Java: Objects, Interfaces, Static Members

Chapters 19 & 20
Announcements

• Java Bootcamp
  – *Tonight!! Towne 100, 6-8 pm*

• HW06: Pennstagram
  – Available very soon
  – Due: Tuesday, March 20 at 11:59:59pm
  – Java programming
  – For now, we’ll stay in Codio

• Midterm Course Survey
  – Look for a piazza post.
A Brief Word on Equality
Equality

• like OCaml, Java has two ways of testing reference types for equality:
  – “pointer equality”
    o1 == o2
  – “deep equality”
    o1.equals(o2)

• Normally, you should use == to compare primitive types and “.equals” to compare objects
Strings

• **String** is a *built in* Java class (i.e. not a primitive type)
• Strings are sequences of (unicode) characters
  "" "Java" "3 Stooges" "富士山"
• + means String concatenation (overloaded)
  "3" + " " + "Stooges" ⇒ "3 Stooges"
• Text in a String is immutable (like OCaml)
  – but variables that store strings are not
  – String x = "OCaml";
  – String y = x;
  – Can't do anything to x so that y changes
• The `.equals` method returns true when two strings contain the same sequence of characters
What is the value of ans at the end of this program?

```java
String x = "CIS 120";
String z = "CIS 120";
boolean ans = x.equals(z);
```

1. true
2. false
3. NullPointerException

Answer: true
This is the preferred method of comparing strings!
What is the value of ans at the end of this program?

```java
String x1 = "CIS ";
String x2 = "120";
String x = x1 + x2;
String z = "CIS 120";
boolean ans = (x == z);
```

1. true
2. false
3. NullPointerException

Answer: false
Even though x and z both contain the characters “CIS 120”, they are stored in two different locations in the heap.
What is the value of ans at the end of this program?

String x = "CIS 120";
String z = "CIS 120";
boolean ans = (x == z);

1. true
2. false
3. NullPointerException

Answer: true(!)
Why? Since strings are immutable, two identical strings that are known when the program is compiled can be aliased by the compiler (to save space).
Moral

Always use `s1.equals(s2)` to compare strings!

Compare strings with respect to their content, not where they happen to be allocated in memory...
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 programming**

OCaml (part we've seen)

- Explicitly create objects using a record of higher order functions and hidden state

- Flexibility through *composition*: objects can only implement one interface

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

Java (and C, C++, C#)

- Primitive notion of object creation (classes, with fields, methods and constructors)

- Flexibility through *extension*: *Subtyping* allows related objects to share a common interface

```java
class Button extends Widget {
  /* Button is a subtype of Widget */
}
```
**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
public class Counter {
    private int r;
    public Counter () {
        r = 0;
    }
    public int inc () {
        r = r + 1;
        return r;
    }
    public int dec () {
        r = r - 1;
        return r;
    }
}

public class Main {
    public static void main (String[] args) {
        Counter c = new Counter();
        System.out.println(c.inc());
    }
}
Encapsulating local state

```java
public class Counter {
    private int r;

    public Counter () {
        r = 0;
    }

    public int inc () {
        r = r + 1;
        return r;
    }

    public int dec () {
        r = r - 1;
        return r;
    }
}
```

The variable `r` is `private`, which means it is only accessible within the `Counter` class. The `inc()` and `dec()` methods can refer to `r` because they are defined within the same class. However, other parts of the program can only access the public members of the `Counter` class.

```java
public class Main {
    public static void main (String[] args) {
        Counter c = new Counter();
        System.out.println( c.inc() );
    }
}
```
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.)
What is the value of ans at the end of this program?

```java
public class Counter {
    private int r;
    public Counter () {
        r = 0;
    }
    public int inc () {
        r = r + 1;
        return r;
    }
}
```

Counter x;
x.inc();
int ans = x.inc();

1. 1
2. 2
3. 3
4. Program raises NullPointerException

Answer: Program raises NullPointerException
What is the value of ans at the end of this program?

Counter x = new Counter();
x.inc();
Counter y = x;
y.inc();
int ans = x.inc();

1. 1
2. 2
3. 3
4. Program raises
   NullPointerException

public class Counter {
    private int r;
    public Counter () {
        r = 0;
    }
    public int inc () {
        r = r + 1;
        return r;
    }
}

Answer: 3
“Objects” in OCaml vs. Java

(* The type of “objects” *)

```ocaml
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 {
        let r = r in {
            getX = (fun () -> r.x);
            getY = (fun () -> r.y);
            move = (fun (dx,dy) -> {
                r.x <- r.x + dx;
                r.y <- r.y + dy;
            });
        };
    };
}
```

(public class) Point {

    private int x;
    private int y;

    public Point () {
        x = 0;
        y = 0;
    }

    public intgetX () {
        return x;
    }

    public intgetY () {
        return y;
    }

    public void move (int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}

Type is separate from the implementation

Class specifies both type and implementation of object values
Interfaces

Working with objects abstractly
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

```java
public interface Displaceable {
    public int getX();
    public int getY();
    public void move(int dx, int dy);
}
```

No fields, no constructors, no method bodies!
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 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 local state can satisfy the same interface

Delegation: move the circle by moving the 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; }
}
Interfaces are types

- Can declare variables of interface type
  
  ```java
  void m(Displaceable d) { ... }
  ```

- Can call m with any Displaceable argument...
  ```java
  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
  ```java
  d.move(-1,1);
  ...
  ... d.getX() ... ⇒ 0
  ... d.getY() ... ⇒ 3
  ```
Using interface types

• Interface variables can refer \textit{dynamically}, i.e. during execution, to objects of any class implementing the interface

• Point, Circle, and ColoredPoint are all \textit{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);
d2.move(-2,0);
...
... d0.getX() ... ⇒ -1
... d1.getX() ... ⇒ 0
... d2.getX() ... ⇒ -3

The class that created the object value determines which move code is executed: \textit{dynamic dispatch}
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.

```java
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);
    }
}
```
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”)