Programming Languages and Techniques (CIS120)

Lecture 22

Java: Objects, Interfaces Chapters 19 & 20

Object Oriented Programming

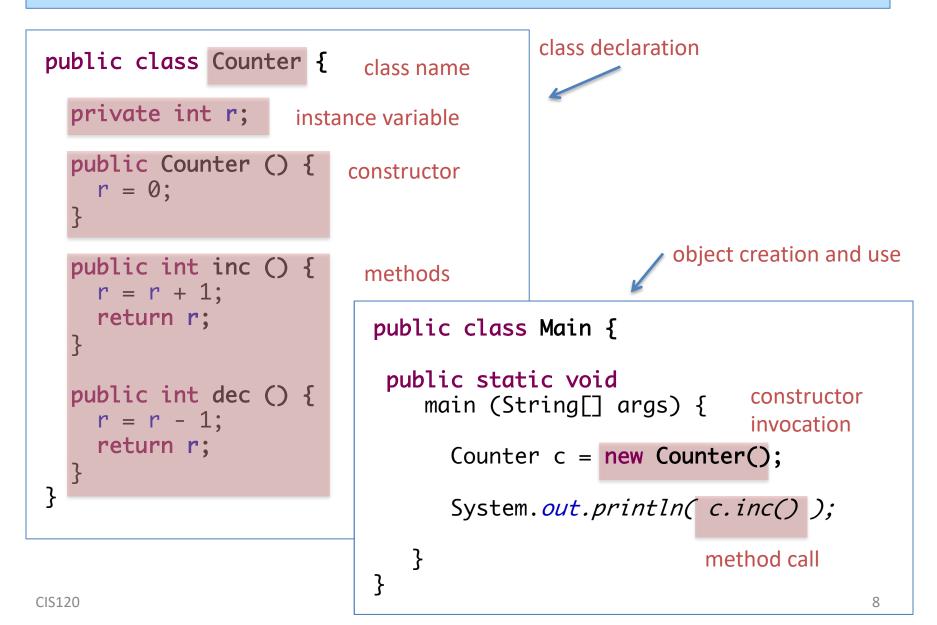
The OO Style

- Core ideas:
 - Objects: state encapsulated with operations
 - Dynamic dispatch: "receiver" of method call determines behavior
 - Classes: "templates" for object creation
 - Subtyping: grouping object types by common functionality
 - Inheritance: creating new classes from existing ones

