Programming Languages and Techniques (CIS120)

Lecture 21
March 1, 2013

Transition to Java II
Interfaces and Declarative Programming
Announcements

• HW06 Due Today at 11:59:59pm

• HW07 will be posted over break, due Monday, March 18th.

• Have a great break!
Interfaces

Thinking about objects abstractly
“Objects” in OCaml vs. Java

(* The type of counter “objects” *)

```ocaml
type counter = {
  inc : unit -> int;
  dec : unit -> int;
}
```

(* Create a counter "object" with hidden state: *)

```ocaml
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;
  }
})

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
And another...

class ColorPoint implements Displaceable {
    private Point p;
    private Color c;
    ColorPoint (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 as types

• Can declare variables of interface type
  
  $$\text{void } m(\text{Displaceable } d) \{ \ldots \}$$

• Can provide any implementation for the variable
  
  $$\text{obj.m(new ColorPoint(1,2,Color.Black))}$$

• ... but can only operate on the object according to the interface
  
  $$\text{d.move(-1,1);}$$
  
  $$\ldots$$
  
  $$\ldots \text{d.getX()} \ldots \Rightarrow 0.0$$
  
  $$\ldots \text{d.getY()} \ldots \Rightarrow 3.0$$
Using interface types

- Interface variables can refer (during execution) to objects of any class implementing the interface.
- Point, Circle, and ColorPoint are all subtypes of Displaceable.

```java
Displaceable d0, d1, d2;
d0 = new Point(1, 2);
d1 = new Circle(new Point(2,3), 1);
d2 = new ColorPoint(-1,1, red);
d0.move(-2,0);
d1.move(-2,0);
d2.move(-2,0);
...
... d0.getX() ... \Rightarrow -1.0
... d1.getX() ... \Rightarrow 0.0
... d2.getX() ... \Rightarrow -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 “Displaceables”, 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);
    }
}

CIS120 / Spring 2013 class
Variants

**OCaml**

```ocaml
type shape =
  | Point of ...
  | Circle of ...

let draw_shape (s:shape) =
  begin match s with
  | Point ... ->
  | Circle ... ->
  end
```

**Java**

```java
interface Shape {
    void draw();
}

class Point implements Shape {
    ...

    ...
}

class Circle implements Shape {
    ...

    ...
}
```
Multiple interfaces

• An interface represents a point of view
  ...but there are multiple 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
  – ColorPoints do

```java
public interface Colored {
    public Color getColor();
}
```
public class ColoredPoint implements Displaceable, Colored {
    private Point center;
    private Color color;
    ...
    public Color getColor() {
        return color;
    }
}
• **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 when the object is first created

• **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
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 (that compute values) and statements (that don’t)
- Statements are *terminated* by semicolons
- Any expression can be used as a statement (but not vice-versa)
Types

- 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>REDACTED, don’t use as an expression!</td>
</tr>
</tbody>
</table>
## Type System Organization

<table>
<thead>
<tr>
<th></th>
<th>OCaml</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>primitive types</strong></td>
<td>int, float, char, bool, ...</td>
<td>int, float, double, char, boolean, ...</td>
</tr>
<tr>
<td>(values stored “directly” in the stack)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tuples, datatypes, records, functions, arrays</td>
<td>objects, arrays</td>
</tr>
<tr>
<td></td>
<td><em>(objects encoded as records of functions)</em></td>
<td><em>(records, tuples, datatypes, strings, first-class functions are a special case of objects)</em></td>
</tr>
<tr>
<td><strong>structured types</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a.k.a. <em>reference types</em> — values stored in the heap)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>generics</strong></td>
<td>‘a list</td>
<td>List&lt;A&gt;</td>
</tr>
<tr>
<td><strong>abstract types</strong></td>
<td>module types (signatures)</td>
<td>interfaces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>public/private modifiers</td>
</tr>
</tbody>
</table>
Java’s Primitive Types

<table>
<thead>
<tr>
<th>Java Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>standard integers</td>
</tr>
<tr>
<td>byte, short, long</td>
<td>other flavors of integers</td>
</tr>
<tr>
<td>char</td>
<td>characters</td>
</tr>
<tr>
<td>float, double</td>
<td>floating-point numbers</td>
</tr>
<tr>
<td>boolean</td>
<td>true and false</td>
</tr>
</tbody>
</table>
# Arithmetic & Logical Operators

<table>
<thead>
<tr>
<th>OCaml</th>
<th>Java</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==, ==</td>
<td>==</td>
<td>equality test</td>
</tr>
<tr>
<td>&lt;&gt;, !=</td>
<td>!=</td>
<td>inequality</td>
</tr>
<tr>
<td>&gt;, &gt;=, &lt;, &lt;=</td>
<td>&gt;=, &gt;, &lt;, &lt;=</td>
<td>comparisons</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>addition (and string concatenation)</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>subtraction (and unary minus)</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>mod</td>
<td>%</td>
<td>remainder (modulus)</td>
</tr>
<tr>
<td>not</td>
<td>!</td>
<td>logical “not”</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>&amp;&amp;</td>
<td>logical “and” (short-circuiting)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
New: Operator Overloading

• The meaning of an operator is determined by the *types* of the values it operates on
  – Integer division
    \[ 4/3 \Rightarrow 1 \]
  – Floating point division
    \[ 4.0/3.0 \Rightarrow 1.3333333333333333 \]
  – Automatic conversion
    \[ 4/3.0 \Rightarrow 1.3333333333333333 \]

• Overloading is a much more general mechanism in Java
  – we’ll see more of it later
  – it should be used with care
Equality

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

  every object provides an “equals” method that “does the right thing”
  depending on the object

