CHAPTER 8
Polymorphism
(Abstract Class and Interface)

Shin-Jie Lee (李信杰)
Assistant Professor
Computer and Network Center
Department of Computer Science and Information Engineering
National Cheng Kung University

Introduction to Polymorphism

- There are three main programming mechanisms that constitute object-oriented programming (OOP)
  - Encapsulation
  - Inheritance
  - Polymorphism

Introduction to Polymorphism

- **Polymorphism**: A same operation can behave differently (be implemented by different methods).

Late Binding

- ***Early binding*** or ***static binding***
  - **which method is to be called** is decided at compile-time
    - ***Overloading***: *an invocation can be operated on arguments of more than one type*

- ***Late binding*** or ***dynamic binding***
  - **which method is to be called** is decided at compile-time
    - ***Overriding***: a derived class inherits methods from the base class, it can change or override an inherited method

Lab: Early binding (through overloading)

```
public class SayHello {

  public String sayHello(String name){
    return "Hello! "+ name;
  }

  public String sayHello(String name, String gender){
    if(gender.equals("boy")){
      return "Hello! Mr. "+ name;
    }
    else if(gender.equals("girl")){
      return "Hello! Miss. "+ name;
    }else{
      return "Hello! "+ name;
    }
  }

  public static void main(String[] args){
    SayHello hello = new SayHello();
    System.out.println(hello.sayHello("S.J.")); //decided at compile time
    System.out.println(hello.sayHello("S.J.","boy")); //decided at compile time
  }
}
```

Lab: Late binding (through overriding)

```
public class Payment {
  public void pay(){
    System.out.println("Pay in cash");
  }
  public void checkout(){
    pay();
  }
}
```

```
public class Store {
  public static void main(String[] args) {
    Payment p1 = new Payment();
    p1.checkout();
  }
}
```

Lab: Late binding (through overriding)

```
public class CreditCardPayment extends Payment{
  public void pay() {
    System.out.println("Pay with credit card");
  }
}
```

```
public class Store {
  public static void main(String[] args) {
    Payment p1 = new Payment();
    p1.checkout();

    Payment p2 = new CreditCardPayment();
    p2.checkout();
  }
}
```

Pitfall: No Late Binding for Static Methods

- Java uses **static binding** with **private**, **`final`**, and **static** methods
  - In the case of **`private`** and **`final`** methods, late binding would serve no purpose
  - However, in the case of a static method invoked using a calling object, it does make a difference

Lab

```
public class Payment {
  public static void pay(){
    System.out.println("Pay in cash");
  }
  public void checkout(){
    pay();
  }
}
```

```
public class CreditCardPayment extends Payment{
  public static void pay() {
    System.out.println("Pay with credit card");
  }
}
```

Then run Store again!

Lab

```
public class Store {
  public static void main(String[] args) {
    Payment p1 = new Payment();
    p1.checkout();

    Payment p2 = new CreditCardPayment();
    p2.checkout();
  }
}
```

the type of **p2** is determined by its variable name, not the object that it references

Upcasting and Downcasting

- *Upcasting* is when an object of a derived class is assigned to a variable of a base class (or any ancestor class)

```
Payment p2 = new CreditCardPayment();
p2.checkout();
```

Upcasting and Downcasting

- *Downcasting* is when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class)
  - Downcasting has to be done very carefully
  - In many cases it doesn't make sense, or is illegal:

```
Payment p1 = new Payment();
CreditCardPayment p2 = (CreditCardPayment)p1; //runtime error
```

Tip: Checking to See if Downcasting is Legitimate

- Downcasting to a specific type is only sensible if the object being cast is an instance of that type
  - This is exactly what the **`instanceof`** operator tests for:

    `object` **`instanceof`** ***`ClassName`***
  - It will return true if ***`object`*** is of type ***`ClassName`***
  - In particular, it will return true if ***`object`*** is an instance of any descendent class of ***`ClassName`***

Lab (Downcasting)

Step1: Remove “static” in **CreditCardPayment and Payment**

Step2

```
public class CreditCardPayment extends Payment{
  public void pay() {
    System.out.println("Pay with credit card");
  }
  public void sign(){
    System.out.println("Signing...");
  }
}
```

Lab (Downcasting)

```
public class Store {

public static void main(String[] args) {
  Payment p1 = new Payment();
  p1.checkout();
  payProcess(p1);

  Payment p2 = new CreditCardPayment();
  p2.checkout();
  payProcess(p2);
}

public static void payProcess(Payment p){
  if(p instanceof CreditCardPayment){
    ((CreditCardPayment)p).sign();
  }
}

}
```

A First Look at the `clone` Method

- Creates and returns a copy of this object.
  - x.clone() != x

- The heading for the **`clone`** method defined in the **`Object`** class is as follows:

```
protected Object clone()
```

- A change to a more permissive access, such as from protected to public, is always allowed when overriding a method definition

