CSE 413 Autumn 2008

Ruby Duck Typing, Classes & Inheritance

Overview

- Next big topic is typing, classes, and inheritance
- But first, a couple of useful things
  - Shorthand for getters/setters
  - An example of an “each” iterator
  - A little more about blocks vs. Procs

Getters/Setters

- Recall that all instance variables are *really* private – need to define methods to access them

```
class PosRat
  def initialize(num, denom=1)
    @num = num
    @denom = denom
  end

  def num
    @num
  end
  def num=(value)
    @num = value
  end
  …
```

An Alternative

- Was:

```
def num
  @num
end
def denom
  @denom
end
…
```

- Instead, can use

```
attr_reader :num, :denom
```

- There is a similar attr_writer shortcut

Iterator Example

- Suppose we want to define a class of Sequence objects that have a from, to, and step, and contain numbers x such that
  - from <= x <= to, and
  - x = from + n*step for integer value n

(Credit: *Ruby Programming Language*, Flanagan & Matsumoto)

Sequence Class & Constructor

```
class Sequence
  # mixin all of the methods in Enumerable
  include Enumerable

  def initialize(from, to, step)
    @from, @to, @step = from, to, step
  end
  …
```

Sequence each method

- To add an iterator to Sequence and make it also work with Enumerable, all we need is this:

```
def each
  x = @from
  while x <= @to
    yield x
    x += @step
  end
end
```

Blocks & Procs Revisited

- Blocks are only usable in the immediate context of their caller

  ```
  thing.each { | x | do_something_with(x) }
  ```

- Procs are real “first-class” objects
  - Create with lambda or Proc.new
  - Proc instances all have a “call” method
  - Can be stored in fields, passed as arguments, etc.
  - This is exactly a closure

Types in Ruby

- Ruby is dynamically typed – everything is an object
- Only notion of an object's "type" is what messages it can respond to
  - i.e., whether it has methods for a particular message
  - This can change dynamically for either all objects of a class or for individual objects

Duck Typing

- “If it walks like a duck and talks like a duck, it must be a duck”
  - Even if it isn’t
  - All that matters is how an object behaves
    - (i.e, what messages it understands)

Thought Experiment (1)

- What must be true about x for this method to work?

```
def foo x
  x.m + x.n
end
```

Thought Experiment (2)

- What is true about x?

  x.m + x.n

- Less than you might think
  - x must have 0-argument methods m and n
  - The object returned by x.m must have a + method that takes one argument
  - The object returned by x.n must have whatever methods are needed by x.m.+ (!)

Duck Typing Tradeoffs

- Plus
  - Convenient, promotes code reuse
  - All that matters is what messages an object can receive
- Minus
  - “Obvious” equivalences don’t hold: x+x, 2*x, x*2
  - May expose more about an object than might be desirable (more coupling in code)

Classes & Inheritance

- Ruby vs Java:
  - Subclassing in Ruby is *not* about type checking (because of dynamic typing)
  - Subclassing in Ruby is about *inheriting methods*
- Can use super to refer to inherited code
- See examples in points.rb
  - ThreeDPoint inherites methods x and y
  - ColorPoint inherits distance methods

Overriding

- With dynamic typing, inheritance alone is just avoiding cut/paste
- Overriding is the key difference
  - When a method in a superclass makes a *self* call, it resolves to a method defined in the subclass if there is one
  - Example: distFromOrigin2 in PolarPoint

Ruby Digression

- Since we can add/change methods on the fly, why use a subclass?
- Instead of class ColorPoint, why not just add a color field to Point?
  - Can’t do this in Java
  - Can do it in Ruby, but it changes all Point instances (including subclasses), even existing ones
  - Pro: now all Point classes have a color
  - Con: Maybe that breaks something else