CSE 341: Programming Languages

Winter 2006 Lecture 28— Closures in Java; What We Didn't Do; Wrap-Up

Goals for today

- Anonymous inner classes in Java
- Give a flavor of big areas of PL we didn't even get to
- Put in context what we did get to

Anonymous inner class example

```
class NPresses {
int objMax;
NPresses(int n) { objMax = n; }
void addToButton(JButton b, final int buttonMax) {
  b.addActionListener(
     new ActionListener() { // ActionListener a library class
       int m = 0;
       public void actionPerformed(ActionEvent e) {
         ++m;
         if(m==objMax)
           System.out.println("enough presses (obj)!");
         else if(m==buttonMax)
           System.out.println("enough presses (button)!");
       }
     });
}}
```

Just sugar?

Did we "need" an anonymous class?

- No: could use an inner class
- No: could use a top-level class and share an object holding objMax and a second field for buttonMax

Higher-order functions in Java?

- Anonymous inner classes are a convenience for making higher-order functions less burdensome.
- Regardless, OO and downcasts let you manually create closures (not shown).
- C# has delegates, which are even closer to first-class functions.

What else?

Are all programming languages imperative, OO, or FP? No.

- Logic languages (e.g., Prolog)
- Scripting languages (Perl, Python, Ruby)
- Query languages (SQL)
- Purely functional languages (no `ref` or `set!`)
- Visual languages, spreadsheet languages, GUI-builders, text-formatters (TEX, LATEX— the typesetting language used for CSE341!), hardware-synthesis, ...

Prolog in one example

```
append(cons(Hd,Tl), Lst2, cons(Hd,Tl2)) :=
     append(Tl, Lst2, Tl2).
append(nil, Lst2, Lst2).

append(cons(1, cons(2, nil)), cons(3, cons(4, nil)), X)
% X = cons(1,cons(2,cons(3,cons(4,nil))))
append(cons(1, nil), cons(2,nil), cons(1, cons(2, nil)))
% yes
append(nil, cons(2,nil), cons(1, cons(2, nil)))
% no
append(cons(Hd,nil), Y, cons(1, cons(2, cons(3, nil))) )
% Hd = 1  Y = cons(2,cons(3,nil))
```

Prolog key ideas

- A program is a set of declarative proof rules.
- Operationally, it's like a function that doesn't distinguish inputs from outputs.
- The implementation searches for the minimal constraints necessary for a formula to be true.
- Different "queries" can run "forward" or "backward"
- This is Turing-complete; killer app is inherently search-oriented tasks, which are common in AI.

Scripting Languages

Few “new” language constructs, but convenience for some quick-and-dirty programs.

- File-system access very lightweight
- Lots of support for string-processing via regular expressions (a different “pattern-matching”)
- Dynamically typed with implicit coercions (such as int to string)
- Tend to have very few “errors” (array resizing, implicit variable declaration, etc.)

Opinion:

- A fine tool for *small* tasks
- They tend to hide bugs rather than prevent them
- But you should learn to automate repetitive tasks!

Query Languages

Canonical example: Suppose there's a big database and many people need data from it. We could make lots of copies or let people submit queries.

Key idea: Move the code to the data, not the data to the code.

Interestingly: We do not necessarily want the query language to be as powerful as a Turing-machine!

SQL was carefully designed so every query terminates.

Purely Functional Languages

Mutation seemed necessary in ML and Scheme for building data structures with cycles. It's not:

- You can build equivalent structures without cycles.
- You can build cycles by cleverly applying functions to themselves
- In fact, you can build recursion the same way (recall `((lambda (x) x x) (lambda (x) x x))`).
- In fact, this subset of Scheme is Turing-complete:

```
e ::= x | (lambda (x) e) | (e1 e2)
```

This language is "impractical" but it's an important fact. For example, SQL can't include these features.

Real Purely Functional Languages

Example: Haskell

To make life without refs palatable, the default is "lazy" (call-by-need) evaluation.

One-line example: `let ones = 1::ones`

Laziness can lead to elegant programming and really increases the number of equivalent programs. In Haskell, `(f x) + (f x)` and `(f x) * 2` are contextually equivalent, always.

- Haskell does have *monads*, which allow a more imperative style.
- The implementation of laziness uses mutation, but in a controlled way (we did this in Scheme).

Each-of types and operations

- Given a type with several variants/subtypes and several functions/methods, there's an obvious 2D-grid of code you need:

	Int	Negate	Add	Mult
eval				
toString				
hasZero				

- OO and FP lay out the code differently.
- Which is more convenient depends on what you're doing and how the variants/operations "fit together"
- Often, tools let you view "the other dimension"
- Opinion: Dimensional structure of code is greater than we have on a computer, so we'll always have limits in text-based languages.

Extensibility

Life gets interesting if need to extend code w/o changing existing code.

- ML makes it easy to write new operations; Java does not.
- Java makes it easy to write new variants; ML does not.
- In ML the original code must *plan* for extensibility:
  - For operations, use polymorphism and function arguments. For example, use folds or even abstract the constructors.
  - For types, can use polymorphism, but the lack of subtyping makes it awkward to use the extended types.
- In Java the extended code can *suffer from downcasts*...
  - For types, use interfaces and have new classes implement them.
  - For operations, must downcast to know the operations exist.
- ...or plan for extensibility (e.g., the "visitor pattern")

Unextensibility

Extensibility is not all it's cracked up to be:

- Makes original code more difficult to change later.
- Makes code harder to reason about locally (e.g., dynamic dispatch or functions-as-arguments mean you never know what code might execute next)

Ignored Language Features

- Threads (potential safety problem: race conditions)
- Interoperability (component / software-architecture languages, foreign-function interfaces, more "open" garbage collectors)
- Aspects (yet another way to change program layout—beyond the 2-D grid)
- `eval` and reflection: For over 50 years, LISP (and later Scheme) programs have been able to build arbitrary programs at run-time and evaluate them.
- ...

But we still did a lot

A thorough understanding of higher-order programming, variable scope, semantics of FP and OO, important idioms, static typing, ...

Oh, and you learned a healthy amount of 3 new languages.

Hopefully:

- The time you need to "pick up" a language will drop dramatically (though you have to learn big libraries too)
- You will use mutation for what it's good for and not to create brittle programs with lots of unseen dependencies
- Understand syntax matters, but it's not that interesting
- Apply idioms in languages other than where you learned them
- Recognize language-design is hard and semantics should not be treated lightly.

Context

In most courses and jobs, a programming language is just a means to an end (and only one of many means).

This course was perhaps your one chance to study languages as designs that are *themselves* fascinating, beautiful, and sometimes awkward

I believe this makes you a better programmer, even if the rest of your life is spent in Java (which it won't be)

"Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot."

Eric Raymond, "How to Become a Hacker"