Programming Languages and Techniques (CIS120)

Lecture 21
October 22, 2018

Transition to Java
Announcements

• HW05: GUI programming
  – Due: TOMORROW at 11:59:59pm

• HW06: Pennstagram
  – Available soon
  – Due: Tuesday, October 30th at 11:59:59pm
  – Java programming
  – Start Early to avoid Halloween conflict!

• Java Bootcamp!!
  – Wednesday, October 24, 6-8 pm
  – STIT B6
Midterm Exam Scores

AVERAGE: 65
MEDIAN: 66
STD. DEV: 13.5
MAX: 95
Goodbye OCaml...
...Hello Java!
Smoothing the transition

• Java Bootcamp!
  – Wednesday, October 24, 6-8 pm, STIT B6

• General advice for the next few lectures: Ask questions, but don’t stress about the details until you need them.

• Java resources:
  – Our lecture notes
  – CIS 110 website and 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
CIS 120 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”
Java and OCaml together

Stephanie Weirich, Penn Prof. (CIS 120 co-developer, major contributor to Haskell)

Guy Steele, one of the principal designers of Java

Xavier Leroy, one of the principal designers of OCaml

Moral: Java and OCaml are not so far apart...
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
  – small-scale compositional design
  – “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, transform and fold
- Generic types
- Abstract types through module signatures

Java

- No primitive data structures, no tail recursion
- Trees must be encoded by objects, mutable by default
- First-class functions less common*
- Generic types
- Abstract types through public/private modifiers

*completely unsupported until recently (Java 8)
OCaml provides a succinct, clean notation for working with generic, immutable, tree-structured data. Java requires a lot more "boilerplate".

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

let is_empty (t:'a tree) : bool =
  begin match t with
    | Empty -> true
    | _ -> 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();
    }
}
```
Other Popular Functional Languages

- **F#**: Most similar to OCaml, Shares libraries with C#
- **Haskell** (CIS 552)
  - Purity + laziness
- **Swift**
  - iOS programming
- **Clojure**
  - Dynamically typed
  - Runs on JVM
- **Racket**: LISP descendant;
  - widely used in education
- **Scala**
  - Java / OCaml hybrid
Recap: The imperative style

• 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
  – implicit coordination between components (queues, GUI)
  – explicit interaction with hardware
Imperative programming

- 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.

- Most types have a **null** element. Partial functions can return **null**.

- Code is a sequence of **statements** that have effects, sometimes using expressions to compute values.

- References are **mutable** by default, must be explicitly declared to be constant.
Explicit vs. Implicit Partiality

**OCaml** identifiers

- Cannot be changed once created; only mutable fields can change

```ocaml
let x = { contents = counter () } ;; x.contents <- counter ()
```

- Cannot be null, must use options

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

- Accessing the value requires pattern matching

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

**Java** variables

- Can be assigned to after initialization

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

- Can always be null

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

- Check for null is implicit whenever a variable is used

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

- If null is used as if it were an object (i.e. for a method call) then a NullPointerException occurs
"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. ... 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
Java Core Language

differences between OCaml and Java
Structure of a Program

• All code lives in (perhaps implicitly named) **modules**.

• Modules may contain multiple **type definitions**, **let**-bound value **declarations**, and top-level **expressions** that are executed in the order they are encountered.

• All code lives in explicitly named **classes**.

• Classes are themselves types.

• Classes contain **field declarations** and **method definitions**.

• There is a single "entry point" of the program where it starts running, which must be a method called **main**.
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)
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><em>don’t use as an expression!!</em></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><strong>structured types</strong></td>
<td>tuples, datatypes, records, functions, arrays (objects encoded as records of functions)</td>
<td>objects, arrays (records, tuples, datatypes, strings, first-class functions are special cases of classes)</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 public/private modifiers</td>
</tr>
</tbody>
</table>
# Arithmetic & Logical Operators

<table>
<thead>
<tr>
<th>OCaml</th>
<th>Java</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>=, ==</td>
<td>==</td>
<td></td>
</tr>
<tr>
<td>&lt;&gt;!, !=</td>
<td>!=</td>
<td></td>
</tr>
<tr>
<td>&gt;, &gt;=, &lt;, &lt;=</td>
<td>&gt;, &gt;=, &lt;, &lt;=</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>mod</td>
<td>%</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>!</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Equality Test**: `=, ==` in OCaml, `==` in Java
- **Inequality**: `<>!, !=` in OCaml, `!=` in Java
- **Comparisons**: `>, >=, <, <=` in OCaml, `>`, `>=`, `<`, `<=` in Java
- **Addition (and string concatenation)**: `+` in OCaml, `+` in Java
- **Subtraction (and unary minus)**: `-` in OCaml, `-` in Java
- **Multiplication**: `*` in OCaml, `*` in Java
- **Division**: `/` in OCaml, `/` in Java
- **Remainder (modulus)**: `mod` in OCaml, `%` in Java
- **Logical “not”**: `not` in OCaml, `!` in Java
- **Logical “and” (short-circuiting)**: `&&` in OCaml, `&&` in Java
- **Logical “or” (short-circuiting)**: `||` in OCaml, `||` in Java
Java: Operator Overloading

• The *meaning* of an operator in Java 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 from int to float
    \[ 4/3.0 \Rightarrow 1.3333333333333333 \]

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

- 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

  every object provides an “equals” method that should “do the right thing” depending on the class of the object
Strings

• **String** is a *built in* Java class
• Strings are sequences of (unicode) 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);
```

- true
- false
- NullPointerException
What is the value of ans at the end of this program?

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);
```
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.
What is the value of *ans* at the end of this program?

```java
String x = "CIS 120";
String z = "CIS 120";
boolean ans = (x == z);
```

- true
- false
- NullPointerException
What is the value of ans at the end of this program?

```java
String x = "CIS 120";
String z = "CIS 120";
boolean ans = (x == z);
```

1. true
2. false
3. NullPointerException

Answer: true(!)

Why? Since strings are immutable, two identical strings that are known when the program is compiled can be aliased by the compiler (to save space).
Always use `s1.equals(s2)` to compare strings!

Compare strings with respect to their content, not where they happen to be allocated in memory...
Object Oriented Programming
Recap: 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
OO programming

**OCaml** (part we've seen)

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

- Flexibility through *composition*: objects can only implement one interface

```ocaml
type 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

```java
class Button extends Widget {
  /* Button is a subtype of Widget */
}
```
OO terminology

• **Object**: a structured collection of encapsulated *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
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() );
    }
}

Objects in Java

- Class name
- Instance variable
- Constructor
- Methods
- Class declaration
- Object creation and use
- Constructor invocation
- Method call
Encapsulating local state

```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;
    }
}
```

```
Encapsulating local state

CIS120

other parts of the program can only access public members

public class Main {
    public static void main (String[] args) {
        Counter c = new Counter();
        System.out.println( c.inc() );
    }
}
```
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

(Java offers a couple of other protection levels — “protected” and “package protected”. The details are not important at this point.)