

CS1020: DATA STRUCTURES AND ALGORITHMS I

Tutorial 4 – Inheritance, Polymorphism

(Week 6, starting 15 February 2016)

1. Inheritance

New McDonald has a farm with some animals.

Each animal has a name, and makes a sound. Some animals are flyers. These animals can fly – once they start to fly, their sound is “flap” till they stop. Some flyers are gliders. These animals can glide – if they are flying, once they start to glide, their movement is “whoosh” till they stop.

Your junior has written this code to describe the animals in the farm. However, his code cannot be compiled, and will not work properly. There are 3 errors within the 3 classes; 1 of them affects compilation.

Tip: Try to compile your code in sunfire! Once the compilation error is rectified, instantiate objects and test.

```
class Animal {
    private String _name; // e.g. Cow
    private String _sound; // e.g. Moo
    public Animal(String name, String sound) {
        _name = name; _sound = sound; }
    public String getName() { return _name; }
    public void makeSound() {
        System.out.println(_name + " goes " + _sound); }
}

class Flyer extends Animal {
    protected String _name;
    protected String _sound;
    protected boolean _isFlying;
    public Flyer(String name, String sound) { Animal(name, sound); }
    public void makeSound() {
        if (_isFlying)
            System.out.println(getName() + " goes flap");
        else super.makeSound();
    }
    public void fly() { _isFlying = true; }
    public void stop() { _isFlying = false; }
}

class Glider extends Flyer {
    private boolean _isGliding;
    public Glider(String name, String sound) { super(name, sound); }
    public void glide() { if (_isFlying) _isGliding = true; }
    public void stop() { _isFlying = false; _isGliding = false; }
    public void makeSound() {
        if (_isGliding)
            System.out.println(getName() + " goes whoosh");
        else makeSound();
    }
}
```

- (a) What does the **protected** keyword mean? In this example, how is it useful?
- (b) Within a method in the `Flyer` class, why can `getName()` be invoked?
- (c) How is overriding demonstrated here, and how is it useful?
- (d) Identify and rectify the 3 errors.

Related Concepts

- Member inheritance
- Overriding, **super** keyword
- Only instance methods overridden
- Constructors chained, **NOT**

Answers

(a)

protected is an access modifier, in between **public** and the default modifier. Protected access means the **member is accessible from subclasses, but not other** classes (disregarding the concept of package). This allows subclasses to use the superclass' data and functionality, while protecting them from the outside world. The keyword can be applied to both attributes and methods.

In this example, we want to encapsulate whether a `Flyer` object is currently flying, from the outside world, but allow a `Glider` object (which IS also a `Flyer`) to read and modify this data.

(b)

Protected and public **members are inherited** from the superclass `Animal` into the subclass `Flyer`, both methods and attributes. This means that a `Flyer` object uses the `getName()` **behavior implemented within the superclass** `Animal`. In this example, `getName()` is equivalent to **super**.`getName()`.

Similarly, the protected attribute `_isFlying` can be used within a method in the `Glider` class because it is inherited. Such use is equivalent to **super**.`_isFlying`.

(c)

`Flyer` and `Glider` have their own implementations of `makeSound()` and `stop()`. A `Glider` object would **choose to use its own implementation of the method** `Glider.stop()` instead of `Flyer.stop()`, because `Glider`'s `stop()` is said to override `Flyer`'s `stop()`.

Through overriding, for the same functionality, we can selectively make subclasses have different behavior from that of their superclass.

Something extra: Can we call **super.super.method()**? See

<http://stackoverflow.com/questions/586363/why-is-super-super-method-not-allowed-in-java>

(d)

Error 1 – Attributes **DO NOT override** attributes with the same name from the superclass, unlike instance methods. Therefore, a `Flyer` object has two names, `Animal._name` (hidden from the `Flyer` class because it is private) and `Flyer._name`! This point and warning holds true for class methods (static methods) as well.

Lesson:

In a subclass, **avoid declaring attribute that has the same name** as that of its superclass' attribute. You will confuse yourself.

If `Animal._name` is **protected** or **public**, it will be shadowed (hidden) by `Flyer._name`.

Solution: As an `Animal` already has a name and sound, we **do not want to duplicate these data** for `Flyers`. We can use the `getName()` and `makeSound()` inherited from `Animal` to achieve the needed functionality. Therefore, within the `Flyer` class, strings `_name` and `_sound` should not be declared.

Error 2 – Within `Glider.makeSound()`, there is a call to `makeSound()`. As there are 3 implementations of `makeSound()` in `Glider` and its superclasses, the **lowest in the inheritance hierarchy will be chosen**, `Glider.makeSound()`. This means that we have a method that repeatedly invokes itself. After resolving the compilation error, a call to this method will result in infinite recursion.

Solution: After overriding `Flyer.makeSound()` with `Glider.makeSound()`, we want to **use the parent functionality**. Use the `super` keyword to achieve this, i.e. `super.makeSound()`.

Error 3 – For compilation errors, **learn to read error messages** and identify the problems. At the very least, you may be able to identify where the error occurred, based on the **line number**.

