Programming Languages and Techniques (CIS120)

Lecture 20

February 27, 2013

Transition to Java

Objects, classes, and interfaces

Announcements

- HW06 Due Friday, March 1st at 11:59:59pm

- Lab this week: setting up eclipse for Java!

- For the Java portion of the course, we recommend creating a *new* Eclipse workspace
  - So that you don't have to switch settings between OCaml/Java when you move back and forth

Upcoming HW due dates

	Tentative date	
HW 07	Monday, March 18th	released over break
HW 08	Monday, March 25th	
Exam 2	Friday, March 29th	
HW 09	Tuesday, April 9th	
HW 10	Tuesday, April 23rd	last day of classes

Looking Back…

Course Overview

- Declarative (Functional) programming
  - *persistent* data structures
  - *recursion* is main control structure
  - heavy use of functions as data
- Imperative programming
  - *mutable* data structures (that can be modified "in place")
  - *iteration* is main control structure
- Object-oriented (and reactive) programming
  - mutable data structures / iteration
  - heavy use of functions (objects) as data
  - pervasive "abstraction by default"

OCaml: What's Left

OCaml is not a very large language — we've actually seen most of its important features. But we've omitted a few…

- Module system
  - One of OCaml's most interesting features is its excellent support for large-scale programming
  - We saw just the tip of the iceberg: *structures* and *signatures*
  - Key feature: *functors* (functions from structures to structures)
- Object system
  - OCaml actually includes a powerful system of classes and objects
  - We left them out to avoid confusion with Java's way of doing things
- Miscellaneous handy type-system features
  - e.g. "polymorphic variants" (used, for example, to support parameter passing by name instead of by position)
  - Type inference – almost all of the type annotations we've been using can be omitted.

Recap: The Functional Style

- Core ideas:
  - value-oriented programming
  - immutable (persistent / declarative) data structures
  - recursion (and iteration) over tree structured data
  - functions as data
  - generic types for flexibility (i.e. ‘a list)
  - abstract types to preserve invariants (i.e. BSTs)
- Good for:
  - simple, elegant descriptions of complex algorithms and/or data
  - parallelism, concurrency, and distribution
  - “symbol processing” programs (compilers, theorem provers, etc.)

Language Support for FP

- “Functional languages” (OCaml, Standard ML, F#, Haskell, Scheme) promote this style as a default and work hard to implement it efficiently
- “Hybrid languages” (Scala, Python) offer it as one possibility among others
- Mainstream “Object Oriented” languages (Java, C#, C++, Objective C) favor a different style by default
  - But many common OO idioms and *design patterns* have a functional flavor (e.g. the “Visitor” pattern is analogous to transform)
  - And most of them are gradually adding features (like anonymous functions) that make functional-style programming more convenient
  - Best practices discourage use of imperative state

Functional programming

OCaml

- Immutable lists primitive, tail recursion
- Datatypes and pattern matching for tree structured data
- First-class functions
- Generic types
- Abstract types through module signatures

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

- No primitive data structures, no tail recursion
- Trees must be encoded by objects
- No first-class functions.* Must encode first-class computation with objects
- Generic types
- Abstract types through public/private modifiers

*until Java 8, coming this summer

OCaml vs. Java

```ocaml
type 'a tree =
  | Empty
  | Node of ('a tree) * 'a * ('a tree)

let is_empty (t:'a tree) =
  begin match t with
    | Empty -> true
    | Node(_,_,_) -> false
  end

let t : int tree = Node(Empty,3,Empty)
let ans : bool = is_empty t
```

```java
interface Tree<A> {
  public boolean isEmpty();
}
class Empty<A> implements Tree<A> {
  public boolean isEmpty() {
     return true;
  }
}
class Node<A> implements Tree<A> {
  private final A v;
  private final Tree<A> lt;
  private final Tree<A> rt;

  Node(Tree<A> lt, A v, Tree<A> rt) {
    this.lt = lt; this.rt = rt; this.v = v;
  }

  public boolean isEmpty() {
    return false;
  }
}

class Program {
  public static void main(String[] args) {
    Tree<Integer> t =
    new Node<Integer>(new Empty<Integer>(),
     3, new Empty<Integer>());
    boolean ans = t.isEmpty();
  }
}
```

Recap: Imperative programming

- Core ideas:
  - computation as change of state over time
  - distinction between primitive and reference values
  - aliasing
  - linked data-structures and iteration control structure
  - generic types for flexibility (i.e. ‘a queue)
  - abstract types to preserve invariants (i.e. queue invariant)

- Good for:
  - numerical simulations
  - implicit coordination between components

Imperative programming

OCaml

- No null. Partiality must be made explicit with **options**.
- Code is an **expression** that has a value. Sometimes computing that value has other effects.
- References are **immutable** by default, must be explicitly declared to be mutable

