CSE 143
Lecture 21

Inheritance, `static`

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

Inheritance

- We've seen how the mechanics of inheritance work
- We seen some things about extending classes, super calls, and inherited methods
- 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"]
```

3

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

4

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> {

}
```

5

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

6

Example: `TranslatePoint`

- We want a `Point` that keeps track of how many times it has been translated:

```
import java.awt.*;

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

7

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!

8

Example: `TranslatePoint`

- We can also add more functionality to the class:

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

9

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...

10

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

11

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;
}
```

12

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);
}
```

13

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?

14

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

15

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);
  }
}
```

16

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!

17

What does `static` mean?

- Why `static` in `public static void main`?
- You can think of `static` as meaning that there is only one copy per class.
- A `static` method is also known as a class method.
- A non-`static` method is also known as an instance method.

18

What does `static` mean?

```
public class Counter {
   public static int classCounter;
   public int instanceCounter;

   public Counter() {
      classCounter++;
      instanceCounter++;
   }

   public static void staticIncrement() {
      classCounter++;
      //instanceCounter++; // illegal!  Which instanceCounter would we increment?
   }

   public void instanceIncrement() {
      classCounter++;
      instanceCounter++;
   }

   public void print(String msg) {
      System.out.println(msg);
      System.out.println("class = " + classCounter);
      System.out.println("instance = " + instanceCounter);
      System.out.println();
   }

   public static void main(String[] args) {
      Counter a = new Counter();
      a.print("a created");
      Counter b = new Counter();
      b.print("b created");

      Counter.staticIncrement();
      //Counter.instanceIncrement(); // illegal!  Which instanceIncrement() would we call?

      a.print("after static increment");
      a.instanceIncrement();
      a.print("a instance increment");
      b.print("b print");
   }
}
```

19