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” *)
type counter = {
    inc  : unit -> int;
    dec  : unit -> int;
}

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

Type is separate from the implementation

```
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;
  }
}
```

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

```
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

```
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;
  }
}
```

interfaces implemented

methods required to satisfy contract

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 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; }
}
```

Flexibility:
Classes may contain more methods than the interface

Interfaces as types

- Can declare variables of interface type

```
void m(Displaceable d) { … }
```

- Can provide any implementation for the variable

```
obj.m(new ColorPoint(1,2,Color.Black));
```

- … but can only operate on the object according to the interface

```
d.move(-1,1);
…
… d.getX() …          ⇒ 0.0
… d.getY() …          ⇒ 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

```
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() …      ⇒ -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 "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);
  }
}
```

Variants

OCaml

```
type shape =
    | Point of …
    | Circle of …

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

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 that have a color
  - Circles and Points don't
  - ColorPoints do

```
public interface Colored {
      public Color getColor();
}
```

ColoredPoints

```
public class ColoredPoint
 implements Displaceable, Colored {
  private 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 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

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 (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

Expression form	**Example**	**Type**
Variable reference	x	Declared type of variable
Object creation	new Counter ()	Class of the object
Method call	c.inc()	Return type of method
Equality test	x == y	boolean
Assignment	x = 5	REDACTED, don't use as an expression!

Type System Organization

	OCaml	**Java**
primitive types (values stored "directly" in the stack)	int, float, char, bool, …	int, float, double, char, boolean, …
structured types (a.k.a. *reference types* — values stored in the heap)	tuples, datatypes, records, functions, arrays (*objects encoded as records of functions*)	objects, arrays (*records, tuples, datatypes, strings, first-class functions are a special case of objects*)
generics	'a list	List<A>
abstract types	module types (signatures)	interfaces public/private modifiers

Java's Primitive Types

`int`	standard integers
`byte, short, long`	other flavors of integers
`char`	characters
`float, double`	floating-point numbers
`boolean`	`true` and `false`

Arithmetic & Logical Operators

OCaml	Java	
=, ==	==	equality test
<>, !=	!=	inequality
>, >=, <, <=	>, >=, <, <=	comparisons
+	+	addition (and string concatenation)
-	-	subtraction (and unary minus)
*	*	multiplication
/	/	division
mod	%	remainder (modulus)
not	!	logical “not”
&&	&&	logical “and” (short-circuiting)
||	||	logical “or” (short-circuiting)

New: Operator Overloading

- The meaning of an operator is determined by the *types* of the values it operates on
  - Integer division

    `4/3` ⇒ `1`
  - Floating point division

    `4.0/3.0` ⇒ `1.3333333333333333`
  - Automatic conversion

    `4/3.0` ⇒ `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

kind	*part-of-speech*	*identifier*
class	noun	`RacingCar`
variable	noun	`initialSpeed`
constant	noun	`MAXIMUM_SPEED`
method	verb	`shiftGear`

Beware: Identifier abuse

Class, instance variable, constructor, and method with the *same name*…

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

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

Static Methods

aka “functions”

Static methods: by example

```
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);
  }
}
```

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

Internally, call with just the method name

if then and else cases must be statements

return statement terminates a method call

```
public class Main {

 public static void
    main (String[] args) {

      System.out.println(Max.max(3,4));
      return;
   }
}
```

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

```
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:

Static

```
C.m();
```

Dynamic

```
o.m();
```

- 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

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

only call these if isNil() == false

```
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;
   }
}
```

```
class Nil implements StringList {

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

Creating lists

OCaml

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

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

OCaml

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

Java

```
public static int numberOfSongs (StringList pl) {
   if (pl.isNil()) {
      return 0;
   } else {
      return 1 + numberOfSongs (pl.tl());
   }
}
```

List Iteration

OCaml (Better)

```
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)

```
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