CSC 143 Java

Shape Case Study

interface Shape

- Some operations:

```
public int getX();              public int getY();
public int getCenterX();   public int getCenterY();
public int getWidth();       public int getHeight();
public void moveBy(int deltaX, int deltaY);
public void moveTo(int x, int y);
public void addTo(GWindow gw);
public void removeFromWindow();
public Rectangle getBoundingBox();
public boolean intersects(Shape other);
public void paint(Graphics g);
```

  - Note: this is a different design from the *Shape* interface in the java.awt package

    That *Shape* has 10 methods, mostly different from these

Designer's Dilemma

- Lots of concrete shapes might be needed
  - Oval, Rectangle, Line, Corkscrew, etc.
  - Implementing each from *Shape* may result in lots of duplicated code, lots of reinventing the wheel.
- Common strategy: Define an abstract class to sit between the interface and the concrete classes
  - Include instance variables and sample implementations, helper methods, etc.
  - Concrete class designers inherit from the abstract class

    Use what they need, override what they need
- Then... is the interface still necessary??

abstract class ShapeImpl implements Shape

- Provide default implementation of as many methods of Shape as possible
  - Can override in subclasses if they have a better way to do it
  - Leave others abstract, but can still call them by other non-abstract methods
- Include default representation (instance variables) to support those implementations
  - Cannot override in subclasses, so must be careful!
- If ShapeImpl isn't right for some implementor of Shape, they can always go it alone, and just implement Shape but not extend ShapeImpl

Coordinate-based Methods

- Lots of operations relate to the X, Y, width, & height of the shape
- Can define these in terms of the bounding box of the shape

```
// public abstract Rectangle getBoundingBox();   // inherited from Shape
public int getX() { return getBoundingBox().getX(); }
public int getY() { return getBoundingBox().getY(); }
public int getWidth() { return getBoundingBox().getWidth(); }
public int getHeight() { return getBoundingBox().getHeight(); }
// do intersects as an exercise....
```

- Then can compute center coordinates from these methods

```
public int getCenterX() { return getX() + getWidth()/2; }
public int getCenterY() { return getY() + getHeight()/2; }
```

- All subclasses have to do is implement getBoundingBox(), inherit the rest "for free"

Implementing getBoundingBox()

- Right now, ShapeImpl stores the bounding box as an instance variable, and implements getBoundingBox()

```
protected Rectangle boundingBox;   // set in subclass constructors
public Rectangle getBoundingBox() { return boundingBox; }
```

- What are the advantages of this? disadvantages?

Moving Shapes

- Shapes should implement moveBy and moveTo
- But we can implement one in terms of the other (and getX() and getY())
- One design:

```
// public abstract void moveTo(int x, int y);    // inherited from Shape
public void moveBy(int deltaX, int deltaY) {
    moveTo(getX() + deltaX, getY() + deltaY);
}
```

- Now clients only implement moveTo, inherit moveBy "for free"

Moving Bounding Boxes

- If we move a shape, then we need to move its bounding box, too
- Provide a default implementation of moveTo that does the bounding box updates

```
public void moveTo(int x, int y) {
    getBoundingBox().moveTo(x, y);
}
```

- Subclasses override this implementation to also move the real shape, if necessary. They can refer to the above **moveTo** via **super.moveTo.**

For Subclasses To Do

- ShapeImpl doesn't implement the following:

```
public abstract void paint(Graphics g);
```

- Subclasses should override moveTo, if they need to
- Subclasses should provide constructors
- Subclasses should implement toString

abstract class PolyShape extends ShapeImpl

- An abstract class for all shapes represented with a list of vertices
- Provides a constructor, an addPoint method, a paint method, a toString method
- Overrides moveTo:

```
public void moveTo(int x, int y) {
    //... a lot of code to move each of the vertices ...
    super.moveTo(x, y);  // do the ShapeImpl code
}
```

- Concrete subclasses Polygon, Triangle, and Line are just constructor and toString!

concrete class Rectangle extends ShapeImpl

- Stores x, y, width, and height values directly

```
protected int x;   …
```

- Rectangle is its own bounding box

```
public Rectangle(…) {
    …
    this.boundingBox = this;
}
```

- Must override all operations that would have referenced boundingBox to instead do some real work

```
public void getX() { return x; }
…
```