1.00 Lecture 14

Inheritance, part 2

Reading for next time: Big Java: sections 9.1-9.4

Exercise: Plants

- Create a base class `Plant`: (File->New->Class)
  - Private `Strings genus, species, boolean isAnnual`
  - Write the constructor
    `public Plant( …) { … }`
- Create a derived class `Tree`: (File->New->Class)
  - Class declaration `extends _____`
  - Private data `Strings barkColor, leafType`
  - Write the constructor
    `public Tree( …) { … }`
    - Use `super( … )` to call its superclass constructor
    - All trees are perennials

Plant Exercise, p.2

- Create a derived class `Flower`: (File->New->Class)
  - Class declaration `extends _____`
  - Private data `String petalColor`
  - Write constructor
- Create a derived class `Rose`: (File->New->Class)
  - Class declaration `extends _____`
  - Private data `boolean isHybrid`
  - Write constructor
  - All roses are perennials

Exercise, p.3

- Write a class `PlantTest`
  - It has just a `main()` method, which:
    - Creates a `Plant, Tree, Flower, Rose`
    - Genus and species examples:
      - Rosa villosa (rose)
      - Quercus alba (white oak)
      - Narcissus jonquilla (daffodil)
      - Nabalus boottii (Boott's rattlesnake root)
    - The other data is:
      - Bark color= brown, leaf type= rounded for oak
      - Petal color= red for rose, yellow for daffodil
      - Rosa villosa is not hybrid
      - Nabalus is perennial, Narcissus is annual
  - Step through the debugger to see how the constructors are called (Run->Debug as)

Abstract classes

- Classes can be very general at the top of a class hierarchy.
  - For example, MIT could have a class `Person`, from which `Employees`, `Students`, `Visitors`, `Faculty` inherit
  - `Person` is too abstract a class for MIT to ever use in a computer system but it can hold `name`, `address`, `birthdate`, etc. that is in common to all the subclasses
  - We can make `Person` an abstract class: `Person` objects cannot be created, but subclass objects, such as `Student`, can be
- Example:

```
public abstract class Person {
   private String name;
   protected String address;
   public Person(String n, String a) {
         name= n; address= a; }
   // And additional methods}
```

Abstract classes, p.2

- Another example (leading to graphics in the next lectures)
  - `Shape` class in a graphics system
  - `Shapes` are too general to draw; we only know how to draw specific shapes like circles or rectangles
  - `Shape` abstract class can define a common set of methods that all shapes must implement (e.g., draw()), so the graphics system can count on certain things being available in every concrete class
  - `Shape` abstract class can implement some methods that every subclass must use, for consistency: e.g., `getObjectID()`, `getForegroundColor()`

Shape class

```
public abstract class Shape {
   public abstract void draw();
   // Drawing function must be implemented in each concrete
   // derived class but no default is possible: abstract

   public void setVisible(boolean v) { … }
   // setVisible function must be implemented in each derived
   // class and a default is available: non-abstract method

   public final int objectID() { … }
   // Object ID function: each derived class must have one
   // and must use this implementation: final method

   …};

public class Square extends Shape {…};
public class Circle extends Shape {…};
```

Abstract class, method

- Shape is an abstract class (keyword)
  - No objects of type Shape can be created
- Shape has an abstract method draw()
  - draw() must be redeclared by any concrete (non-abstract) class that inherits it
  - There is no definition of draw() in Shape
  - This says that all Shapes must be drawable, but the Shape class has no idea of how to draw specific shapes

Non-abstract method

- Shape has a non-abstract method `setVisible()`
  - Each derived class may define its own setVisible method using this signature, overriding the superclass method
  - Or it may use the super class implementation as a default
- If it overrides the superclass method, it must have exactly the same signature as the superclass method
  - If you write a method with same name but different arguments, it's considered a new method in the subclass
- Be careful. If new derived classes are added and you fail to review and, if needed, redefine non-abstract methods, the default will be invoked but may do the wrong thing
  - E.g., kangaroos

Final method

- Shape has a final method `objectID`
  - Final method is invariant across derived classes
  - Behavior is not supposed to change, no matter how specialized the derived class becomes
- Super classes should have a mix of methods
  - Don't make all abstract super class methods abstract
  - If you can make methods final, do so

An aside: final classes

- To prevent someone from inheriting from your class, declare it final:

```
public final class Grad extends Student { …
```

  - This would not allow SpecGrad to be built
  - Class can have `abstract`, `final` or no keyword

Exercise: Vehicle

```
public abstract class Vehicle {       // In your download
   private int ID;
   protected double mass;
   protected double maxSpeed;
   protected String name;
   private static int nextID= 1;

   public Vehicle(double mass, double maxSpeed, String name) {
        ID++;
        this.mass = mass;
        this.maxSpeed = maxSpeed;
        this.name = name;
}
```