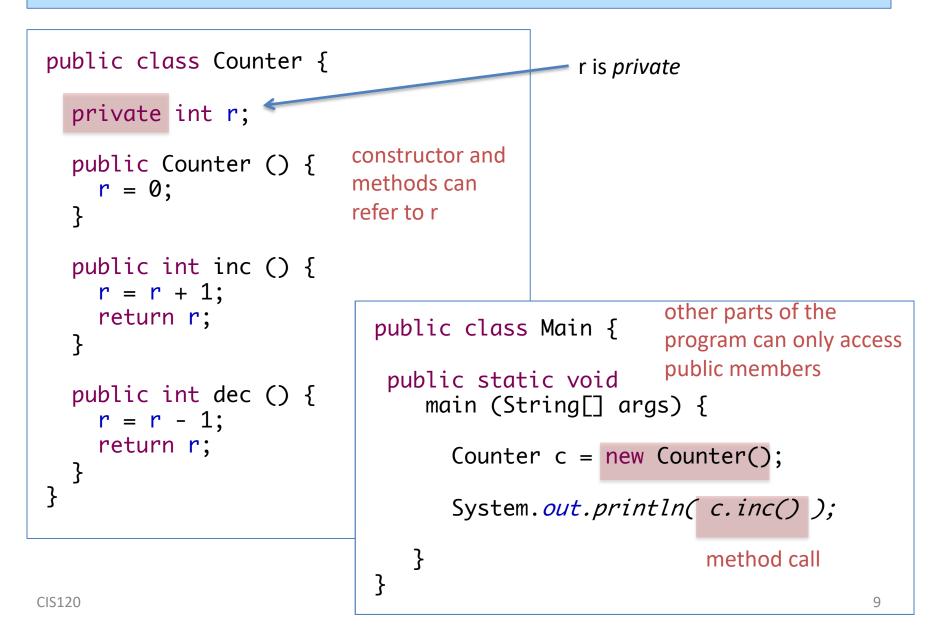
OO terminology

- Object: a structured collection of encapsulated fields (aka instance variables) and methods
- *Class*: a template for creating objects
- The class of an object specifies...
 - the types and initial values of its local state (*fields*)
 - the set of operations that can be performed on the object (*methods*)
 - one or more *constructors*: create new objects by (1) allocating heap space, and (2) running code to initialize the object (optional, but default provided)
- Every (Java) object is an *instance* of some class
 - Instances are created by invoking a constructor with the new keyword

Objects in Java



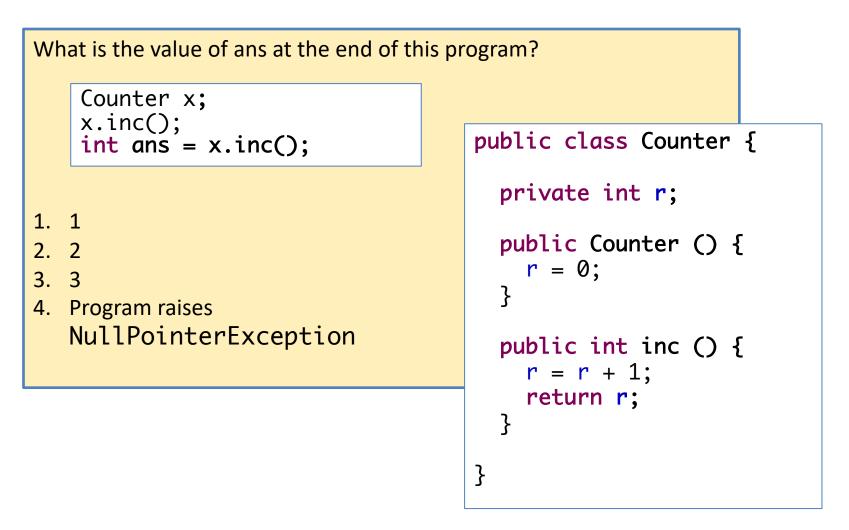
Encapsulating local state



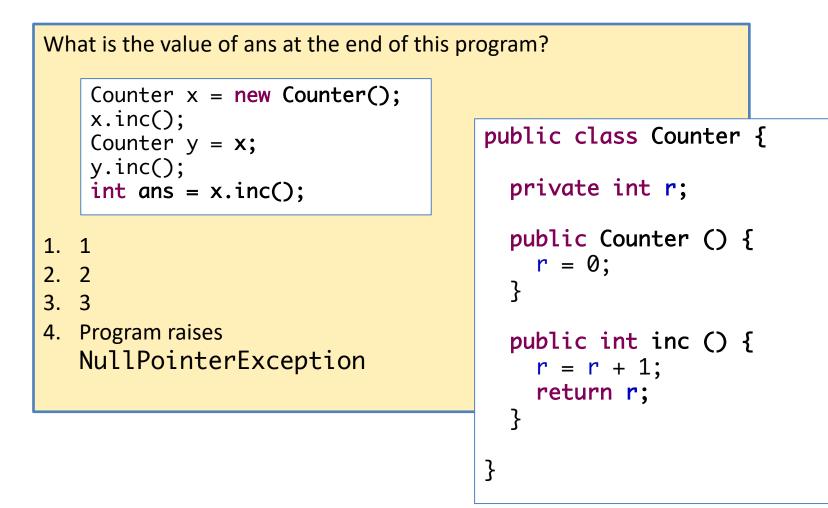
Encapsulating local state

- 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". The details are not important at this point.



