CSE341: Programming Languages

Introduction To Ruby; Dynamic OOP; "Duck Typing"

Alan Borning
Autumn 2012
(slides borrowed from Dan Grossman)

The plan

• Lecture materials may not recount every little language feature we use
  - Thomas book (2nd edition, Chapters 1-9) quite readable
  - Can skip/skim regexps and ranges
  - Also see online library documentation [large, searchable]

• Focus in class will be on OOP, dynamic typing, blocks, mixins

Logistics

• We will use Ruby 1.9.3
  - Installed on the Lab machines (Windows and linux)
  - Ruby 1.8.7 is not hugely different – should work as well
  - We’ll keep an eye out for any differences that are relevant for this course; in the worst case you may need to upgrade
• Installation instructions, etc. on course web page
  - Can run programs with a REPL called irb
• Homework 7 is a Ruby warmup exercise

A Bit of History

• Some notable examples of early object-oriented languages and systems:
  - First object-oriented programming language: Simula I, then Simula 67, created by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center in Oslo.
  - Smalltalk: developed at Xerox Palo Alto Research Center by the Learning Research Group in the 1970's (Smalltalk-72, Smalltalk-76, Smalltalk-80)
  - Today: mature language paradigm. Some significant examples: C++, Java, C#, Python, Ruby

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 (neither Java interfaces nor 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 (Ruby on Rails)

Where Ruby fits

<table>
<thead>
<tr>
<th></th>
<th>dynamically typed</th>
<th>statically typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>functional</td>
<td>Racket</td>
<td>Haskell</td>
</tr>
<tr>
<td>object-oriented</td>
<td>Ruby</td>
<td>Java</td>
</tr>
</tbody>
</table>

Note: Racket also has classes and objects when you want them
  - In Ruby everything uses them (at least implicitly)

Historical note: Smalltalk also a dynamically typed, class-based, pure OOP language with blocks and convenient reflection
  - Smaller just-as-powerful language
  - Contrast Ruby’s “why not add that” attitude
  - Ruby less elegant, more widely used

Dynamically typed OO helps identify OO’s essence by not having to discuss types
Defining a class

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

```ruby
Class PosRational
  # no instance variable (field) decls
  # just assign to @foo to create field foo
  def initialize(num, den=1)
    @num = num
    @den = den
    end
  def to_s
    end
  def add r ...
    end
end
```

Using a class

• `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`
• You use an object with a `method` call
  – Also known as a message send
  – Every object has a class, which determines its behavior
• Examples: `x. m 4 x.m.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 / variable visibility

• `private`: only available to object itself
• `protected`: available only to code in the class or subclasses
• `public`: available to all code

This is different than what the words mean in Java
• All instance variables and class variables are `private`
• Methods are `public` by default
  – There are multiple ways to change a method's visibility

Some syntax / scoping gotchas

• You 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 with 0 or 1 argument 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
  def foo= a
    @foo = a
  end
  x.foo = 42
  attr_reader :foo
  attr_writer :foo
```

• 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 with 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
  - Very occasionally useful (or cute) to add your own method to the Array class for example, but it is visible to all arrays
- Changing a class affects even already-created instances
- Disastrous example: Changing Fixnum's + method
- Overall: A simple language definition where everything can be changed and method lookup uses instance's classes

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
- Plus: More code reuse; very OO approach
- What messages an object receive is all that matters
- Minus: Almost nothing is equivalent
  - x*x versus x*2 versus 2*x
  - Callers may assume a lot about how callees are implemented

Duck Typing Example

```
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 and calls x= with the result of multiplying result of x and -1"
- Duck typing: "Takes anything with method 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="