CCS0023/L

Object Oriented Programming (Java)

Polymorphism and Abstraction

07

LESSON #7

Polymorphism and Abstraction

Polymorphism

- **Polymorphism is the ability of objects belonging to different types to respond to methods of the same name, each one according to the right type-specific behavior.**

- **It is the ability to redefine methods for derived classes**.

Polymorphism

Implementing Polymorphism

1. **Method Overloading**

   Using one method identifier to refer to multiple functions in the same class, In the Java programming language, methods can be overloaded but not variables or operators.

Polymorphism

Method Overloading

- **Constructor Overloading**
  - creating more than one constructor in a class

- **Method Overloading**
  - creating multiple methods having same name in one class.

Polymorphism

Example : Constructor Overloading

```
public Student(){
            …
}

public Student(String name, int studNo){
            this.name = “Anonymous”;
}
        //more constructor here…
```

- Polymorphism and Abstraction

Polymorphism

Example : Method Overloading

```
public void eat(){
            …
}

public void eat(String food){
       System.out.println("The animal is eating " +food);
}
       //more methods here…
```

- Polymorphism and Abstraction

Polymorphism

Implementing Polymorphism

2. **Method Overriding**
   Providing a different implementation of a method in a subclass of the class that originally defined a method.

Polymorphism

Example : Method Overriding

```
public class ElectronicDevice{
        …
public void on(){
        System.out.println("The device is turned on!");
        }
}

public class Computer extends ElectronicDevice{
public void on(){
        System.out.println("The computer boots…");
        System.out.println("The computer loads drivers…");
        System.out.println("Welcome to Windows XP!");
}
}
```

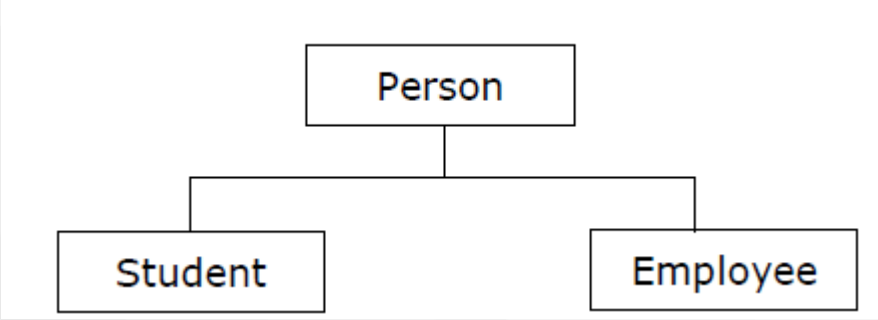
Polymorphism

Overloading VS. Overriding

Overloading	Overriding
Overloaded functions supplement each other.	Overriding function replaces the function it overrides.
Overloaded functions can exist, in any number, in the same class.	Each function in a base class can be overridden at most once in any one derived class.
Overloaded functions must have different argument lists.	Overriding functions must have argument lists of identical type and order.
The return type of an overloaded function may be chosen freely.	The return type of an overriding method must be identical to the function it overrides.

Polymorphism

Now, given the parent class Person and the subclass Student of our previous example, we add another subclass of Person which is Employee. Below is the class hierarchy for that,



Polymorphism

In Java, we can create a reference that is of type superclass to an object of its subclass.

For example,

```
public static main( String[] args )
{
        Person ref;
        Student studentObject = new Student();
        Employee employeeObject = new Employee();
        ref = studentObject; //Person ref points to a
        // Student object
        //some code here
}
```

Polymorphism

Now suppose we have a **getName** method in our **superclass** Person, and we override this method in both the subclasses Student and Employee,

```
public class Person
{
        public String getName(){
        System.out.println(“Person Name:” + name);
        return name;
        }
}
```

Polymorphism

Now suppose we have a **getName** method in our **superclass** Person, and we override this method in both the subclasses Student and Employee,

```
public class Student extends Person
{
        public String getName(){
        System.out.println("Student Name:" + name);
        return name;
        }
}
```

Polymorphism

Now suppose we have a **getName** method in our **superclass** Person, and we override this method in both the subclasses Student and Employee,

```
public class Employee extends Person
{
        public String getName(){
        System.out.println("Employee Name:" + name);
        return name;
        }
}
```

