



CSE 413 Spring 2011

Ruby Tips, Modules & Mixins, Duck Typing & Inheritance

Overview

- Next big topic is typing, classes, and inheritance
- But first, a couple of useful things
 - Where's "main"?
 - Shorthand for getters/setters
 - (Later) an example of an "each" iterator
- Then modules & mixins; duck typing

Where's "main"?

- Traditional programming languages start programs in `publicstaticvoidmain` or equiv.
- Ruby? No main method
 - A "program" is just a sequence of statements / expressions executed in order
 - But these can include class definitions, methods
 - Code outside a class belongs to the default, top-level Object class

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

  ...
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

Organizing Large(r) Programs

■ Issues

- Idea: divide code into manageable components
- Also: want to take advantage of reusable chunks of code (libraries, classes, etc.)

■ Strategy: Split code into separate files

- Typically, one or more classes per file
- Use “require” (or sometimes “load”) to access
- What about components that aren't classes?



Namespaces & Modules

- Idea: Want to break larger programs into pieces where names can be reused independently
 - Avoids clashes combining libraries written by different organizations or at different times
- Ruby solution: modules
 - Separate source files that define name spaces, but not necessarily classes

Example (from Programming Ruby)

```
module Trig
  PI = 3.14
  def Trig.sin(x)
    # ...
  end
  def Trig.cos(x)
    # ...
  end
end
```

```
module Moral
  VERY_BAD = 0
  BAD      = 1
  def Moral.sin(badness)
    # ...
  end
end
```


Using Modules

```
# ...  
require 'trig'  
require 'moral'  
y = Trig.sin(Trig::PI/4)  
penance = Moral.sin(  
    Moral::VERY_BAD)  
# ...
```

- Key point: Each module defines a namespace
 - No clashes with same names in other modules
- Module methods are a lot like class methods

Mixins

- Modules can be used to add behavior to classes – *mixins*
 - Define instance methods and data in module
 - “include” the module in a class – incorporates the module definitions into the class
 - Now the class has its original behavior plus whatever was added in the mixin
 - Provides most of the capabilities of multiple inheritance and/or Java interfaces

Example

```
module Debug
  def trace
    # ...
  end
end
class Something
  include debug
  # ...
end
```

```
class SomethingElse
  include debug
  # ...
end
```

- Both classes have the trace method defined, and it can interact with other methods and data in the host class as if it was defined there
 - (trace is not “shared” by the classes and can’t pass information back and forth)

Exploiting Mixins – Comparable

- The real power of this is when mixins build on or interact with code in the classes that use them
- Example: library mixin Comparable
 - Class must define operator `<=>`
 - `(a <=> b)` returns -1, 0, +1 if `a<b`, `a==b`, `a>b`)
 - Comparable mixin uses “client” `<=>` to define `<`, `<=`, `==`, `>=`, `>`, and `between?` for that class

Another example – Enumerable

- Container/collection class provides an each method to call a block for each item in the collection
- Enumerable module builds many mapping-like operations on top of this
 - map, include?, find_all, ...
 - If items in the collection implement \leq you also get sort, min, max, ...

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
 - $\text{from} \leq x \leq \text{to}$, and
 - $x = \text{from} + n * \text{step}$ 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
```


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)
- May allow objects to “work” in unintended / inappropriate contexts

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` inherits 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 – Why Subclasses?

- 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 or is the wrong abstraction for some Point clients