# CSE 143
# Lecture 19

## Programming with inheritance

slides created by Alyssa Harding
http://www.cs.washington.edu/143/

# Inheritance

- We've seen how the mechanics of inheritance work
- We remember some things about extending classes, super calls, and inherited methods
  - ...right?
- Now we're going to see how we can program with inheritance to make our lives easier

# Example: `StutterList`

- We want a class that has all the functionality of **`ArrayList`** but adds everything twice

- For instance, the following code

```
StutterList<String> s =
          new StutterList<String>();
s.add("hello");
System.out.println(s);
```

outputs

```
["hello","hello"]
```

# Example: `StutterList`

- How would we do this?
- We could write an entirely new class by copying and pasting the **`ArrayList<E>`** code
  - But that's redundant
- We could change the **`ArrayList<E>`** code to include our new functionality
  - But this is invasive change
  - It would ruin any code that depended on the original functionality

# Example: `StutterList`

- We want *additive*, not *invasive*, change!
- Instead, we just want to add onto the **`ArrayList<E>`**
- We want to extend its functionality using inheritance:

```
public class StutterList<E>
                extends ArrayList<E> {

}
```

# Example: `StutterList`

- Now we override the old **`add`** method to include our stutter functionality:

```
public class StutterList<E>
                    extends ArrayList<E> {

  public boolean add(E value) {
     super.add(value);
     super.add(value);
     return true;
  }
}
```

Instead of worrying about the details, we can use the super class's **`add`** method

# Example: `TranslatePoint`

- Or maybe we want a **`Point`** that keeps track of how many times it has been translated:

```
public class TranslatePoint extends Point {
   private int count;

}
```

We need a field to keep track of our new state, but we rely on the **`Point`** class to keep track of the regular state

# Example: `TranslatePoint`

- We then need to override the **`translate`** method to update our new state while still keeping the old functionality:

```
public void translate(int x, int y) {
   count++;
   super.translate(x,y);
}
```

We need to make sure we call the super class' method, otherwise we would have infinite recursion!

# Example: `TranslatePoint`

- We can also add more functionality to the class:

```
public int getTranslateCount() {
   return count;
}
```

# Example: `TranslatePoint`

- We still need to think about constructors.
- Constructors are ***not*** inherited like the rest of the methods.
- Even when we don't specify one, Java automatically includes an empty, zero-argument constructor:

```
public TranslatePoint() {
   // do nothing
}
```

But it doesn't actually do nothing...

# Example: `TranslatePoint`

- Java needs to construct a **`TranslatePoint`**, so it at least needs to construct a **`Point`**

- It automatically includes a call on the super class' constructor:

```
public TranslatePoint() {
   super();
}
```

We can use the **`super()`** notation to call the super class' constructor just like we use the **`this()`** notation to call this class' constructor

# Example: `TranslatePoint`

- But we want to be able to specify coordinates, so we will need to make a constructor

- The first thing we need to do is include a call on the super class' constructor:

```
public TranslatePoint(int x, int y) {
   super(x,y);
   count = 0;
}
```

# Example: `TranslatePoint`

- Now that we have a constructor, Java won't automatically give us a zero-argument constructor

- Since we still want one, we have to explicitly program it:

```
public TranslatePoint() {
   this(0,0);
}
```

# Inheritance Recap

- **Fields**. What additional information do you need to keep track of?
- **Constructors**. If you want a constructor that takes parameters, you have to create one and call the super class' constructor in it.
- **Overridden methods**. What methods affect the new state? These need to be overridden. Again, call the super method when you need to accomplish the old task.
- **Added methods**. What new behavior does your class need?

# Graphics

- Another useful application:
  graphical user interfaces (GUIs)
- Think of all the different programs on your computer. You wouldn't want to code each type of window, textbox, scrollbar individually!
- Java includes basic graphical classes that we can extend to add more functionality
- Here's the API documentation for a frame:
  http://java.sun.com/javase/6/docs/api/java/awt/Frame.html

# Graphics

- We can use this to customize our own type of frame:

```
public class FunFrame extends Frame {
   public FunFrame() {
      setVisible(true);
      setSize(400, 400);
      setTitle("Such fun!");
      setBackground(Color.PINK);
   }
}
```

# Graphics

- We can also override other methods in the class:

```
public void paint(Graphics g) {
   System.out.println("in paint");
}
```

- This will be called whenever the frame needs to be redrawn on the screen

- Play around with more methods!