<file name>:<line>:<message>

Line of code, with error here

^

Test.java:12: error: constructor Animal in class Animal cannot be applied to given types;

```
public Flyer(String name, String sound) { Animal(name,
sound); }
```

^

required: String,String

found: no arguments

reason: actual and formal argument lists differ in length

Test.java:12: error: cannot find symbol

```
public Flyer(String name, String sound) { Animal(name,
sound); }
```

^

symbol: method Animal(String,String)

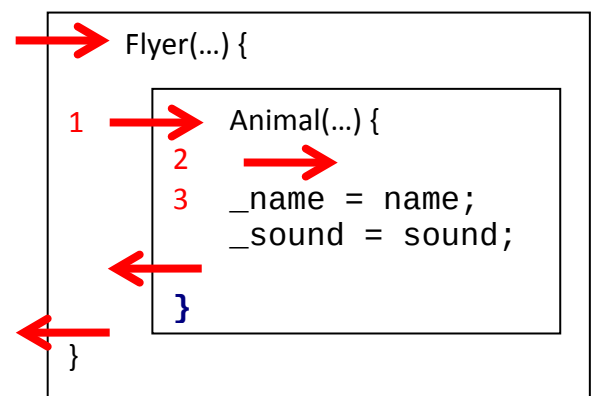
location: class Flyer

The first error shows that we did not call an appropriate constructor in the `Animal` class. This shows that **constructor execution is chained**. (See diagram on the right.)

Before running the body of a subclass constructor, all the **constructors above it have to complete execution first**, so that the attributes belonging to the superclass are **properly initialized**.

Therefore, the first constructor to complete execution is that of `java.lang.Object`, the superclass of all classes.

The second error shows that we are attempting to call a method with signature `Animal(String, String)` in the `Flyer` class, which does not exist. This shows that **constructors are NOT inherited**.



Solution: We have to **invoke the superclass constructor as the first statement** in the subclass, using the `super (...)` keyword.

The exception to this rule is when the superclass has a default constructor or a no-argument constructor. If `Animal` class does not have any constructor declared, or has a constructor that has no parameters, then the compiler implicitly makes a call to `super()` in the absence of `super(String, String)`.

Corrected Code Snippet

```
class Animal {
    private String _name;
    private String _sound;
    public Animal(String name, String sound) {
        _name = name; _sound = sound; }
    public String getName() { return _name; }
    public void makeSound() {
        System.out.println(_name + " goes " + _sound); }
}

class Flyer extends Animal {
    protected boolean _isFlying; // Error 1: Shadowed variables removed
    public Flyer(String name, String sound) { super(name, sound); }
    public void makeSound() { // Error 3: Superclass constructor call
        if (_isFlying)
            System.out.println(getName() + " goes flap");
        else super.makeSound();
    }
    public void fly() { _isFlying = true; }
    public void stop() { _isFlying = false; }
}

class Glider extends Flyer {
    private boolean _isGliding;
    public Glider(String name, String sound) { super(name, sound); }
    public void glide() { if (_isFlying) _isGliding = true; }
    public void stop() { _isFlying = false; _isGliding = false; }
    public void makeSound() {
        if (_isGliding)
            System.out.println(getName() + " goes whoosh");
        else super.makeSound(); // Error 2: Superclass method call
    }
}
```

2. Inheritance & Polymorphism

Related to T2Q2 and T4Q1, New McDonald still has a farm with some animals. You want to **make use of polymorphism** to allow 5 animals in the farm to makeSound(), without having to concern yourself about what type of Animal each is. To simplify things, let's ignore Gliders. There are only Flyers and non-Flyers.



Parrot
"squak"
Perched



Cow
"moo"



Mosquito
"buzz"
Flying



Sheep
"mehh"



Fish
"blurp"

Picture credits: clipartpanda.com

```
class Animal { ... }
class Flyer extends Animal { ... }

public class NewMcDonald {
    private Animal[] _farm; // New McDonald had a farm (still has now)
    private static final int SIZE = 5; // Fixed farm size of 5

    public NewMcDonald() {
        /* TODO: Create your farm, an Animal array */
    }

    public void makeSomeNoise() {
        /* TODO: Make sound(s) without looking out for Flyers...! */
    }

    public void fillThisFarm() {
        _farm[0] = new Flyer("Parrot", "squak");
        _farm[1] = new Animal("Cow", "moo");
        _farm[2] = new Flyer("Mosquito", "buzz");
        ((Flyer)_farm[2]).fly();
        _farm[3] = new Animal("Sheep", "mehh");
        _farm[4] = new Animal("Fish", "blurp");
    }
}
```

- (a) What is the data type of `_farm[0]`? Why can a Flyer object be assigned to `_farm[0]`?
- (b) Why can't `((Flyer)_farm[2]).fly()` be replaced with `_farm[2].fly()`?
- (c) Complete the constructor and `makeSomeNoise()` method. The output of `makeSomeNoise()` is:
Parrot goes squak
Cow goes moo
Mosquito goes flap
Sheep goes mehh
Fish goes blurp

Tip: If you are unsure of your answers, **try out** the program, and ensure you **revisit lecture** concepts!

