COMP 401 – Fall 2017

Recitation 6: Inheritance
Agenda

- Brief Review of Inheritance
- Examples of extending existing classes
- Exercises and Quiz
High-level

- Classes are Abstract Data Types
- We can define a set of operations, properties, and semantics via an Interface
  - Why?
  - How?
- One or more classes may then implement a particular Interface
- Interfaces can be extended to define additional operations
  - Does extending an interface make it more or less general/specific?
Consider the `Line` example from the last recitation:

```java
public interface Line {
    public int getX();
    public void setX(int newX);
    public int getY();
    public void setY(int newY);
    public int getWidth();
    public void setWidth(int newVal);
    public int getHeight();
    public void setHeight(int newVal);
}
```

- Defines properties....? Semantics...?
- Suppose we would like to be able to control the thickness of the line. What might this look like?
So let’s add operations to change the “Stroke” width:

```java
public interface StrokedLine extends Line {
    double getStrokeWidth();
    void changeStrokeWidth(double width);
    Stroke getStroke();
}
```
Interface vs. Implementation

- Recall that an Interface only defines a set of methods (and possibly constants)
- To actualize an Interface we must create a *concrete* implementation.
- In our example, we created:

```java
public class ALine implements Line {...}
```

- How can/should we implement our StrokedLine?
AStrokedLine

- We will derive our StrokedLine implementation from our existing Aline implementation:

```java
public class AStrokedLine extends ALine implements StrokedLine {
  protected BasicStroke stroke;
  @Override
  public double getStrokeWidth() {...}
  ...
}
```

- We *inherit* all of the data and methods of ALine
- We implement the new functionality defined by StrokedLine
- Is the derived class more or less general/abstract?
Worked Example
Suppose we would like a line object where the line thickness is automatically scaled as the line’s length changes.

- What might this look like?
- Does the interface fundamentally change?
public class AnAutoScaledStrokedLine extends AStrokedLine {

    @Override
    public void setX(int newX) {
        double oldLength = getLength();
        super.setX(newX);
        updateStrokeWidth(oldLength);
    }

    // Assumes dimensions have already been changed
    protected double updateStrokeWidth(double oldLength) {
        double currentLength = getLength();
        double newStrokeWidth =
            computeNewStrokeWidth(oldLength, currentLength);
        setStrokeWidth(newStrokeWidth);
        return getStrokeWidth();
    }

    ...
We implement the behavior of AnAutoScaledStrokedLine by overriding the setters for the x, y, width, and height properties.

So a call to setX() on an object of type AnAutoScaledStrokedLine will invoke our new code and change the line thickness.

Question: Did I have to override the base class’s constructor?
NB: `getLength()` was made `public` solely for the convenience of being able to display the length in ObjectEditor. Per Dr. Dewan’s conventions, I made it part of a new interface `HasLength`. If I weren’t using ObjectEditor, this method would be `protected`.
Q&A

- What happens if we omit `super` in the new setters?
- What happens if we omit the code at POI #1 in `computeNewStrokeWidth()`:
  ```java
  if (getStroke() == null) {
    return 1.0;
  }
  ```
  - Why is this here?
  - How else might we address this issue?
  - Why might I have chosen to do it this particular way?
- Bonus: is there a way that the author of Aline could have prevented me from overriding the setters?