println("Animal is travelling");  
    }  
}
```

```
public class Demo {  
    public static void main(String args[]) {  
        Animal a = new Animal();  
        a.eat();  
        a.travel();  
    }  
}
```

❖ We **can't** create an object from an **interface**

Demonstration about concrete ,abstract and interface

ሐምሌ, 2013 ዓ.ም

Concrete class	Abstract Class	Interface
<p>Contains</p> <ul style="list-style-type: none">➤ Fields➤ Constructors➤ Implemented methods➤ Instantiable➤ Real world entities	<p>Contains</p> <ul style="list-style-type: none">➤ Fields➤ Un implemented methods➤ Constructor➤ Implemented methods➤ Not Instantiable➤ Represents generic entities	<p>Contains</p> <ul style="list-style-type: none">➤ Constant Fields➤ Only abstract methods➤ Not Instantiable➤ Can't represent real world entities

ሐምሌ, 2013 ዓ.ም

THANK YOU

Any questions?
Feel free !