Programming Languages and Techniques (CIS1200)

Lecture 22

Java: Objects, Interfaces, Static methods Chapters 19 & 20

Announcements

- HW05: GUI programming
 - Due: Tuesday at 11.59pm
- Java Bootcamp / Refresher: Wednesday, March 19
 - 7-9pm, Towne 100
 - Will be recorded
 - Look for more details on Ed
- HW06: Pennstagram
 - Java array programming
 - Available on course website
 - Due Tuesday, March 25th
- Midterm 2: Friday, March 28th
 - OCaml: ASM, mutability, queues/deques, closures, GUI, and Java basics

Object Oriented Programming

"Objects" in OCaml

```
(* The type of counter objects *)
type counter = {
    inc : unit -> int;
    dec : unit -> int;
}
(* Create a counter "object" *)
let new_counter () : counter =
    let r = {contents = 0} in
    {
        inc = (fun () ->
        r.contents <- r.contents + 1;
        r.contents);
        dec = (fun () ->
        r.contents <- r.contents - 1;
        r.contents)
}</pre>
```

Why is this an object?

- Encapsulated local state only visible to the methods of the object
- Object is *defined by what it* can do—local state does not appear in the interface
- There is a way to *construct* new object values that behave similarly

OO programming



- Explicitly create objects using a record of higher order functions and hidden state
- Flexibility through *composition*: objects can only implement one interface

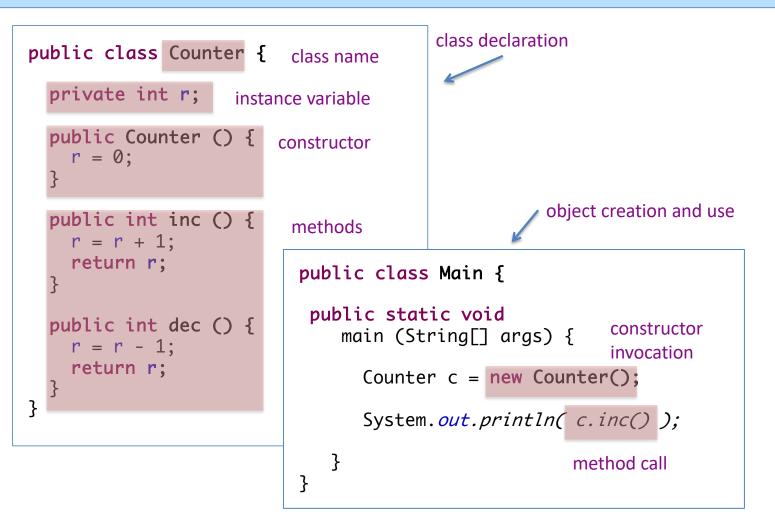
```
type button =
   widget *
   label_controller *
   notifier_controller
```



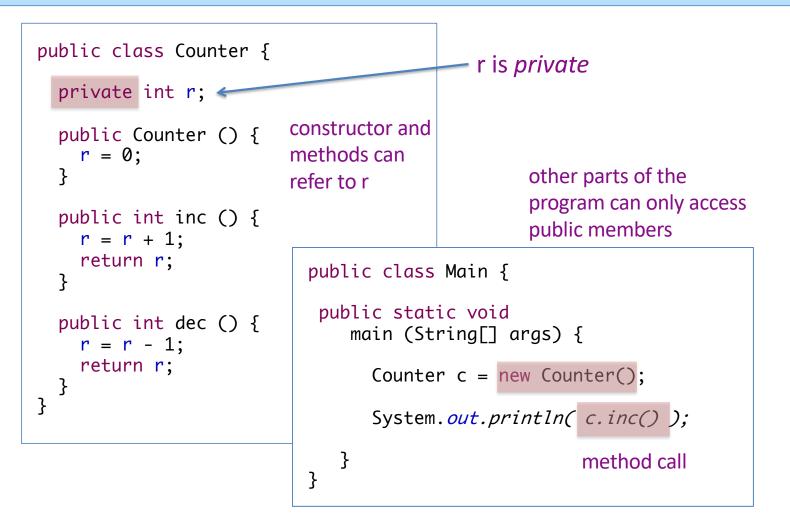
- Primitive notion of object creation (classes, with fields, methods and constructors)
- Flexibility through *extension*:
 Subtyping allows related objects to share a common interface

```
class Button extends Widget {
    /* Button is a subtype
    of Widget */
}
```