Related Concepts

- Substitutability principle
- Compile-time: Method must be in ref. data type (or above)
- Runtime: Implementation follows object data type (or above)

Answers

(a)

Datatype of `_farm[0]` is `Animal`. Each array element stores the address of an `Animal` object.

In Java, we can **substitute a superclass-typed object with a subclass-typed object**. In other words, since we expect the address of an `Animal` to be in `_farm[0]`, we allow the array element to store the address of a `Flyer`, because a `Flyer` IS an `Animal`.

(b)

`(_farm[2].)` **dereferences to an `Animal`, which does not have the method `fly()`**. Although this `Animal` is a `Flyer`, the **compiler** does not know that. Therefore, we first have to **explicitly downcast the reference** from supertype `Animal` to subtype `Flyer`. Then, dereference it to get a `Flyer` object, which has `fly()`.

Remember, we are **casting the reference**, not the object. The `Flyer` object is always a `Flyer` object!

In Java, the proper terminology for downcasting is “narrowing reference conversion”.

(c)

Java supports runtime-polymorphism. The method executed will **follow the implementation in the actual object type**, regardless of the reference data type. Even though there is only one type of reference in the `farm` array, there is more than one different implementation of `makeSound()`.

The flying `Mosquito` goes `flap`. As `Mosquito` is a `Flyer`, `_farm[2].makeSound()` uses `Flyer.makeSound()`, and not `Animal.makeSound()`.

```
public NewMcDonald(){ _farm = new Animal[SIZE]; }

public void makeSomeNoise() {
    for (int i = 0; i < SIZE; i++) _farm[i].makeSound();
}
```

3. Enhancing Data Structures

You are given a generic Pair<T, S> class. Each object of this class can refer to 2 objects of different types:

```
class Pair <T, S> {  
    private T _objLeft;  
    private S _objRight;  
  
    public Pair(T pObjLeft, S pObjRight) {  
        _objLeft = pObjLeft;  
        _objRight = pObjRight;  
    }  
    public T getLeft() { return _objLeft; }  
    public S getRight() { return _objRight; }  
}
```

We now want to create a GenericTriple<T, U, V> class. Each object of this class can refer to 3 objects of different types. Restriction: For each part, you may only have **ONE** attribute. There are 2 ways to achieve this:

- (a) Use inheritance.
- (b) Use composition. GenericTriple is composed of a Pair, i.e. it will have a reference to a Pair object as an attribute, and the Pair object only lives as long as the GenericTriple lives.
[Hint: How do you store 3 objects in a Pair? Use 2 Pair objects.]

We can now instantiate a GenericTriple object that refers to 3 objects. A Person has a name (String), weight (double), and height (double). Create a Person data structure and use a GenericTriple to help you store data:

- (c) Use inheritance. [Hint: Are you inheriting a family of classes, or just one specific type?]
- (d) Use composition. Person is composed of a GenericTriple.

Tip: Don't test your brain power!

Draw out how each data structure should look like in memory **BEFORE** coding.

Answer

```
class GenericTripleInh <T, U, V> extends Pair <T, V> { // (a)
    private U _objMid; // extending left and right is cleanest
                       // otherwise, inherited getters will return wrong data
    public GenericTripleInh(T pObjLeft, U pObjMid, V pObjRight) {
        super(pObjLeft, pObjRight);
        _objMid = pObjMid;
    }
    public U getMid() { return _objMid; }
}

class GenericTripleComp <T, U, V> { // (b)
    private Pair<T, Pair<U,V>> _objPair;

    public GenericTripleComp(T pObjLeft, U pObjMid, V pObjRight) {
        _objPair = new Pair<T, Pair<U,V>>(pObjLeft,
            new Pair<U,V>(pObjMid, pObjRight));
    }
    public T getLeft() { return _objPair.getLeft(); }
    public U getMid() { return _objPair.getRight().getLeft(); }
    public V getRight() { return _objPair.getRight().getRight(); }
}

class PersonInh extends GenericTriple <String, Double, Double> { // (c)
    public PersonInh(String pstrName, double pdblWt, double pdblHt) {
        super(pstrName, pdblWt, pdblHt); }
    public String getName() { return getLeft(); }
    public double getWt() { return getMid(); }
    public double getHt() { return getRight(); }
} // now we have 3 inherited getter methods left hanging...

class PersonComp { // (d)
    private GenericTriple<String, Double, Double> _objTriple;
    public PersonComp(String pstrName, double pdblWt, double pdblHt) {
        _objTriple = new GenericTriple<String, Double, Double>(
            pstrName, pdblWt, pdblHt);
    }
    public String getName() { return _objTriple.getLeft(); }
    public double getWt() { return _objTriple.getMid(); }
    public double getHt() { return _objTriple.getRight(); }
} // why does a Double work, even though defined return type is double?
```

- You are now halfway through ☺ -

Have you been revising, learning,
and ***practicing*** daily?