CSci 4223
Principles of Programming Languages

Lecture 15
Prof. Clarkson
Spring 2013

Review
• Functional programming language features: done
  – But SML examples will sometimes continue to occur
• Rest of course: object-oriented language features
  • Today:
    – Mutability (i.e., imperative features)
      • http://www.youtube.com/watch?v=QzntZLHcYy0
      – Intro to Ruby

Mutable features of SML
• Time to finally admit that SML has mutable features
  – Sometimes it really is best to allow values to change
  – E.g., call a function that returns an incremented counter every time
• ML variables really are immutable
• But ML has mutable references
  – Type constructor `t ref` where `t` is a type
  – New expressions:
    • `ref e` to create a reference with initial contents `e`
    • `e1 := e2` to update contents. Evaluates to `{}`
    • `!e` to retrieve contents (note, not negation)
    • `e1; e2` to evaluate `e1`, evaluate `e2`, and return value of `e2`.
      • Value of `e1` is ignored, but it can mutate state

References example

```ml
val x = ref 42
val y = ref 42
val z = x
val _ = x := 43
val w = (!y) + (!z) (* 85 *)
(* x + 1 does not type-check *)
```

• A variable bound to a reference (e.g., `x`) is immutable: it will always refer to
  the same reference
• But the contents of the reference may change via `:=`
• And there may be aliases to the reference, which matter a lot
• Reference are first-class values
• Can think of references as a one-field mutable object
  – so `:=` and `!` don’t specify the field

References in HW4
• There is one piece of code in HW4 specification that you may use for
  mutable effects

```ml
val counter = ref 0
fun new_var () =
  (counter := (!counter) + 1;
   "X" ^ Int.toString(!counter))
```

• `new_var()` returns "X1"
• Then `new_var()` returns "X2"
• Then `new_var()` returns "X3"
• Etc.
• Try not to abuse your new-found power!

OOP features
Ruby

- Pure object-oriented: all values are objects (even numbers)
- Class-based: every object has a class that determines behavior
  - Like Java, unlike JavaScript
  - Mixins (between Java interfaces and C++ multiple inheritance)
- Dynamically typed
- Convenient reflection: run-time inspection of objects
- Blocks and libraries encourage lots of closure idioms
- Syntax and scoping rules of a "scripting language"
  - Often many ways to say the same thing
  - Variables "spring to life" on use
  - Lots of support for string manipulation (we won’t do this)
- Popular for building server-side web applications
  - But we won’t discuss Ruby on Rails

Where Ruby fits

<table>
<thead>
<tr>
<th>Functional</th>
<th>Dynamic</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>SML</td>
<td>Ruby</td>
<td>Java</td>
</tr>
</tbody>
</table>

Smalltalk is also a dynamically typed, class-based, pure OOP language with blocks and convenient reflection that’s been around since 1980’s

- Probably more elegant than Ruby
- Probably less useful than Ruby

Why use dynamically typed language?

- Experience: other "scripting" languages are similar
- Essence of OOP: not having types makes it clearer what objects really are

Logistics

- Lecture materials won’t recount every little language feature we use
  - Thomas book quite readable
  - Can skip/skim regexp and ranges
  - Also see online library documentation (large, searchable)
- Focus in class will be on OOP, dynamic typing, blocks, mixins
- We will use Ruby 1.9.3
  - Ruby 1.8.x is not compatible, but not hugely different
  - "The real world" is still using both a lot
- Installation instructions, etc. on course website
  - Can run programs with a REPL called irb

Defining a class

[For full code details and various expression constructs, see lec19.rb]

```
class Rational
  # no instance variable (field) decls
  def initialize (num, den=1)
    @num = num
    @den = den
  end
  def print ...
  end
  def add r ...
  ...
end
```

Using a class

- Every object has a class, which determines its behavior

- ClassName.new(args) creates a new instance of ClassName and calls its initialize method with args
- Every variable holds an object (possibly the nil object)
  - Local variables (in a method): foo
  - Instance variables (fields): @foo
  - Class variables (static fields): @@foo
- Objects interact by method call (a.k.a. message send)
- Examples: x.y  x.m2(y.m3)  -42.abs
  - m and m(.) are sugar for self.m and self.m(.)
  - e1 + e2 is sugar for e1.+(e2) (really?)

Method accessibility

- private: accessible only by the same object
- protected: accessible only by objects of the same class or one of its subclasses
- public: accessible by all objects

Be careful: private and protected are same words as Java, but different meanings.
Some syntax / scoping gotchas

• Create variables (including instance variables) implicitly by assigning to them
  – So a misspelling just creates a new variable
  – Different instances of a class could have different fields

• Newlines matter
  – Often need more syntax to put something on one line
  – Indentation is only style (not true in some languages)

• Class names must be capitalized

• Message sends don’t need parentheses

• self is a special keyword (Java’s this)

Getters and setters

• If you want outside access to get/set instance variables, must define methods

```ruby
def foo
  @foo
end
```

```ruby
def foo= a
  @foo = a
end
```

• Syntactic sugar for methods names ending with =

```ruby
x.foo
x.foo = 42
```

• Shorter syntax for defining getters and setters is:

```ruby
attr_reader :foo
attr_writer :foo
```

• Or both at same time with attr_accessor

• Overall, requiring getters and setters is more uniform and more OO
  – Can change the methods later without changing clients
  – Particular form of change is subclass overriding (next lecture)

Top-level

• Expressions at top-level are evaluated in the context of an implicit “main” object of class Object

• That is how a standalone program would “get started” rather than requiring an object creation and method call from within irb

• Top-level methods are added to Object, which makes them available everywhere

Class definitions are dynamic

• All definitions in Ruby are dynamic

• Example: Any code can add or remove methods on existing classes
  – Could add your own method to the Array class, for example, but it would be visible to all arrays

• Changing a class affects even already-created instances

• Disastrous example: Changing Fixnum’s + method

Duck Typing

"If it walks like a duck and quacks like a duck, it’s a duck"
  – Or don’t worry that it may not be a duck

When writing a method you might think, “I need a Foo argument” but really you need an object with enough methods similar to Foo’s methods that your method works

– Embracing duck typing is always making method calls rather than assuming/testing the class of arguments

Pre: More code reuse; very OO approach

– What messages an object receive is all that matters

Con: Almost nothing is equivalent

– x+x versus x*2 versus 2*x

– Callers might assume a lot about how callees are implemented

Duck Typing Example

```ruby
def mirror_update pt
  pt.x = pt.x * (-1)
end
```

• Natural thought: "Takes a Point object (definition not shown here), negates the x value"

  – Makes sense, though a Point instance method more OO

• Closer: "Takes anything with getter and setter methods for @x instance variable and multiplies the x field by -1"

• Closer: "Takes anything with methods @x= and @x where result of @x= has a * method that can take -1. Sends result of calling @x the * message with -1 and sends that result to @x="