Programming Languages and Techniques (CIS120)

Lecture 20

March 7, 2014

Transition to Java

How would you rate your familiarity with Java

```
1) Never used anything like it
2) I've used a typed OO language before
   (C#, C++, Objective C) but not Java
3) Really rusty, not sure I remember it
4) Have written ~100 line programs
   (CIS 110 / AP CS level)
5) Have written larger programs, using
   the standard libraries
6) I could teach a course on Java
```

Smoothing the transition

- DON'T PANIC
- Ask questions, but don't worry about the details until you need them.
- Java resources:
  - Lecture notes
  - CIS 110 website, textbook
  - 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!

Java and OCaml together

Xavier Leroy, one of the principal designers of OCaml

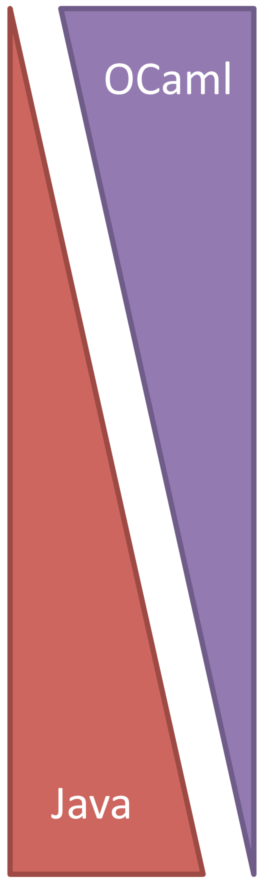
Guy Steele, one of the principal designers of Java

Moral: Java and OCaml are not so far apart…

Looking Back…

Course Overview

- Declarative (Functional) programming
  - *persistent* data structures
  - *recursion* is main control structure
  - frequent 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"



Recap: The Functional Style

- Core ideas:
  - 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)
  - *simple model of computation (substitution)*

- Good for:
  - elegant descriptions of complex algorithms and/or data
  - “symbol processing” programs (compilers, theorem provers, etc.)
  - parallelism, concurrency, and distribution

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, mutable by default
- No first-class functions.* Must encode first-class computation with objects
- Generic types
- Abstract types through public/ private modifiers

*until Java 8, coming March 18th
http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html

OCaml vs. Java for FP

```
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
```

```
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();
  }
}
```

Moar FP

OCaml

- Type inference
- Modules and support for large scale programming
- Objects (real, but different)
- Many other extensions
- Growing ecosystem
- Real World OCaml, OPAM

Microsoft Visual F#

Most similar to OCaml,
Shares libraries with C#

ERLANG

Scalable concurrency
Powers WhatsApp

Haskell (CIS 552)
Purity and laziness

Clojure
Runs on JVM

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)
  - *Abstract Stack Machine model of computation*

- Good for:
  - numerical simulations (nbody)
  - implicit coordination between components (queues, GUI)

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

Explicit vs. Implicit Partiality

OCaml variables

- Cannot be changed once created, must use mutable record

```
type 'a ref = { mutable contents: 'a }
let x = { contents = counter () }
;; x.contents <- counter ()
```

- Cannot be null, must use options

```
let y = { contents = Some (counter ())}
;; y.contents <- None
```

- Accessing the value requires pattern matching

```
;; match y.contents with
     | None -> failwith "NPE"
     | Some c ->  c.inc ()
```

Java variables

- Can be assigned to after initialization

```
Counter x = new Counter ();
x = new Counter ();
```

- Can always be null

```
Counter y = new Counter ();
y = null;
```

- Check for null is implicit whenever a variable is used

```
y.inc();
```

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

The Billion Dollar Mistake

"***I call it my billion-dollar mistake. It was the invention of the null reference in 1965.*** *At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.* ***This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years****.* "

Sir Tony Hoare, QCon, London 2009

Recap (and coming): 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)
- Good for:
  - GUIs!
    - complex software systems that include many different implementations of the same "interface" (set of operations) with different behaviors
  - Simulations
    - designs with an explicit correspondence between "objects" in the computer and things in the real world

"Objects" in OCaml

```
(* The type of counter objects *)
type counter = {
    inc  : unit -> int;
    dec  : unit -> int;
}

(* Create a counter “object” *)
let 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

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)

OO terminology

- *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

Objects in 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;
  }
}
```

class declaration

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

Constructors with Parameters

```
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

```
public class Main {

 public static void
     main (String[] args) {

       Counter c = new Counter(3);

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

   }
}
```

constructor invocation

Creating Objects

- *Declare* a variable to hold a `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();
```

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;
```

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

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 non-helper methods public

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

Did you attend class today?

1. Yes

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 great for programming
  - minor: syntax is a bit 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;
}
```