Inheritance

CSE 142, Summer 2003

Computer Programming 1

http://www.cs.washington.edu/education/courses/142/03su/

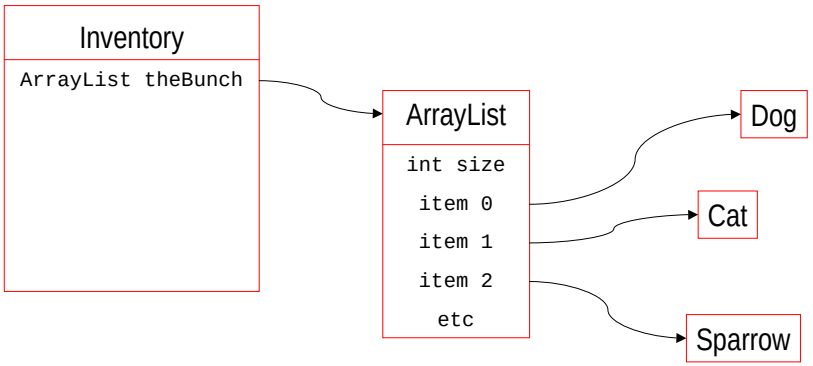
Readings and References

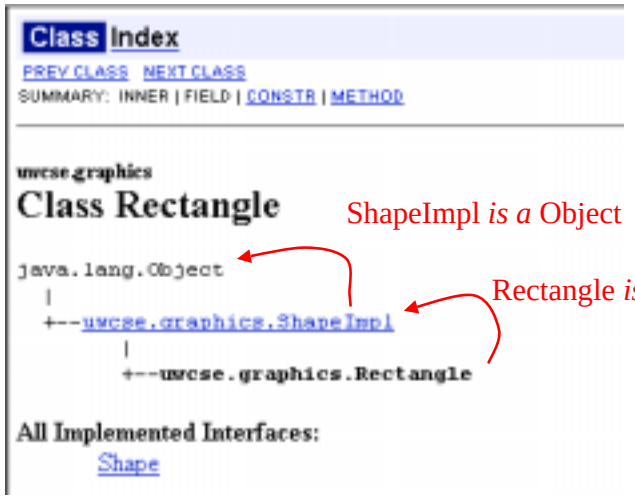
- Reading
  - » Chapter 14 through 14.5, *Intro to Programming and Object-Oriented Design Using Java*, Niño and Hosch
- Other References
  - » Object Basics and Simple Data Objects
  - » Classes and Inheritance
  - » http://java.sun.com/docs/books/tutorial/java/TOC.html#concepts

Relationships between classes

- Classes can be related via composition
  - » This is often referred to as the “has-a” relationship
  - » eg, a PetList *has a* list of Animals
- Classes can also be related via inheritance
  - » This is often referred to as the “is-a” relationship
  - » eg, a String *is a* Object
  - » eg, a Rectangle *is a* ShapeImpl

PetList *has a* list of Animals





Why use inheritance?

- Code simplification
  - » Deal with objects based on their common behavior, and don't need to have special cases for each subtype
  - » Avoid doing the same operation in two places
  - » Avoid storing "matching state" in two places
  - » Lots of elegant code has already been written - use it, don't try to rewrite everything from scratch

Why use inheritance?

- Example: Animals
  - » What is some behavior common to all animals?
    - eat, sleep
  - » What are some attributes common to all animals?
    - mealSize, weight
- We can define behaviors that an Animal must using the Animal interface
- But even with an interface defined, we still need implementations for each method

The Animal interface

```
public interface Animal {
   /**
    * Provide this animal with a way to rest when weary.
    */
    public void sleep();
    /**
    * Eat some goodies.  There is some weight gain after eating.
    * @param pounds the number of pounds of food provided.
    */
    public void eat(double pounds);
    /**
     * get the meal size defined for this animal.
     * @return meal size in pounds
     */
    public double getMealSize();
    /**
    * Provide this animal with a voice.
    */
    public void noise();
}
```

Reduce the need for duplicated code

- Think about our collection of pets
  - » Dog has getMealSize() and eat(double w) methods
  - » Cat has getMealSize() and eat(double w) methods
  - » and they were implemented exactly the same way
- We can define a class named BasicAnimal that implements these methods once, and then the subclasses can extend it and add their own implementations of other methods if they like

BasicAnimal class

Class BasicAnimal

java.lang.Object
  |
  +--BasicAnimal

All Implemented Interfaces:
Animal

Direct Known Subclasses:
Cat, Dog, Sparrow

public abstract class BasicAnimal
extends java.lang.Object
implements Animal

Constructor Summary

BasicAnimal(java.lang.String theName, double serving, double weight)	Create a new BasicAnimal, using supplied parameter values.

Method Summary

void	eat(double pounds) Eat some goodies.
double	getMealSize() get the meal size defined for this animal.
void	sleep() Provide the animal with a way to rest when weary.
java.lang.String	toString() print information about the animal.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface Animal

noise

Dog as a subclass of BasicAnimal

Package Class Tree De

Class Dog

java.lang.Object
  |
  +--BasicAnimal
        |
        +--Dog

All Implemented Interfaces:
Animal

public class Dog
extends BasicAnimal

Constructor Summary

Dog(java.lang.String theName) Create a new Dog with default characteristics.
Dog(java.lang.String theName, double serving, double weight) Create a new Dog, using supplied parameter values.

Method Summary

static void	main(java.lang.String[] args) Run this animal through a typical day.
void	noise() Provide this animal with an appropriate voice.

