Programming Languages and Techniques (CIS120)

Lecture 22
March 14th, 2016

Object Oriented Programming in Java
Announcements

• Java Bootcamp tonight
  – Monday, March 14 from 6-8pm in Levine 101 (Wu & Chen)

• Midterm 2
  – March 22nd, 6:15-8:15PM, location TBA
  – Make-up exam, Wed March 23rd, 9-11AM
  – Sign up for make-up exam on course website by March 20th

• HW5: Java Programming
  – Will be available soon
  – Due: Tuesday, March 29th at 11:59pm (after the exam)
Midterm 2

• Focus of exam: Programming in OCaml with higher-order functions and mutable state
• Homeworks 4 (queues) and 5 (GUI)
• Through Wednesday's lecture
  – everything from first exam (1-10)
  – mutable & immutable records (11-13)
  – ASM (14)
  – options (15), queues, deques and tail recursion (16)
  – object encoding, local state (17) and reactive programming (18)
  – comparisons between OCaml and Java (19 & 20)

• Practice exams on website
  – Old exams were held later in the course (after HW 6)
  – Not covered this time: writing Java code, Java arrays, Java subtyping and dynamic dispatch, Java ASM
Object-Oriented Programming in Java
• **Object**: a structured collection of *fields* (aka local state or *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*: code that is executed when the object is created (optional)

• Every (Java) object is an *instance* of some class
"Objects" in OCaml

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

(* The type of counter objects *)

```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)
  }
```
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() );
}
}
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

CIS120
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

(There are 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?

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

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

1. 1
2. 2
3. 3
4. NullPointerException

Answer: 2
What is the value of ans at the end of this program?

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. NullPointerException

Answer: NPE
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. NullPointerException

Answer: 3
OO comparison

OCaml

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

• Flexibility through *composition*: objects can only implement one interface (i.e. 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 objects to implement multiple interfaces (i.e. button <: widget)
Interfaces

Working with objects abstractly
"Objects" in OCaml vs. Java

(* The type of point "objects" *)

```ocaml
type point = {
  getX : unit -> int;
  getY : unit -> int;
  move : int*int -> unit;
}

(* Create an "object" with hidden state: *)

```public class Point {

```private int x;
private int y;

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

public int getX () {
  return x;
}

public int getY () {
  return y;
}

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

Type is separate from the implementation

Class specifies both type and implementation of object values
Interfaces

• Give a type for an object based on what it does, not on how it was constructed
• Describes 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!
Implementing the interface

• A class that implements an interface provides appropriate definitions for the methods specified in the interface
• That class fulfills the contract implicit in the interface

```java
public class Point implements Displaceable {
    private int x, y;
    public Point(int x0, int y0) {
        x = x0;
        y = y0;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```
Another implementation

```java
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
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 method 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() ... ⇒ 0.0
... d.getY() ... ⇒ 3.0
... d.getColor() ... Doesn't type check
```
Using interface types

• Interface variables can refer (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);
d2.move(-2,0);
...
... d0.getX() ... ⇒ -1.0
... d1.getX() ... ⇒ 0.0
... d2.getX() ... ⇒ -3.0
```
Class that created the object value determines what move function is called.
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);
    }
}
```
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 have a color
  - Circles and Points don’t implement Colored
  - ColoredPoints do

```java
public interface Colored {
    public Color getColor();
}
```
public class ColoredPoint implements Displaceable, Colored {

    Point center;
    private Color color;
    public Color getColor() {
        return color;
    }

    ...
}

Recap

- **Object**: A collection of related *fields* (or *instance variables*)
- **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”)


Java Core Language

differences between OCaml and Java
Expressions vs. Statements

• OCaml is an *expression language*
  – Every program phrase is an expression (and returns a value)
  – The special value `()` of type `unit` is used as the result of expressions that are evaluated only for their side effects
  – Semicolon is an *operator* that combines two expressions (where the left-hand one returns type `unit`)

• Java is a *statement language*
  – Two sorts of program phrases: expressions (which compute values) and statements (which don’t)
  – Statements are *terminated* by semicolons
  – Any expression can be used as a statement (but not vice-versa)
As in OCaml, every Java *expression* has a type

The type describes the value that an expression computes

<table>
<thead>
<tr>
<th>Expression form</th>
<th>Example</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable reference</td>
<td>x</td>
<td>Declared type of variable</td>
</tr>
<tr>
<td>Object creation</td>
<td>new Counter ()</td>
<td>Class of the object</td>
</tr>
<tr>
<td>Method call</td>
<td>c.inc()</td>
<td>Return type of method</td>
</tr>
<tr>
<td>Equality test</td>
<td>x == y</td>
<td>boolean</td>
</tr>
<tr>
<td>Assignment</td>
<td>x = 5</td>
<td><em>don’t use as an expression!!</em></td>
</tr>
</tbody>
</table>
## Type System Organization

<table>
<thead>
<tr>
<th>OCaml</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>primitive types</strong>&lt;br&gt;(values stored “directly” in the stack)</td>
<td>int, float, char, bool, ...</td>
</tr>
<tr>
<td>structured types&lt;br&gt;(a.k.a. reference types — values stored in the heap)</td>
<td>tuples, datatypes, records, functions, arrays&lt;br&gt;(objects encoded as records of functions)</td>
</tr>
<tr>
<td>generics</td>
<td>‘a list</td>
</tr>
<tr>
<td>abstract types</td>
<td>module types (signatures)</td>
</tr>
</tbody>
</table>
# Arithmetic & Logical Operators

<table>
<thead>
<tr>
<th>OCaml</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=, ==</code></td>
<td><code>==</code></td>
</tr>
<tr>
<td><code>&lt;&gt;</code>, <code>!=</code></td>
<td><code>!=</code></td>
</tr>
<tr>
<td><code>&gt;</code>, <code>&gt;=</code>, <code>&lt;</code>, <code>&lt;=</code></td>
<td><code>&gt;</code>, <code>&gt;=</code>, <code>&lt;</code>, <code>&lt;=</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td><code>+</code></td>
</tr>
<tr>
<td><code>-</code></td>
<td><code>-</code></td>
</tr>
<tr>
<td><code>*</code></td>
<td><code>*</code></td>
</tr>
<tr>
<td><code>/</code></td>
<td><code>/</code></td>
</tr>
<tr>
<td><code>mod</code></td>
<td><code>%</code></td>
</tr>
<tr>
<td><code>not</code></td>
<td><code>!</code></td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td><code>&amp;&amp;</code></td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
</tbody>
</table>
• The meaning of an operator is determined by the types of the values it operates on
  – Integer division
    \[ \frac{4}{3} \Rightarrow 1 \]
  – Floating point division
    \[ \frac{4.0}{3.0} \Rightarrow 1.3333333333333333 \]
  – Automatic conversion
    \[ \frac{4}{3.0} \Rightarrow 1.3333333333333333 \]

• Overloading is a general mechanism in Java
  – we’ll see more of it later
• 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

---

**Equality**

- 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
- Strings are sequences of 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.