- Write abstract getSafetyRating() method
- Write non-abstract getMaxEnergy() method
  - Returns 0.5*mass*maxSpeed^2; used to design brakes
- Write final method getID() that returns ID

Exercise, p.2

- **Write a concrete Jeep class**
  - Extends Vehicle
  - Has additional private variable: double maxGrade (0-1.0)
  - Write constructor
  - Write getSafetyRating() method
    - Returns max((100- 100*maxGrade – 0.5*maxSpeed), 0)
    - Must have same signature as base class (omit `abstract`)
    - What happens if you don't write one?
  - Write getMaxEnergy() method
    - Return 0.5*mass*maxSpeed^2 + 9.8*mass*maxGrade*100
      - This reflects jeep use on steep grades
    - You are overriding the superclass method that Jeep inherits from Vehicle
    - You may call super.getMaxEnergy() but it is not mandatory
    - What happens if you don't write one?
  - Try to write a getID() method
    - What happens?

Exercise, p.3

- Write a class VehicleTest with main() that:
  - Tries to create a Vehicle object
    - What happens? Comment it out if it doesn't work.
  - Creates a Jeep object
    - Mass 2000 kg, maxSpeed 30 m/sec, max grade 0.2, "jeep"
  - Prints its safety rating
  - Prints its max energy
  - Prints its ID

Fun with animals

```
public class Bird {
  public void fly();          // Birds can fly
   // Method body omitted
  };
```

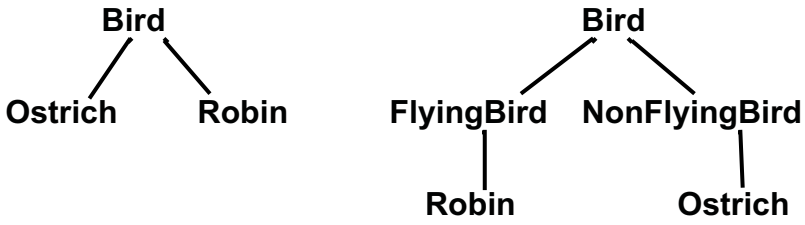
Fun with animals

```
public class Bird {
  public void fly();          // Birds can fly
   // Method body omitted
  };

public class Ostrich extends Bird {  // Ostriches are birds
   // Class body omitted
  };
```

Fun with animals

```
public class Bird {
  public void fly();          // Birds can fly
   // Method body omitted
  };

public class Ostrich extends Bird {  // Ostriches are birds
   // Class body omitted
  };

//  Problems:
//  If superclass method fly() is final, Ostriches must fly

//  If superclass method fly() is abstract or non-abstract,
//  Ostrich's fly() can print an error, etc. It's clumsy

//  With inheritance, every subclass has every method and
//  data field in the superclass. You can never drop
//  anything. This is a design challenge in real systems.
```

Possible solutions

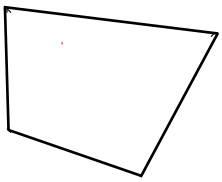


Decision depends on use of system:

If you're studying feet, difference between flying and not flying may not matter

More issues

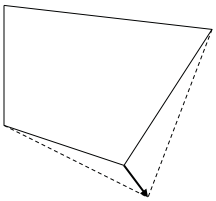
Quadrilateral



Rectangle

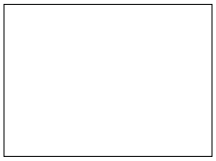
More issues

Quadrilateral



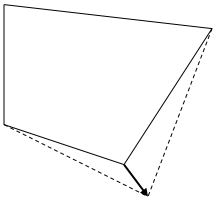
moveCorner()

Rectangle



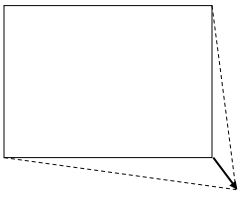
More issues

Quadrilateral



moveCorner()

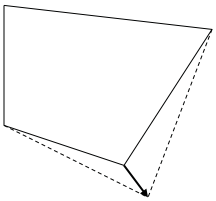
Rectangle



moveCorner()

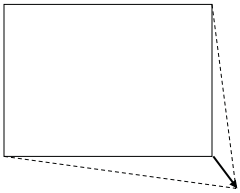
More issues

Quadrilateral



moveCorner()

Rectangle



moveCorner()

Must override the moveCorner() method in subclasses to move multiple corners to preserve the correct shape

1.00 / 1.001 / 1.002 Introduction to Computers and Engineering Problem Solving
Spring 2012