Polymorphism

Going back to our main method, when we try to call the getName method of the

reference Person ref, the getName method of the Student object will be called.

```
public static main( String[] args )
{
        Person ref;
        Student studentObject = new Student();
        Employee employeeObject = new Employee();
        ref = studentObject; //Person reference points to a
        // Student object
        String temp = ref.getName(); //getName of Student
        //class is called
        System.out.println( temp );
}
```

Polymorphism

Now, if we assign ref to an Employee object, the getName method of Employee will be called.

```
public static main( String[] args )
    {
        Person ref;
        Student studentObject = new Student();
        Employee employeeObject = new Employee();
        ref = employeeObject; //Person reference points to an
        // Employee object
        String temp = ref.getName(); //getName of Employee
        //class is called
        System.out.println( temp );
        }
```

- Polymorphism and Abstraction

Polymorphism

This ability of our reference to change behavior according to what object it is holding is called **polymorphism**. **Polymorphism** allows multiple objects of different subclasses to be treated as objects of a single superclass, while automatically selecting the proper methods to apply to a particular object based on the subclass it belongs to.

Polymorphism

Another example that exhibits the property of polymorphism is when we try to pass a reference to methods.

Suppose we have a static method **printInformation** that takes in a Person object as reference, we can actually pass a reference of type Employee and type Student to this method as long as it is a subclass of the class Person.

Polymorphism

```
public static main( String[] args )
{
    Student studentObject = new Student();
    Employee employeeObject = new Employee();
    printInformation( studentObject );
    printInformation( employeeObject );
}
    public static printInformation( Person p ){
    . . . .
    }
```

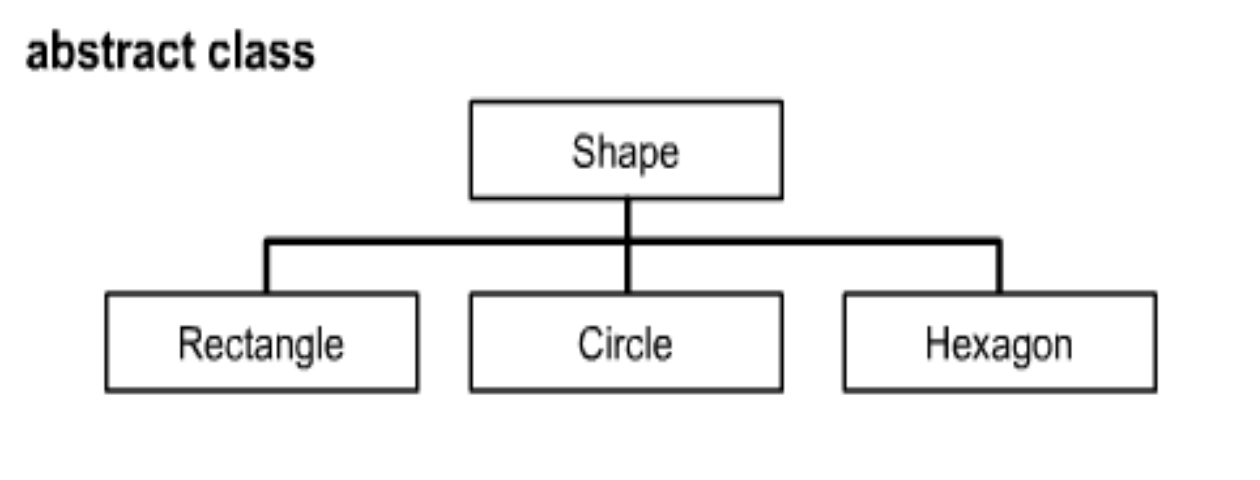
- Polymorphism and Abstraction

Abstract Classes

The Abstract Class and Interface:

➢**Abstract Class** – contains one or more abstract methods and, therefore, can never be instantiated. It is defined so that other classes can extend them and make them concrete by implementing the abstract methods.