Lab

```
public class A implements Cloneable{
  int num = 1;
  B b = new B();

  public Object clone(){
    try{
      return super.clone();
    }catch(Exception e){
      return null;
    }
  }
}


public class B implements Cloneable{
  int speed = 100;
}
```

Lab (Shallow Copy)

```
public class Test {

  public static void main(String[] args) {
    A a = new A();
    System.out.println(a.num);              //1
    System.out.println(a.b.speed);          //100

    A clone_a = (A) a.clone();
    System.out.println(clone_a.num);        //1
    System.out.println(clone_a.b.speed);    //100

    clone_a.num = 2;
    clone_a.b.speed = 200;
    System.out.println(a.num);              //1
    System.out.println(a.b.speed);          //200

  }

}
```

A First Look at the `clone` Method

- If a class has a copy constructor, the **`clone`** method for that class can use the *copy constructor* to create the copy returned by the **`clone`** method

```
public Sale clone()
{
  return new Sale(this);
}
```

Lab (Deep Copy)

```
public class A implements Cloneable{
  int num = 1;
  B b = new B();

  public A(A a){
    num = a.num;
    b.speed = a.b.speed;
  }
  public A(){}

  public Object clone(){
    return new A(this);
  }
}
```

Lab (Deep Copy)

```
public class Test {

  public static void main(String[] args) {
    A a = new A();
    System.out.println(a.num);              //1
    System.out.println(a.b.speed);          //100

    A clone_a = (A) a.clone();
    System.out.println(clone_a.num);        //1
    System.out.println(clone_a.b.speed);    //100

    clone_a.num = 2;
    clone_a.b.speed = 200;
    System.out.println(a.num);              //1
    System.out.println(a.b.speed);          //100

  }

}
```

Introduction to Abstract Classes

- In order to postpone the definition of a method, Java allows an *abstract method* to be declared
  - An abstract method has a heading, but no method body
  - The body of the method is defined in the derived classes

- The class that contains an abstract method is called an *abstract class*

Abstract Method

- An abstract method is like a placeholder for a method that will be fully defined in a descendent class
- It has a complete method heading, to which has been added the modifier **`abstract`**
- **It cannot be private**
- It has no method body, and ends with a semicolon in place of its body

```
public abstract double getPay();
public abstract void doIt(int count);
```

Abstract Class

- A class that has at least one abstract method is called an *abstract class*
  - An abstract class must have the modifier **`abstract`** included in its class heading:

```
public abstract class Employee
{
  private instanceVariables;
  . . .
  public abstract double getPay();
  . . .
}
```

Abstract Class

- An abstract class can have any number of abstract and/or fully defined methods
- If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add `abstract` to its modifier

- A class that has no abstract methods is called a *concrete class*

Pitfall: You Cannot Create Instances of an Abstract Class

- An abstract class can only be used to derive more specialized classes
  - While it may be useful to discuss employees in general, in reality an employee must be a salaried worker or an hourly worker
- **An abstract class constructor cannot be used to create an object of the abstract class**

Lab

```
public abstract class Animal {
  public abstract void run();
  public void sit(){ System.out.println(“Sit down…”); }
}
```

```
public class Dog extends Animal {
  public void run(){
    System.out.println("The dog is running");
  }
}
```

```
public class Cat extends Animal{
  public void run(){
    System.out.println("The cat is running");
  }
}
```

Lab

```
public class House {

  public static void main(String[] args) {
    Animal dog = new Dog();
    Animal cat = new Cat();

    playWith(dog);
    playWith(cat);

    dog.sit();
    cat.sit();
  }

  public static void playWith(Animal animal){
    animal.run();

  }

}
```

Lab

_____ binding refers to the method definition being associated with the method invocation when the code is compiled.

(a)Dynamic

(b)Late

(c)Early

(d)None of the above

Lab

Java does not use late binding for methods marked as:

(a)final

(b)static

(c)private

(d)all of the above

Lab

Assigning an object of a derived class to a variable of a base class is called:

(a)static binding

(b)dynamic binding 

(c)Upcasting

(d)downcasting

Lab

Assigning an object of an ancestor class to a descendent class is called:

(a)static binding

(b)dynamic binding

(c)Upcasting

(d)downcasting

Lab

If you choose to use the method clone in your code, you must _____ the clone method.