Methods inherited from class BasicAnimal

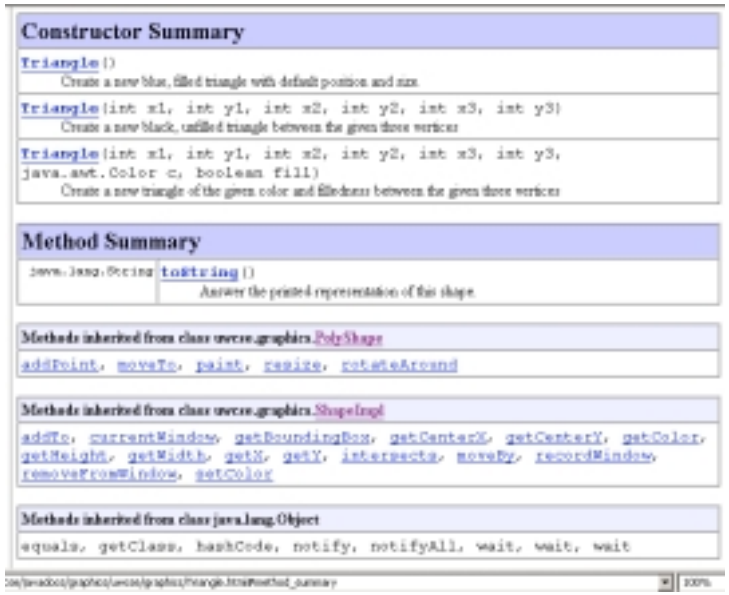
eat, getMealSize, sleep, toString

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Why use inheritance?

- Sometimes it takes several levels of abstraction to get to concrete objects
  - » a Triangle is a PolyShape, which is a ShapeImpl, which is an Object. At each of these levels, there might be behavior to "factor out" or abstract away.
- All Shapes must implement similar methods
  - » we want to do "**`int x = blob.getX()`**"
  - » if both Triangles and Ovals implement this the same way, we can implement getX() in one *base class*, and use it in the *subclasses* instead of rewriting it each time



Constructor Summary

`Triangle()`
Create a new blue, filled triangle with default position and size.

`Triangle(int x1, int y1, int x2, int y2, int x3, int y3)`
Create a new black, unfilled triangle between the given three vertices

`Triangle(int x1, int y1, int x2, int y2, int x3, int y3, java.awt.Color c, boolean fill)`
Create a new triangle of the given color and filledness between the given three vertices

Method Summary

`java.lang.String toString()`
Answer the printed representation of this shape

Methods inherited from class uwcse.graphics.PolyShape

addPoint, moveTo, paint, resize, rotateAround

Methods inherited from class uwcse.graphics.ShapeImpl

addTo, currentWindow, getBoundingBox, getCenterX, getCenterY, getColor, getHeight, getWidth, getX, getY, intersects, moveBy, recordWindow, removeFromWindow, setColor

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Syntax of inheritance

- Specify inheritance relationship using **`extends`**
  - » this is just like we did with interfaces

```
public class Triangle extends PolyShape {
```

```
public abstract class PolyShape extends ShapeImpl  {
        private int npoints;
```

```
public abstract class ShapeImpl implements Shape {
        protected Rectangle boundingBox;
        …
        public int getX() {
                return boundingBox.getX();
        }
}
```


Using the superclass constructor

- Constructor of the superclass is called to do much (or all) of the initialization for the subclass

```
public class Dog extends BasicAnimal {
    public Dog(String theName) {
        super(theName,0.5,20);
    }
    public Dog(String theName,double serving,double weight) {
        super(theName,serving,weight);
    }
```

```
public class BasicAnimal implements Animal {
    public BasicAnimal(String theName,double serving,double weight) {
        name = theName;
        mealSize = serving;
        currentWeight = weight;
        System.out.println("Created "+name);
    }
```


this() and super() as constructors

- You can use an alias to call another constructor
  - » **`super(...)`** to call a superclass constructor
  - » **`this(...)`** to call another constructor from same class
- The call to the other constructor must be the first line of the constructor
  - » If neither this() nor super() is the first line in a constructor, a call to super() is inserted automatically by the compiler. This call takes no arguments. If the superclass has no constructor that takes no arguments, the class will not compile.

Overriding methods

- Overriding methods is how a subclass refines or extends the behavior of a superclass method
- Dog and Cat classes extend BasicAnimal
- How do we specify different behavior for Dogs and Cats?
  - » BasicAnimal:

```
public void sleep() {...}
```

  - » Cat:

```
public void sleep() {... ? ...}
```


Overriding methods

```
public class BasicAnimal {
  // other stuff
  public void sleep() {
    System.out.println(name+" : Snrf ... mutter ... snrf ...");
  }
}


public class Cat extends BasicAnimal {
  // other stuff
  public void sleep() {
    System.out.println(name+" : Snore ... snore ... sigh ...");
  }
}
```


Overriding rules

- A method cannot be made more private than the superclass method it overrides

```
// in superclass
public void sleep() {...}

// in subclass
public void sleep() {...} // valid
private void sleep() {...} // invalid
```


Overriding rules

- A method's return type and parameters must match those in the overridden superclass method exactly in order to override it.

```
// in superclass
public int pay(int hours) {}

// in subclass
public int pay(int b) {}    // okay, overrides
public long pay(int b) {}   // compile error
```

instanceof

- Used to test an object for class membership

```
if (bunch.get(i) instanceof Dog) {…}
```

- One way to ensure that a cast will succeed
- Tests for a relationship anywhere along the hierarchy
- Also can be used to test whether a class implements an interface

```
if (bunch.get(i) instanceof Animal) {…}
```