Abstract Classes



- Polymorphism and Abstraction

Abstract Classes

Syntax for Declaring Abstract Class:

```
abstract class ClassName    {
    // abstract method declarations
    public abstract returnType methodName(ArgsList);
}
```

Abstract Classes

Take Note:

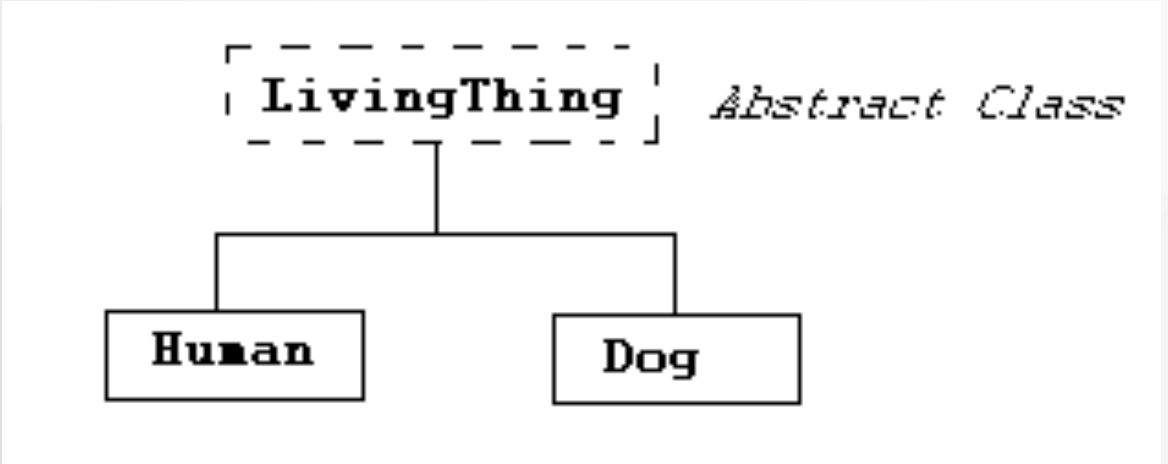
All abstract classes are public by default and cannot be instantiated. Constructors and public methods cannot be declared as abstract.

Abstract Classes

For example, we want to create a superclass named **LivingThing**. This class has certain methods like **breath**, **eat**, **sleep** and **walk**. However, there are some methods in this superclass wherein we cannot generalize the behavior. Take for example, the **walk** method. Not all living things walk the same way. Take the humans for instance, we humans walk on two legs, while other living things like dogs walk on four legs. However, there are many characteristics that living things have in common, that is why we want to create a general superclass for this.

Abstract Classes

In order to do this, we can create a superclass that has some methods with
implementations and others which do not. This kind of class is called an **abstract class**.



Abstract Classes

An **abstract class** is a class that cannot be instantiated. It often appears at the top of an object-oriented programming class hierarchy, defining the broad types of actions possible with objects of all subclasses of the class.

Abstract Classes

Those methods in the abstract classes that do not have implementation are called
abstract methods.

To create an abstract method, just write the method declaration without the body and use the abstract keyword. For example,

```
public abstract void someMethod();
```

Abstract Classes

Now, let's create an example abstract class.

```
public abstract class LivingThing
{
        public void breath(){
                System.out.println("Living Thing breathing...");
        }
        public void eat(){
                System.out.println("Living Thing eating...");
        }
        /**
        * abstract method walk
        * We want this method to be overridden by subclasses of
        * LivingThing*/
        public abstract void walk();
}
```

Abstract Classes

When a class extends the LivingThing abstract class, it is required to override the
abstract method walk(), or else, that subclass will also become an abstract class, and
therefore cannot be instantiated. For example,

```
public class Human extends LivingThing
{
    public void walk(){
    System.out.println("Human walks...");
    }
}
```

Abstract Classes

If the class Human does not override the walk method, we would encounter the following error message,

```
Human.java:1: Human is not abstract and does not override
abstract method walk() in LivingThing
public class Human extends LivingThing
       ^
1 error
```

Abstract Classes

Coding Guidelines:

Use abstract classes to define broad types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

Interfaces

An **interface** is a special kind of block containing method signatures (and possibly constants) only. **Interfaces** define the signatures of a set of methods without the body.