(a)overload

(b)encapsulate

(c)override

(d)protect

Lab

You cannot create an object using a/an:

(a)superclass constructor

(b)subclass constructor

(c)ancestor class constructor

(d)abstract class constructor

Lab

An abstract method cannot be modified by:

(a)public

(b)protected

(c)private

(d)none of the above

Lab

A class that has at least one abstract method is called an:

(a)concrete class

(b)encapsulated class

(c)abstract class

(d)private class

Lab

A class with no abstract methods is called a

(a)concrete class

(b)encapsulated class

(c)abstract class

(d)private class

Interfaces

- An *interface* is something like an extreme case of an abstract class
  - However, *an interface is not a class*
  - *It is a type that can be satisfied by any class that implements the interface*
- The syntax for defining an interface is similar to that of defining a class
  - Except the word **`interface`** is used in place of **`class`**

Interfaces

- An interface specifies a set of methods that any class that implements the interface must have
  - It contains **method headings** and **constant definitions** only
    - Any variables defined in an interface must be public, static, and final
  - It contains **no instance variables nor any complete method definitions**

Lab (Constants)

```
public interface Shape {
  int color = 1; // => public static final int color = 1;
}
```

```
public class Paint {
  public static void main(String[] args) {
    System.out.println(Shape.color);
  }
}
```

Interfaces

- All methods in an interface are **implicitly public and abstract**, so you can omit the public modifier.
  - They cannot be given private or protected

```
public interface ISpec1 {
        private void run(); //Not allowed
        protected void run(); //Not allowed

        void run(); // Allowed. Equal to the following definition
        public abstract void run(); //Allowed

}
```

Lab

```
public interface Shape {

  int color = 1; // => public static final int color = 1;

  public abstract double area(); //=> double area();
}
```

Interfaces

- *Multiple inheritance* is not allowed in Java
- Instead, Java's way of approximating multiple inheritance is through interfaces

```
public class ConcreteClass implements ISpec1, ISpec2, ISpec3{
        …
}
```

Interfaces

- To *implement an interface*, a concrete class must do two things:

1. `implements` ***`Interface_Name`***

2. The class must implement *all* the method headings listed in the definition(s) of the interface(s)

Lab

```
public class Rectangle implements Shape{
  int x1=0;
  int y1=0;
  int x2=10;
  int y2=10;
  public double area(){
    return (x2-x1)*(y2-y1);
  }
}


public class Circle implements Shape{

  double radius = 3;
  public double area(){
    return radius*radius*3.14;
  }
}
```

Lab

```
public class Paint {

  public static void main(String[] args) {
    System.out.println(Shape.color);

    Shape shape1 = new Rectangle();
    printArea(shape1);

    Shape shape2 = new Circle();
    printArea(shape2);
  }

  public static void printArea(Shape shape){
    System.out.println(shape.area());
  }
}
```

Abstract Classes Implementing Interfaces

- Abstract classes may implement one or more interfaces
  - Any method headings given in the interface that are not given definitions are made into abstract methods
- A concrete class must give definitions for all the method headings given in the abstract class *and the interface*

Abstract Class vs. Interface

```
public abstract class Animal {
  public abstract void run();
  public void sit(){ System.out.println("Sit down…"); }
}
```

vs.

```
public interface Shape {

  int color = 1; // => public static final int color = 1;

  public abstract double area(); //=> double area();
}
```

Derived Interfaces

- Like classes, an interface may be derived from a base interface
  - This is called *extending* the interface
  - The derived interface must include the phrase

    ```
    extends BaseInterfaceName
    ```
- A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface

Lab

```
public interface Drawing {

   public abstract void drawBorder();
}
```

```
public interface Shape extends Drawing{

  int color = 1; // => public static final int color = 1;

  public abstract double area();
}
```

Lab

```
public class Rectangle implements Shape{

  int x1=0;
  int y1=0;
  int x2=10;
  int y2=10;

  public double area(){
    return (x2-x1)*(y2-y1);
  }
  public void drawBorder(){
    System.out.println("Drawing the border of the rectangle...");
  }
}
```

Lab

```
public class Circle implements Shape{

  double radius = 3;
  public double area(){
    return radius*radius*3.14;
  }

  public void drawBorder(){
    System.out.println("Drawing the border of the circle...");
  }
}
```

Lab

A class that uses an interface must use the keyword:

(a)Extends

(b)Inherits

(c)Super

(d)Implements

Lab

An interface and all of its method headings are normally declared to be:

(a)public

(b)private

(c)Protected

(d)package access

Lab

An interface may contain:

(a)instance variables

(b)primitive variables

(c)constant variables

(d)all of the above

Lab

```
public interface Printable {
  void printAll();
}
```

```
class Person implements Printable {
  private String name = new String("Bill");
  private int age = 22;

  public void printAll() {
    System.out.println("Name is " + name + ", age is " + age);
  }
}
```

```
public class PrintableTest {
  public static void main(String[] args) {
    Printable p = new Person();
    p.printAll();
  }
}
```

Reference

- "Absolute Java". Walter Savitch and Kenrick Mock. Addison-Wesley; 5 edition. 2012
- "Java How to Program". Paul Deitel and Harvey Deitel. Prentice Hall; 9 edition. 2011.
- "A Programmers Guide To Java SCJP Certification: A Comprehensive Primer 3rd Edition". Khalid Mughal, Rolf Rasmussen. Addison-Wesley Professional. 2008