Objects in Java



Encapsulating local state



Encapsulating local state

- *Visibility modifiers* make the state local by controlling access
- Basically*:
 - public : accessible from anywhere in the program
 - private : only accessible inside the class
- Design pattern first cut:
 - Make *all* fields private
 - Make constructors and non-helper methods public

*Java offers a couple of other protection levels — "protected" and "package protected" for structure larger code developments and libraries. The details are not important at this point.

Constructors with Parameters

<pre>public class Counter {</pre>		Constructor methods can take parameters
<pre>private int r; public Counter (int r0) { r = r0; }</pre>		Constructor must have the same name as the class
<pre>public int inc () { r = r + 1; return r; }</pre>	public class	object creation and use Main {
<pre>public int dec () { r = r - 1; return r; }</pre>	<pre>public static void constructor main (String[] args) { invocation Counter c = new Counter(3);</pre>	
}	System	.out.println(c.inc());
	} }	

Creating Objects

- *Declare* a variable to hold a **Counter** object
 - Type of the object is the *name* of the class that creates it
- *Invoke* the constructor for **Counter** to create a **Counter** instance with keyword "new" and store it in the variable

Counter c = new Counter();

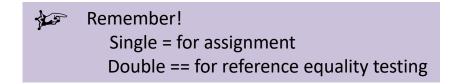
Creating Objects

• Every Java variable is mutable

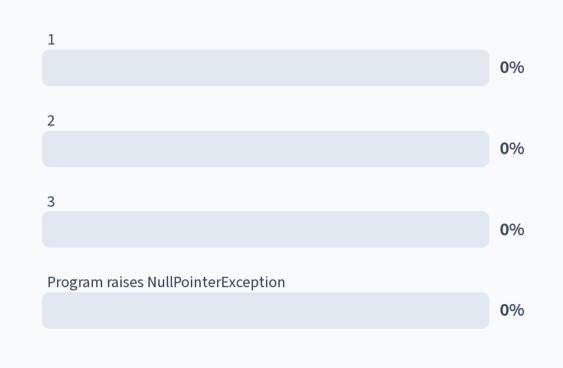
```
Counter c = new Counter(2);
c = new Counter(4);
```

• A Java variable of *reference* type can also contain the special value "null"

Counter c = null;

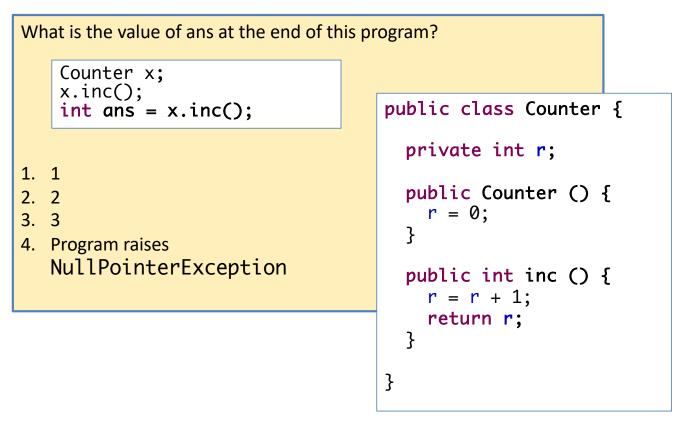


22: What is the value of *ans* at the end of this program?



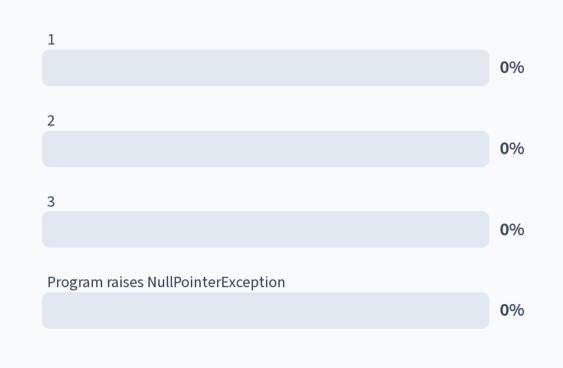
Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

@0



Answer: Program raises NullPointerException

22: What is the value of *ans* at the end of this program?



Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

@0

```
What is the value of ans at the end of this program?
    Counter x = new Counter();
   x.inc();
                                     public class Counter {
    Counter y = x;
    y.inc();
                                       private int r;
   int ans = x.inc();
                                       public Counter () {
1. 1
                                         r = 0;
2. 2
                                       }
3. 3
4. Program raises
                                       public int inc () {
  NullPointerException
                                         r = r + 1;
                                         return r;
                                       }
                                     }
```

Answer: 3

Interfaces

Working with objects abstractly

"Objects" in OCaml vs. Java

```
(* The type of "objects" *)
                                            public class Point {
OCaml
                                                                                   Java
   type point = {
        getX : unit -> int;
                                              private int x;
        getY : unit -> int;
                                              private int y;
        move : int*int -> unit;
   }
                                              public Point () {
                                                \mathbf{X} = 0;
   (* Create an "object" with
                                                y = 0;
       hidden state: *)
                                              ł
   type position =
                                              public int getX () {
      { mutable x: int;
                                                return x;
        mutable y: int; }
                                              public int getY () {
   let new_point () : point =
                                                return y;
      let r = \{x = 0; y=0\} in {
        getX = (fun () \rightarrow r.x);
                                              public void move
        aetY = (fun () -> r.y);
                                                      (int dx, int dy) {
        move = (fun (dx, dy) \rightarrow
                                                \mathbf{x} = \mathbf{x} + \mathbf{d}\mathbf{x};
                                                y = y + dy;
                 r.x < -r.x + dx;
                                              }
                 r.y < -r.y + dy
                                            }
   }
                                                  Class specifies both type and
              Type is separate
                                                  implementation of object values
              from the implementation
```

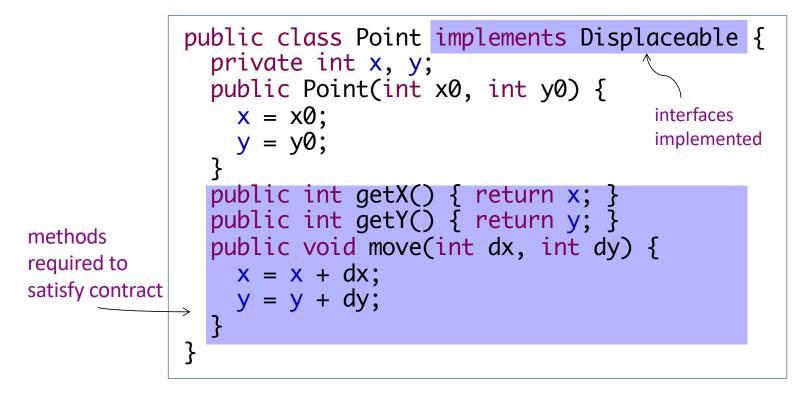
Interfaces

- Give a *type* for an object based on how it can be *used*, not on how it was *constructed*
- Describe a *contract* that objects must satisfy
- Example: Interface for objects that have a position and can be moved

```
public interface Displaceable {
    int getX();
    int getY();
    void move(int dx, int dy);
}
    No fields, no constructors, no
        method bodies!
```

Implementing the interface

- A class that *implements* an interface provides appropriate definitions for the methods specified in the interface
- The class fulfills the contract implicit in the interface



Another implementation

```
public class Circle implements Displaceable {
  private Point center;
  private int radius;
  public Circle(Point initCenter, int initRadius) {
    center = initCenter;
    radius = initRadius;
  public int getX() { return center.getX(); }
  public int getY() { return center.getY(); }
  public void move(int dx, int dy) {
    center.move(dx, dy);
  }
            Objects with different
                                   Delegation: move the
}
            local state can satisfy
                                   circle by moving the
            the same interface
                                   center
```