Interfaces define a standard and public way of specifying the behavior of classes. They allow classes, regardless of their location in the class hierarchy, to implement common behaviors.

Note that interfaces exhibit polymorphism as well, since program may call an interface method and the proper version of that method will be executed depending on the type of object passed to the interface method call.

Interface

Interface – an abstract class that represents a collection of method definitions and constant values. It can later be implemented by classes that define the interface using the implements keyword.

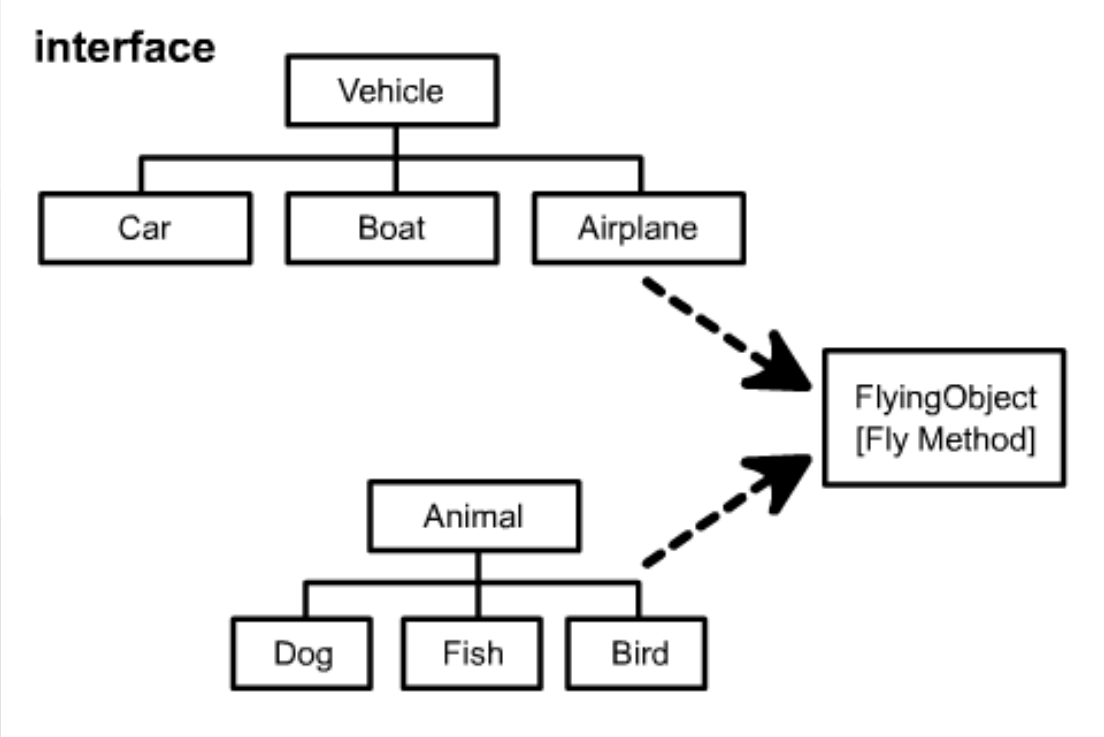
Interfaces

Why do we use Interfaces?

We need to use interfaces if we want unrelated classes to implement similar methods.

Thru interfaces, we can actually capture similarities among unrelated classes without artificially forcing a class relationship.

Interface



- Polymorphism and Abstraction

Interface

Let's take as an example a class **Line** which contains methods that computes the length of the line and compares a **Line** object to objects of the same class.

Now, suppose we have another class **MyInteger** which contains methods that compares a **MyInteger** object to objects of the same class.

As we can see here, both of the classes have some similar methods which compares them from other objects of the same type, but they are not related whatsoever.

In order to enforce a way to make sure that these two classes implement some methods with similar signatures, we can use an interface for this.

Interface

We can create an interface class, let's say interface **Relation** which has some comparison method declarations. Our interface Relation can be declared as,

```
public interface Relation
{
	public boolean isGreater( Object a, Object b);
	public boolean isLess( Object a, Object b);
	public boolean isEqual( Object a, Object b);
}
```

Interface

Another reason for using an object's programming interface is to reveal an object‘s programming interface without revealing its class.