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

- Null is contained in (almost) every type. Partial functions can return **null**.
- Code is a sequence of **statements** that do something, sometimes using expressions to compute values.
- References are **mutable** by default, must be explicitly declared to be constant

Recap (and coming): The OO Style

- Core ideas:
  - objects (state encapsulated with operations)
  - classes ("templates" for object creation)
  - dynamic dispatch ("receiver" of method call determines behavior)
  - subtyping (grouping object types by common functionality)
  - inheritance (creating new classes from existing ones)
- Good for:
  - GUIs!
    - and other complex software systems that include many different implementations of the same "interface" (set of operations) with different behaviors (cf. widgets)
  - Simulations
    - designs with an explicit correspondence between "objects" in the computer and things in the real world

OO programming

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 related objects to share a common interface
  (i.e. button <: widget)

Java and OCaml together

Xavier Leroy, one of the principal designers of OCaml

Guy Steele, one of the principal designers of Java

Moral: Java and OCaml are not so far apart…

Looking Forward

Today: Objects, Classes and Interfaces in Java

Friday: Declarative programming in Java

Smoothing the transition

- DON'T PANIC
- Ask questions, but don't worry about the details until you need them.
- Java resources:
  – Lecture notes and lecture slides
  – Online Java textbook (http://math.hws.edu/javanotes/) linked from "CIS 120 Resources" on course website
  – Penn Library: Electronic access to "Java in a Nutshell" (and all other O'Reilly books)
  – Piazza!

Caveats

- Some aspects of Java involve quite a bit of detail
- There is often much more to the story than presented in the lectures (and more than needed for CIS 120).
- We expect you to use various online and print resources to fill in the details (and you can ask when in doubt)
- But don't worry about details until you need them
- The best way to learn details is to use them in *solving a problem*

Objects

from OCaml to Java

"Objects" in OCaml

```
(* 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)
}
```

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

Critique of Hand-Rolled Objects

- "Roll your own objects" made from records, functions, and references are good for understanding...

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

- ...but not that good for programming
  - minor: syntax is clunky (too many parens, etc.)
  - major: OCaml's record *types* are too rigid, cannot reuse functionality

```
type reset_counter = {
    inc   : unit -> int;
    dec   : unit -> int;
    reset : unit -> unit;
}
```

Java Objects and Classes

- *Object*: a structured collection of *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*: code that is executed when the object is created (optional)
- Every Java object is an *instance* of some class
- Can (optionally) implement an *interface* that specifies it in terms of its operations

Objects in Java

class declaration

```
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 name

instance variable

constructor

methods

object creation and use

```
public class Main {

  public static void
     main (String[] args) {

       Counter c = new Counter();

       System.out.println( c.inc() );

     }
}
```

constructor invocation

method call

Creating Objects

- *Declare* a variable to hold the `Counter` object
  – Type of the object is the *name* of the class that creates it
- *Invoke* the constructor for `Counter` to create a `Counter` instance with keyword "new" and store it in the variable

```
Counter c = new Counter();
```

Constructors with Parameters

```java
public class Counter {

  private int r;

  public Counter (int r0) {
    r = r0;
  }

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

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

Constructor methods can take parameters

Constructor must have the same name as the class

object creation and use

```java
public class Main {

 public static void
    main (String[] args) {

      Counter c = new Counter(3);

      System.out.println( c.inc() );

   }
}
```

constructor invocation

Creating objects

- Every Java variable is mutable

```
Counter c = new Counter(2);
c = new Counter(4);
```

- A Java variable of *reference* type can also contains the special value “null”

```
Counter c = null;
```

☞ Single = for assignment
Double == for reference equality testing

Using objects

- At any time, a Java variable of reference type can contain either the special value “null” or a pointer into the heap
  - i.e., a Java variable of reference type "T" is like an OCaml variable of type "T option ref"
  - The dereferencing of the pointer and the check for “null” are implicitly performed every time a variable is used

```
let f (co : counter option ref) : int =
  begin match co.contents with
  | None ->
     failwith "NullPointerException"
  | Some c -> c.inc()
  end
```

```
class Foo {
  public int f (Counter c) {
    return c.inc();
  }
}
```

- If null value is used as an object (i.e. with a method call) then a NullPointerException occurs

Encapsulating local state

```
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;
  }
}
```

r is *private*

constructor and methods can refer to r

```
public class Main {

 public static void
    main (String[] args) {

      Counter c = new Counter();

      System.out.println( c.inc() );

   }
}
```

other parts of the program can only access public members

method call

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 methods public

(There are a couple of other protection levels — protected and "package protected". The details are not important at this point.)