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

• = is the assignment operator in Java
  – behaves like <- in OCaml
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

- **Always use** `.equals` **to compare Strings**
Pragmatics: Java identifiers

• Variable, class, interface, and method names are identifiers
• Alphanumeric characters or _ starting with a letter or _
  – size
  – myName
  – MILES_PER_GALLON
  – A1
  – the_end
• Interpretation depends on context: variables and classes can have the same name
### Naming conventions

<table>
<thead>
<tr>
<th>kind</th>
<th>part-of-speech</th>
<th>identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>noun</td>
<td>RacingCar</td>
</tr>
<tr>
<td>variable</td>
<td>noun</td>
<td>initialSpeed</td>
</tr>
<tr>
<td>constant</td>
<td>noun</td>
<td>MAXIMUM_SPEED</td>
</tr>
<tr>
<td>method</td>
<td>verb</td>
<td>shiftGear</td>
</tr>
</tbody>
</table>
Beware: Identifier abuse

Class, instance variable, constructor, and method with the same name...

```java
public class Turtle {
    private Turtle Turtle;
    public Turtle() {
    }

    public Turtle Turtle (Turtle Turtle) {
        return Turtle;
    }
}
```
Static Methods

aka “functions”
Static methods: by example

```java
public class Max {

    public static int max (int x, int y) {
        if (x > y) {
            return x;
        } else {
            return y;
        }
    }

    public static int max3(int x, int y, int z) {
        return max( max (x,y), z);
    }
}

public class Main {

    public static void main (String[] args) {
        System.out.println(Max.max(3,4));
        return;
    }
}
```

if then and else cases must be statements

return statement terminates a method call

Must be defined in a class, but closest analogue to functions in OCaml

Externally, call with name of the class
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 the object (fields or dynamic methods)
• Use static methods for:
  – Non-OO programming (i.e. declarative programming)
  – Programming with primitive types: Math.sin(60), Integer.toString(3), Boolean.valueOf("true")
  – “public static void main”
• Basic design guideline: put static methods in classes by themselves
• “Normal” methods are dynamic
  – Need access to the local state of the object that invokes them
  – We only know at runtime which method will get called

```java
void moveTwice (Displaceable o) {
    o.move (1,1); o.move(1,1); 
}
```
Method call examples

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

\[
x = o.m() + 5;
\]

• Calling a static method of another object that returns a number:

\[
x = C.m() + 5;
\]

• Calling a method of another class that returns void:

<table>
<thead>
<tr>
<th>Static</th>
<th>C.m();</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>o.m();</td>
</tr>
</tbody>
</table>

• Calling a static or dynamic method of the same class:

\[
m(); x = m() + 5;
\]

• Calling (dynamic) methods that return objects:

\[
x = o.m().n();
\]
\[
x = o.m().n().x().y().z().a().b().c().d().e();
\]
Datatypes and lists in Java

What is the Java analogue of OCaml (immutable) lists...

```
type string_list = Nil | Cons of string * string_list
```

...and recursive/iterative functions over lists?

```
let rec number_of_songs (pl : string_list) : int =
  begin
    match pl with
    | [] -> 0
    | ( song :: rest ) -> 1 + number_of_songs rest
  end
```
Datatypes and Immutable Lists in Java

```java
interface StringList {
    public boolean isNil();
    public String hd();
    public StringList tl();
}
```

```java
class Nil implements StringList {
    public boolean isNil() {
        return true;
    }
    public String hd() {
        return null;
    }
    public StringList tl() {
        return null;
    }
}
```

```java
class Cons implements StringList {
    private final String head;
    private final StringList tail;
    public Cons (String h, StringList t) {
        head = h; tail = t;
    }
    public boolean isNil() {
        return false;
    }
    public String hd() {
        return head;
    }
    public StringList tl() {
        return tail;
    }
}
```

only call these if isNil() == false
Creating lists

OCaml

```ocaml
let x = Cons "Gagnam Style" (Cons "Dynamite" Nil)
```

Java

```java
StringList x = new Cons ("Gagnam Style", new Cons ("Dynamite", new Nil()))
```

- Both lists are immutable:
  - In Java, can’t say `x.tail = new Nil()`
  - Because tail defined as `final`

- General pattern for datatypes:
  - Define an interface for the datatype type
  - For each data constructor, add a class
  - Add accessors for data (clunky, we'll see better ways to do this later)
### Using lists

<table>
<thead>
<tr>
<th>OCaml</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>let rec number_of_songs (pl : string_list) : int =</td>
<td></td>
</tr>
<tr>
<td>begin match pl with</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[] -&gt; 0</td>
</tr>
<tr>
<td></td>
<td>( song :: rest ) -&gt; 1 + number_of_songs rest</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>public static int numberOfSongs (StringList pl) {</td>
<td></td>
</tr>
<tr>
<td>if (pl.isNil()) {</td>
<td></td>
</tr>
<tr>
<td>return 0;</td>
<td></td>
</tr>
<tr>
<td>} else {</td>
<td></td>
</tr>
<tr>
<td>return 1 + numberOfSongs (pl.tl());</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
List Iteration

**OCaml (Better)**

```ocaml
let number_of_songs (pl : string_list) : int =
  let rec loop (pl:string_list) (n:int) : int =
    begin match pl with
    | [] -> n
    | ( song :: rest ) -> number_of_songs rest (1 + n)
    end
  in loop pl
```

**Java (Better)**

```java
public static int numberOfSongs (StringList pl) {
    int n = 0;
    StringList curr = pl;
    while (!curr.isNil()) {
        n = 1 + n;
        curr = curr.tl();
    }
    return n;
}
```

no tail recursion in Java  Using mutable local variables for value-oriented programming