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`:**
  - **Private data `genus`, `species`, `isAnnual`**
  - **Write the constructor**
- **Create a derived class `Tree`:**
  - **Class declaration `extends` ________**
  - **Private data `crownSize`, `trunkSize`**
  - **Write the constructor**

Exercise

```
class Plant {
   ______________________________
   ______________________________
   ______________________________
   ______________________________
   public Plant(______________________________) {
   ______________________________
   ______________________________
   ______________________________
   ______________________________
   }
}
class Tree extends ______ {
   ______________________________
   ______________________________
   ______________________________
   ______________________________
   public Tree(______________________________) {
   ______________________________
   ______________________________
   ______________________________
   ______________________________
   }
```

Plant Exercise, p.2

- **Create a derived class `Flower`**
  - **Class declaration `extends` ________**
  - **Private data `blossomColor`**
  - **Write constructor**
- **Create a derived class `Rose`**
  - **Class declaration `extends` ______**
  - **Private data `thornDensity`**
  - **Write constructor**
- **Create a derived class `Pine`**
  - **Class declaration `extends` _______**
  - **Private data `needleType`, `coneType`**
  - **Write constructor**

Exercise

```
class Flower extends ______ {
  ____________________________
  ____________________________
  public Flower(_____________________________) {
  ____________________________
  ____________________________
  }
}
class Rose extends ______ {
  ____________________________
  ____________________________
  public Rose(_____________________________) {
  ____________________________
  ____________________________
  }
}
class Pine extends _____ {
  ____________________________
  ____________________________
  public Rose(_____________________________) {
  ____________________________
  ____________________________
}}
```

Exercise, p.3

- **Write a class `PlantTest`**
  - **It has just a `main()` method, which:**
    - **Creates a `Plant, Tree, Flower, Rose, Pine`**
    - **Genus and species examples:**
      - **Pinus contorta**
      - **Rosa villosa**
      - **Quercus alba (white oak)**
      - **Narcissus jonquilla (daffodil)**
      - **Prenanthes boottii (Boott's rattlesnake root)**
    - **Pick other data as you wish**
  - **Step through the debugger to see how the constructors are called**

Constructors

- Sub class invokes constructors of super class
  - Constructors invoked in order of inheritance

```
public class Base{
    public Base() {
        System.out.println("Base"); }  }
public class Derived extends Base {
    public Derived() {
        System.out.println("Derived"); }  }
public class DerivedAgain extends Derived {
    public DerivedAgain() {
        System.out.println("Derived Again"); }  }
public class Constructor1 {
    public static void main(String[] args) {
            DerivedAgain object1= new DerivedAgain();}}
```

```
Output:
  Base
  Derived
  DerivedAgain
```

Default constructor invoked unless another constructor explicitly called. Some constructor must be invoked.

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`, etc. inherit
  - `Person` is too abstract a class for MIT to ever use in a computer system but it can hold name, `address`, `status`, 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
- Classes can be concrete or abstract

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, 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., `objectID`, `objectType`

Shape class

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

   public void error(String message) { … }
   // Error 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 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 `Shape`s 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 `error()`
  - Each derived class may handle errors as it wishes:
    - It may define its own error method using this signature (method arguments/return value)
    - 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 or return type, it's considered a new method in the subclass
    - This is an easy mistake to make. Be careful.
  - This can be dangerous: if new derived classes are added and programmers fail to 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. Take a stand!

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

- Write an abstract Person class
  - Protected variables name, age, address, ID
  - Private static variable nextID
    - Initialize it to 1
  - Write constructor w/name, age, address args
  - Write abstract printData() method
  - Write non-abstract error() method
    - Prints "error" if age<0, or name or address is null
  - Write final method getID()

Exercise, p.2

- Write a concrete Student class
  - Extends Person
  - Has additional private variable: year (undergrad year 1-4)
  - Write constructor
  - Write printData() method
    - Must have same signature as base class'
  - Write error() method
    - Also write error message if year<1 or year>4
  - Try to write a getID() method
    - What happens?

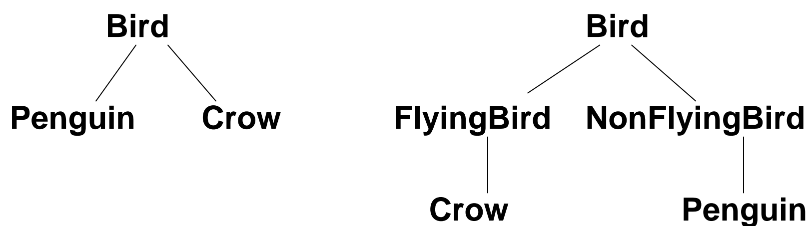
Exercise, p.3

- Write a class AbstractTest
  - Has only a main() method, which:
    - Tries to create a Person object
      - What happens?
    - Creates and prints data for a valid Student
    - Creates an invalid Student with negative age, null address and year=5
    - Calls error method
      - Where should error() really be called?
- Go back to Person and make name private rather than protected
  - What happens?
  - What would you need to add to Person if name were private?

Fun with animals

```
class Bird {
  public void fly();          // Birds can fly
  … };

class Penguin extends Bird {  // Penguins are birds
… };

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

//  If superclass fly() is abstract or non-abstract,
//  Penguin’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

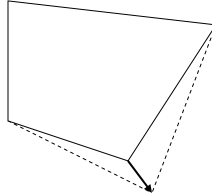


Decision depends on use of system:

If you’re studying beaks, difference between flying and not flying may not matter

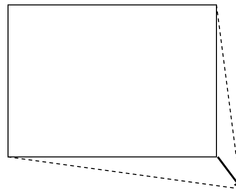
More issues

Quadrilateral



MoveCorner()

Rectangle



MoveCorner()

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