As we can see later on the section Interface vs. Classes, we can actually use an interface as data type.

Finally, we need to use interfaces to model multiple inheritance which allows a class to have more than one superclass.

Multiple inheritance is not present in Java, but present in other object-oriented languages like C++.

Interface vs. Abstract Class

The following are the main differences between an interface and an abstract class:

interface methods have no body,
an interface can only define constants and
an interface have no direct inherited relationship with any particular class, they are defined independently.

Interface vs. Class

One common characteristic of an interface and class is that they are both types. This means that an interface can be used in places where a class can be used.

For example, given a class **Person** and an interface **PersonInterface**, the following declarations are
valid:

```
PersonInterface pi = new Person();
Person pc = new Person();
```

However, you cannot create an instance from an interface. An example of this is:

```
PersonInterface pi = new PersonInterface(); //COMPILE
//ERROR!!!
```

Interface vs. Class

Another common characteristic is that both interface and class can define methods.

However, an interface does not have an implementation code while the class have one.

Creating Interfaces

To create an interface, we write,

```
public interface [InterfaceName]
{
    //some methods without the body
}
```

As an example, let's create an interface that defines relationships between two objects according to the “natural order” of the objects.

```
public interface Relation
{
    public boolean isGreater( Object a, Object b);
    public boolean isLess( Object a, Object b);
    public boolean isEqual( Object a, Object b);
}
```

Creating Interfaces

Now, to use the interface, we use the **implements** keyword. For example,

```
/**
* This class defines a line segment
*/
public class Line implements Relation
{
        private double x1;
        private double x2;
        private double y1;
        private double y2;
        public Line(double x1, double x2, double y1, double y2){
                this.x1 = x1;
                this.x2 = x2;
                this.y1 = y1;
                this.y2 = y2;
        }
```

Creating Interfaces

```
public double getLength(){
        double length = Math.sqrt((x2-x1)*(x2-x1) +
        (y2-y1)* (y2-y1));
        return length;
}

public boolean isGreater( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen > bLen);
}
```

- Polymorphism and Abstraction

Creating Interfaces

```
        public boolean isLess( Object a, Object b){
            double aLen = ((Line)a).getLength();
            double bLen = ((Line)b).getLength();
            return (aLen < bLen);
        }

        public boolean isEqual( Object a, Object b){
            double aLen = ((Line)a).getLength();
            double bLen = ((Line)b).getLength();
            return (aLen == bLen);
        }
}
```

Creating Interfaces

When your class tries to implement an interface, always make sure that you implement all the methods of that interface, or else, you would encounter this error,

```
Line.java:4: Line is not abstract and does not override
abstract method isGreater(java.lang.Object,java.lang.Object) in
Relation
public class Line implements Relation
       ^
1 error
```

Coding Guidelines:
Use interfaces to create the same standard method definitions in may different classes.
Once a set of standard method definition is created, you can write a single method to manipulate all of the classes that implement the interface.

Relationship of an Interface to a Class

As we have seen in the previous section, a class can implement an interface as long as it provides the implementation code for all the methods defined in the interface.

Another thing to note about the relationship of interfaces to classes is that, a class can only **EXTEND ONE** super class, but it can **IMPLEMENT MANY** interfaces. An example of a
class that implements many interfaces is,

```
public class Person implements
PersonInterface,LivingThing,WhateverInterface {
    //some code here
}
```

Relationship of an Interface to a Class

Another example of a class that extends one super class and implements an interface is,

```
public class ComputerScienceStudent extends Student
implements PersonInterface,LivingThing {
        //some code here
}
```

Take note that an interface is not part of the class inheritance hierarchy. Unrelated classes can implement the same interface

Inheritance among Interfaces

Interfaces are not part of the class hierarchy.
However, interfaces can have inheritance relationship among themselves.

For example, suppose we have two interfaces **StudentInterface** and **PersonInterface**.

If StudentInterface extends PersonInterface,
it will inherit all of the method declarations in PersonInterface.

```
public interface PersonInterface {
. . .
}
public interface StudentInterface extends PersonInterface {
. . .
}
```

THANK YOU