Inheritance

CSE 142, Summer 2002

Computer Programming 1

http://www.cs.washington.edu/education/courses/142/02su/

Readings and References

- Reading
  - » Sections 14.1 and 14.2 , *An Introduction to Programming and Object Oriented Design using Java*, by Niño and Hosch
- Other References
  - » Sections *Object-Oriented Programming Concepts* and *Classes and Inheritance* of the Java tutorial
  - » http://java.sun.com/docs/books/tutorial/java/

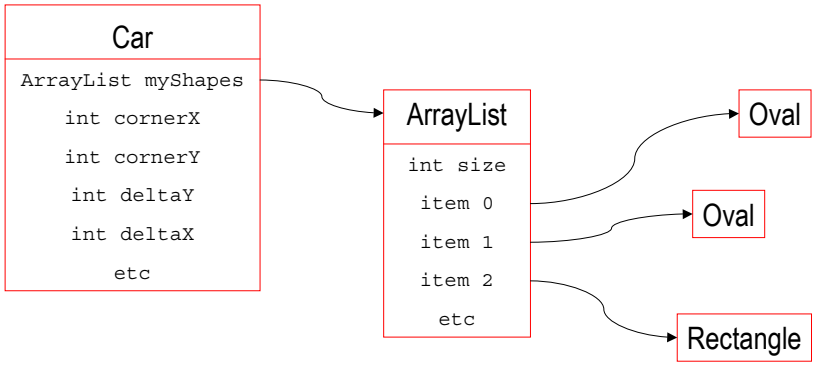
Relationships between classes

- Classes can be related via composition
  - » This is often referred to as the "has-a" relationship
  - » eg, a Car *has a* list in an ArrayList of Shapes
- Classes can also be related via inheritance
  - » This is often referred to as the "is-a" relationship
  - » eg, a Triangle *is a* PolyShape

Composition Vs. Inheritance

- The "has-a" relationship is composition
- The "is-a" relationship is inheritance
- Prefer composition to inheritance
- Beware of inheritance graphs that are either very wide or very deep
  - » very wide means that you are perhaps not abstracting enough at the top layer
  - » very deep means that you are adding only a little functionality at each layer and making fine distinctions that may not survive the test of time

Car *has a* list of Shapes



Class Index

PREV CLASS NEXT CLASS
SUMMARY: INNER | FIELD | CONSTR | METHOD

uwcse.graphics

Class Triangle

is a

is a

is a

```
java.lang.Object
  |
  +--uwcse.graphics.ShapeImpl
        |
        +--uwcse.graphics.PolyShape
              |
              +--uwcse.graphics.Triangle
```

All Implemented Interfaces:
Shape

public class **Triangle**
extends PolyShape
implements Shape

Why use inheritance?

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

Why use inheritance?

- Example: Shapes
  - » What is some behavior common to all shapes?
    movement, intersection
  - » What are some attributes common to all shapes?
    size, location, color
- We defined behaviors that a Shape must have when we discussed the Shape interface
- But even with an interface defined, we still need implementations for each method

The Shape interface

- From OvalSlider.java

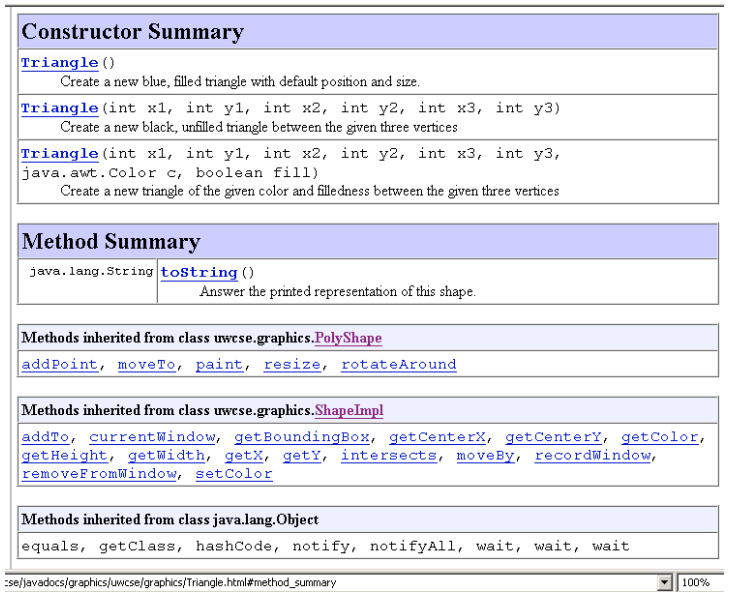
```
/** the Shape that we are moving around on the screen */
private Shape theShape;
```

- OvalSlider doesn't care about the special characteristics of an Oval, it only cares that an Oval can do the things that a good Shape should be able to do

```
void addTo(GWindow gw)
Rectangle getBoundingBox()
int getCenterX()
int getCenterY()
java.awt.Color getColor()
int getHeight()
int getWidth()
int getX()
int getY()
uwcse.graphics.InternalGWindow currentWindow()
boolean intersects(Shape other)
void moveBy(int deltaX, int deltaY)
void moveTo(int x, int y)
void paint(java.awt.Graphics g)
void recordWindow(uwcse.graphics.InternalGWindow gw)
void removeFromWindow()
void rotateAround(int pivotX, int pivotY, double degrees)
void setColor(java.awt.Color c)
```


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();
        }
}
```

Reduce the need for duplicated code

- Remember our collection of pets?
  - » Dog has getMealSize() and eat(double w) methods
  - » Cat has getMealSize() and eat(double w) methods
  - » and they are 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

Package Class Tree Dep

PREV CLASS NEXT CLASS

SUMMARY: NESTED | FIELD | CONSTR |

Class BasicAnimal

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

All Implemented Interfaces:

Animal

Direct Known Subclasses:

Cat, Dog, Sparrow

public 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.
static void	main(java.lang.String[] args) Run this animal through a typical day.
void	noise() Provide this animal with a voice.
void	sleep() Provide this animal with a way to rest when weary.
java.lang.String	toString() print information about this animal.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Dog as a subclass of BasicAnimal

Package Class Tree De

PREV CLASS NEXT CLASS

SUMMARY: NESTED | FIELD | CONSTR

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

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
- Manager and Executive classes extend Employee
- How do we specify different behavior for Managers and Executives?
  - » Employee:
    double pay() {return hours*rate + overtime*(rate+5.00);}
  - » Manager:
    double pay() {return hours*rate;}
  - » Executive:
    double pay() {return salary + bonus;}

Overriding methods

```
public class Employee {
   // other stuff
   public float pay() {
       return hours*rate + overtime*(rate+5.00);
   }
}
public class Manager extends Employee {
   // other stuff
   public float pay() {
       return hours*rate;
   }
}
```


Overriding rules

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

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

// in subclass
public void pay() {...} // valid
private void pay() {...} // 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) {…}
```

- Good way to ensure that a cast will succeed
- Tests for a relationship anywhere along the hierarchy
  - » Also tests whether a class implements an interface
- What class must <classname> represent for the following expression to be true always?

```
if (v instanceof <classname>) { … }
```


`instanceof` example with interface

```
ArrayList onStage = theStage.getActors();
for (int i=0; i<onStage.size(); i++) {
    if (onStage.get(i) instanceof ClickableActor) {
        ClickableActor clickee = (ClickableActor)onStage.get(i);
        if (clickee.intersects(cursor)) {
            clickee.doClickAction(theStage);
            if (clickee == runButton) {
               if (runButton.isEnabled()) {
                   theStage.animate();
               } else {
                   theStage.quitAnimation();
               }
            }
        }
    }
}
```