Yet another implementation

```
public class ColoredPoint implements Displaceable {
  private Point p;
  private Color c;
  public ColoredPoint (int x0, int y0, Color c0) {
     p = new Point(x0,y0);
     C = C0:
  public void move(int dx, int dy) {
     p.move(dx, dy);
  }
  public int getX() { return p.getX(); }
  public int getY() { return p.getY(); }
  public Color getColor() { return c; }
                                          Flexibility: Classes
}
                                          may contain more
                                          methods than
                                          interface requires
```

Interfaces are types

• Can declare variables and method params with interface type

```
void m (Displaceable d) { ... }
```

• Can call m with any Displaceable argument...

obj.m(new Point(3,4)); obj.m(new ColoredPoint(1,2,Color.Black));

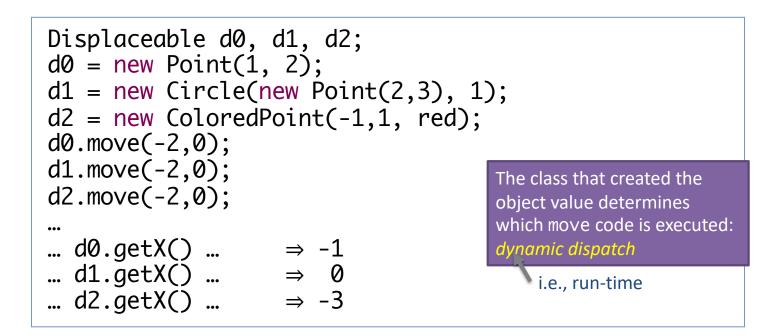
• ... but m can only operate on d according to the interface

d.move(-1,1);
...
... d.getX() ...
$$\Rightarrow 0$$

... d.getY() ... $\Rightarrow 3$

Using interface types

- Variables with interface types can refer, at run time, to objects of any class that implements the interface
- Point and Circle are *subtypes* of Displaceable



Abstraction

The Displaceable interface gives us a single name for all the possible kinds of "moveable things." This allows us to write code that manipulates arbitrary Displaceable objects, without caring whether it's dealing with points or circles.

```
public class DoStuff {
    public void moveItALot (Displaceable s) {
        s.move(3,3);
        s.move(100,1000);
        s.move(1000,234651);
    }
    public void dostuff () {
        Displaceable s1 = new Point(5,5);
        Displaceable s2 = new Circle(new Point(0,0),100);
        moveItALot(s1);
        moveItALot(s2);
    }
}
```

Multiple interfaces

- An interface represents a point of view ...and there can be multiple valid points of view on a given object
- Example: Geometric objects
 - All can move (are Displaceable)
 - Some have Color (are Colored)

Colored interface

- Contract for objects that that have a color
 - Circles and Points don't implement Colored
 - ColoredPoints do

public interface Colored {
 public Color getColor();
}

ColoredPoints

```
public class ColoredPoint
implements Displaceable, Colored {
    ... // previous members
    private Color color;
    public Color getColor() {
        return color;
    }
    ...
}
```

"Datatypes" in Java

OCaml	Java
<pre>type shape =</pre>	<pre>interface Shape { void draw(); } class Point implements Shape {</pre>
<pre>let draw_shape (s:shape) = begin match s with</pre>	<pre> "" public void draw() { "" } class Circle implements Shape { "" public void draw() { "" } }</pre>

Recap: OO terminology

- **Object**: A collection of related *fields* (or *instance variables*) and *methods* that operate on those fields
- Instantiation: Every (Java) object is an *instance* of some class
 - Instances are created by invoking a constructor with the new keyword
- **Class**: A template for creating objects, specifying
 - types and initial values of fields
 - code for methods
 - optionally, a *constructor* that is run each time a new object is created from the class
- Interface: A "signature" for objects, describing a collection of methods that must be provided by classes that *implement* the interface
- **Object Type**: Either a class or an interface (meaning "this object was created from a class that implements this interface")

Static Methods

Java Main Entry Point

- Program starts running at main
 - args is an array of Strings (passed in from the command line)
 - must be public
 - returns void (i.e. is a command)
- What does *static* mean?

Static method example