Answer: Program raises NullPointerException



Answer: 3

Interfaces

Working with objects abstractly

"Objects" in OCaml vs. Java

```
(* The type of "objects" *)
type point = {
    getX : unit -> int;
    getY : unit -> int;
    move : int*int -> unit;
}
(* Create an "object" with
   hidden state: *)
type position =
  { mutable x: int;
    mutable y: int; }
let new_point () : point =
  let r = \{x = 0; y=0\} in {
    getX = (fun () -> r.x);
   getY = (fun () -> r.y);
   move = (fun (dx, dy) ->
            r.x < -r.x + dx;
            r.y <- r.y + dy)
}
         Type is separate
```

from the implementation

```
public class Point {
   private int x;
   private int y;
   public Point () {
     \mathbf{X} = 0:
     y = 0;
   public int getX () {
      return x;
   public int getY () {
      return y;
   }
   public void move
            (int dx, int dy) {
      \mathbf{x} = \mathbf{x} + \mathbf{d}\mathbf{x};
      y = y + dy;
}
```

Class specifies both type and implementation of object values

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

method bodies!

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

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

public class Point implements Displaceable { private int x, y; public Point(int x0, int y0) { $\mathbf{x} = \mathbf{x}\mathbf{0};$ interfaces implemented $\mathbf{y} = \mathbf{y}\mathbf{0};$ } public int getX() { return x; } public int getY() { return y; } methods public void move(int dx, int dy) { required to $\mathbf{x} = \mathbf{x} + d\mathbf{x};$ satisfy contract y = y + dy;

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
```

Another implementation

```
class ColoredPoint implements Displaceable {
  private Point p;
  private Color c;
  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);

$$\begin{array}{ll} \dots & & \\ \dots & d.getX() & \dots & \Rightarrow 0 \\ \dots & d.getY() & \dots & \Rightarrow 3 \end{array}$$

Using interface types

- Interface variables can refer *dynamically*, i.e. during execution, to objects of any class implementing the interface
- Point, Circle, and ColoredPoint are all *subtypes* of Displaceable

```
Displaceable d0, d1, d2;
d0 = new Point(1, 2);
d1 = new Circle(new Point(2,3), 1);
d2 = new ColoredPoint(-1,1, red);
d0.move(-2,0);
d1.move(-2,0);
                                           The class that created the
d2.move(-2,0);
                                           object value determines
                                           which move code is executed:
... d0.getX() ...
                                           dynamic dispatch
                        ⇒ –1
... d1.getX() ...
                       \Rightarrow 0
                       \Rightarrow -3
... d2.getX() ...
```

Abstraction

 The 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.

```
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 ...but there can be multiple valid points of view

- Example: Geometric objects
 - All can move (all 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
type shape =
   | Point of ...
   | Circle of ...
let draw_shape (s:shape) =
   begin match s with
   | Point ... -> ...
   | Circle ... -> ...
   end
```

Java interface Shape { public void draw();

```
class Point implements Shape {
```

```
public void draw() {
```

```
}
class Circle implements Shape {
    ...
    public void draw() {
```

Recap

- Object: A collection of related *fields* (or *instance variables*) and *methods* that operate on those fields
- **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 and Fields

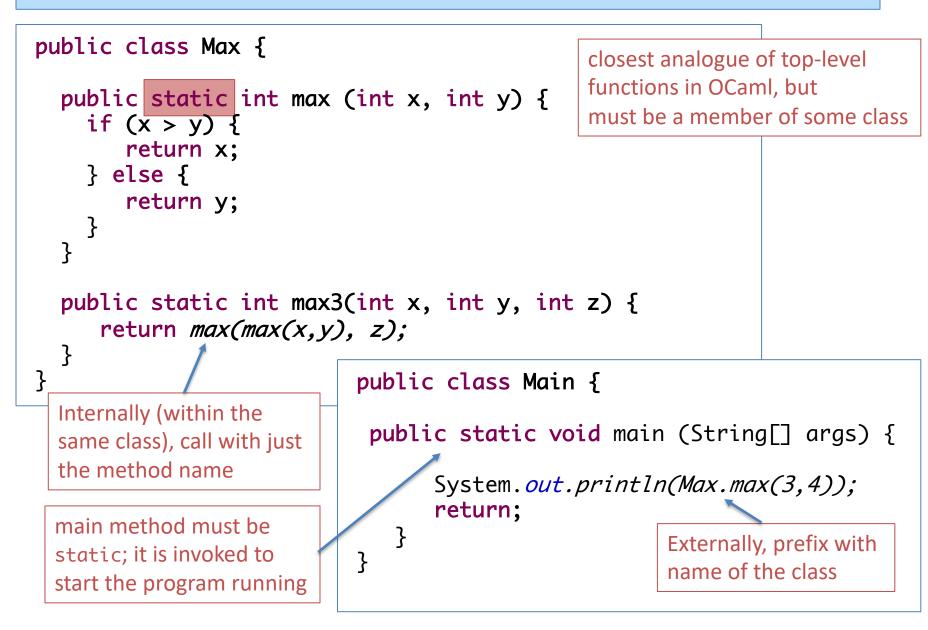
functions and global state

Java Main Entry Point

```
class MainClass {
    public static void main (String[] args) {
        ...
        ...
        }
}
```

