CS102

Introduction to data structures, algorithms, and object-oriented programming

March 7, 2016
• Java only allows *single* inheritance, so a child class can only have one parent, or one subclass can have only one superclass.

• Interfaces can be used to achieve the effects of multiple inheritance.
class Animal

class Dog extends Animal

class Cat extends Animal

class Human extends Animal

class Mime extends Human

• *Each has*

  public void speak()
public void speak() in each class contains the single print statement:

- **Animal:**
  System.out.println("* generic animal noise *");

- **Dog:**
  System.out.println("woof");

- **Cat:**
  System.out.println("meow");

- **Human:**
  System.out.println("hello");

- **Mime:**
  System.out.println();
Animal [] animals = new Animal[MAX];

animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Human();
animals[3] = new Dog();
animals[4] = new Mime();
animals[5] = new Cat();
animals[6] = new Animal();

for ( int i = 0 ; i < MAX ; i++ )
{
    animals[i].speak();
}
Motivation for Abstract Classes

• We don't need to instantiate Animal

• Animal is a place to hold common features and behaviors in the hierarchy.

• So make Animal an abstract class.
abstract class Animal
{
    //-----------------------------------------------------
    abstract public void speak();
    //-----------------------------------------------------
}

The abstract speak method is like a promise: any class which extends class Animal will contain a concrete speak method or will be abstract.
animals[6] = new Animal();

• Prohibited if Animal class is abstract.
• You can't instantiate an abstract class — it has no instances — it's too non-specific.
Person is an abstract class
abstract class Person
{
    private String name;  // instance variable

    public Person ( String name ) {
        this.name = name;  // constructor
    }

    public String toString() {
        return name;  // instance method
    }

    abstract public double pay();  // abstract method (stub)

}
class Volunteer extends Person
{
    //------------------------------
    public Volunteer ( String name )
    {
        super(name);           // calling superclass constructor
    }
    //------------------------------
    public double pay()
    {
        System.out.println("Thank " + super.toString() + ".");
        return 0.0;             // calling superclass toString method
    }
    //-----------------------------
}
This abstract class contains no abstract methods except the one defined in the Person class

abstract class Employee extends Person
{
    protected String ssn;

    public Employee(String name, String ssn)
    {
        super(name);
        this.ssn = ssn;
    }

    public String toString()
    {
        return super.toString() + " (" + ssn + ")";
    }
}

This abstract class contains no abstract methods except the one defined in the Person class
class Salaried extends Employee
{
    private double weeklySalary;
    //calling superclass constructor
    public Salaried ( String name, String ssn, double weeklySalary )
    {
        super(name, ssn);
        this.weeklySalary = weeklySalary;
    }
    //initializing field in subclass
    public double pay()
    {
        System.out.printf("Pay $%7.2f to %s\n", weeklySalary,
                          super.toString());
        return weeklySalary;
    }
}

//adding subtype field to superclass toString
class Hourly extends Employee
{
    protected double hourlyRate;
    protected double hoursWorked;

    // calling superclass constructors
    public Hourly ( String name, String ssn, double hourlyRate )
    {
        super(name, ssn);
        this.hourlyRate = hourlyRate;
        this.hoursWorked = 0.0;
    }

    // initializing fields in subclass
    public void addHours ( double hours ) { hoursWorked += hours; }

    // calling superclass toString after printing subclass field
    public double pay()
    {
        double amount = hoursWorked * hourlyRate;
        System.out.printf("Pay $%7.2f to %s\n", amount, super.toString());
        hoursWorked = 0.0;
        return amount;
    }
}
class UnionMember extends Hourly
{
    //calling superclass constructors
    public UnionMember ( String name, String ssn, double hourlyRate )
    {
        super(name, ssn, hourlyRate);
        this.hoursWorked = 0.0;
    }
    //initializing field in subclass

    //calling superclass toString after printing subclass field
    public double pay()
    {
        double amount = hoursWorked * hourlyRate;
        if ( hoursWorked > 40 ) amount += 0.5 * (hoursWorked - 40) * hourlyRate;
        System.out.printf("Pay $%7.2f (includes overtime) to %s.\n",
                amount,
                super.toString());
        hoursWorked = 0.0;
        return amount;
    }
}
The Two Meanings of "Interface"

• 1 — The set of public methods offered by a class.

   Rule: Change the implementation but don't change the interface.

• 2 — A set of public method stubs which may be offered by any number of classes.

   A class may implement an interface if it offers the methods promised by that interface.
public interface Talker
{
    public void talk ( PrintStream ps );  \textit{method stub}
}

Interface Talker
class Dog implements Talker
{
    public void talk ( PrintStream ps )
    {
        ps.println("Woof!");
    }
}
class Cat implements Talker {
    public void talk ( PrintStream ps ) {
        ps.println("Meow!");
    }
}
class Human implements Talker {

    private String name;

    public Human ( String name ) {
        this.name = name;
    }

    public void talk ( PrintStream ps ) {
        ps.println("My name is " + name + ".");
    }

    // implementation of supertype method stub
}
class Mime implements Talker
{
    private String name;

    public Mime ( String name )
    {
        this.name = name;
    }

    public void talk ( PrintStream ps )
    {
        ps.println(" ");
    }
}
Variables Can Be of Type `Talker`

```java
Talker t;
t = new Dog();
t = new Cat();
t = new Mime("Marcel");
t = new Human("Ashton");
```

The variable `t` can refer to objects drawn from different classes, provided that the classes implement `Talker`. This is polymorphism.

A variable that can hold a reference to an object of type `Talker` can also hold a reference to an object belonging to any subtype of `Talker`. This can change over time.
Code:

t = new Dog();
t.talk();
t = new Cat();
t.talk();
t = new Mime("Marcel");
t.talk();
t = new Human("Ashton");
t.talk();

Output:

Woof!
Meow!
My name is Ashton.
Arrays Can Be of Type Talker

Talker[] a = new Talker[5];

t[0] = new Dog();
t[1] = new Cat();
t[2] = new Human("Bill");
t[3] = new Mime("Marcel");
t[4] = new Human("Sue");
Talker[] a = new Talker[5];

a[0] = new Dog();
a[1] = new Cat();
a[2] = new Human("Bill");
a[3] = new Mime("Marcel");
a[4] = new Human("Sue");
Polymorphism

```java
for ( int i = 0 ; i < a.length ; i++ ) {
    a[i].talk(System.out);
}
```

Woof!
Meow!
My name is Bill.

My name is Sue.

In the statement `a[i].talk(...)`, the actual talk method invoked is not known until run-time.
Lab 6

Implement a structurally-recursive linked list hierarchy of classes.

Recall the cons list structure from CMPU 101. This list contained 2 parts: first and rest.

The classes you will implement are called ConsList (for a non-empty list), and MTList (for empty).
Lab 6

This lab exercise will use polymorphism because both the ConsList and MTList classes will be subtypes of an interface called IList.

The ConsList class will have 2 instance variables: int first and IList rest. It will also provide implementations of all methods specified in the IList interface.

The MTList class will have no instance variables but it will provide implementations of all methods specified in the IList interface.
The IList interface will contain the following method "stubs":

1. int length(): Consumes nothing and returns the number of items in the list.
2. IList cons(int): Adds element of type int to the front of the list and returns the new list.
3. int sum(): Returns the sum of the numbers in the list.
4. IList doubleList(): Returns a list containing numbers that are each double the value of the numbers in the given list.

A stub is an unimplemented method. Specifies only the return type, method name, and parameter list.