- 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



mantra

Static == Decided at *Compile* Time Dynamic == Decided at *Run* Time

Static vs. Dynamic Methods

- Static Methods are *independent* of object values
 - Similar to OCaml functions
 - Cannot refer to the local state of objects (fields or normal methods)
- Use static methods for:
 - Non-OO programming
 - Programming with primitive types: Math.sin(60), Integer.toString(3), Boolean.valueOf("true")
 - "public static void main"
- "Normal" methods are *dynamic*
 - Need access to the local state of the particular object on which they are invoked
 - We only know at *runtime* which method will get called

```
void moveTwice (Displaceable o) {
    o.move (1,1); o.move(1,1);
}
```

Method call examples

• Calling a (dynamic) method of an object (o) that returns a number:

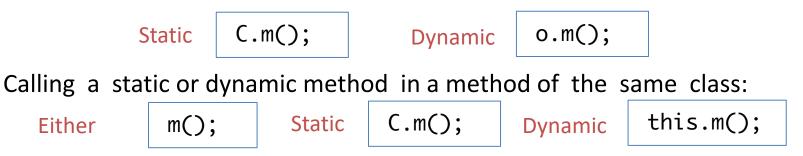
x = 0.m() + 5;

• Calling a static method of a class (C) that returns a number:

x = C.m() + 5;

• Calling a method that returns void:

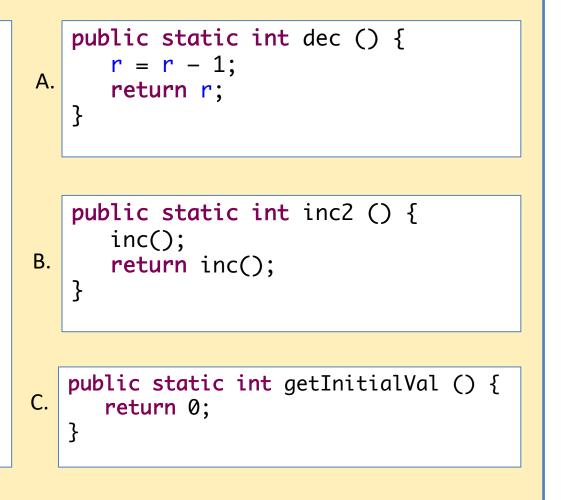
٠



• Calling (dynamic) methods that return objects:

x = o.m().n(); x = o.m().n().x().y().z().a().b().c().d().e(); Which static method can we add to this class?

```
public class Counter {
  private int r;
  public Counter () {
    \mathbf{r} = 0;
  }
  public int inc () {
    r = r + 1;
    return r;
  }
  // A,B, or C here ?
}
```



Answer: C

Static vs. Dynamic Class Members

```
public class FancyCounter {
  private int c = 0;
  private static int total = 0;
  public int inc () {
    c += 1;
    total += 1;
    return c;
 }
  public static int getTotal () {
    return total;
  }
}
                FancyCounter c1 = new FancyCounter();
                FancyCounter c2 = new FancyCounter();
                int v1 = c1.inc();
                int v^2 = c^2.inc();
                int v3 = c1.getTotal();
                System.out.println(v1 + " " + v2 + " " + v3);
```

Static Class Members

- Static methods can depend *only* on other static things
 - Static fields and methods, from the same or other classes
- Static methods *can* create *new* objects and use them
 - This is typically how main works
- public static fields are the "global" state of the program
 - Mutable global state should generally be avoided
 - Immutable global fields are useful: for constants like pi

public static final double PI = 3.14159265359793238462643383279;