CSC/ECE 517

Introduction to Ruby
Your host for this timepart

  - Self-taught software developer for 40 years
  - Has written code in 20+ languages
  - Has worked on F/LOSS for two decades
  - Core member of the Apache Group and founding member of The Apache Software Foundation
  - Currently principal s/w engineer in Red Hat IT, working on automation and technical debt payoff
  - Author of *Apache Server for Dummies* and co-author of *Apache Cookbook*
  - Ruby coder for 6+ years
Conventions

- In Ruby code, you'll see a lot of syntax elements being omitted if the compiler can figure things out without them.
- I prefer to be explicit:
  - I parenthesize argument lists.
  - When invoking a method, I prefix it with 'self.' just so it doesn't confuse people into thinking it might be a local variable.
  - I try to keep my code lines under 80 characters (I did use punchcards for years).
  - I use the rocket ('=>') for hash element assignment, rather than the ':' key prefix.
Everything is an object

- Classes, modules, variables, constants, numbers – even methods
- Statements (*class*, *module*, *def*, *etc.*) are language elements, not objects
#initialize versus #new

• The constructor method for a class is named 'initialize', but you instantiate the class with the 'new' method – why?
  - 'new' is a class method that takes care of allocating storage for the instance, before calling in 'initialize'
Access control

- Relates to inheritance (subclasses, etc.) and access to methods
  - public
    Anyone can access a public method
  - protected
    Only instances of the declaring class, or of subclasses, may access the method
  - private
    Only instances of the declaring class may invoke a private method
Access Declaration

• There are two ways of specifying the access limitations of a method:

  1. Including the keyword by itself in the code. All declarations that follow will inherit that protection

  2. Using the keywords as functions, and providing a list of methods to be given that protection. *E.g.*, 

     ```ruby
     def method1(args)
     end
     protected(:method1)
     ```

• I personally greatly prefer the latter; it avoids the complications of large-scale cut&paste into other portions of the file – with possibly different default protection semantics
Modules

- Modules perform two main functions:
  - Namespacing, and
  - Providing sets of common attributes and methods that can be automatically passed down to classes that include the module. A module used in this way is called a *mixin*. 
Namespacing for Module Methods

- The namespacing aspect of a module not only wraps all the declarations inside it within a protective shield to protect against name collisions, but with the user of the eigenclass concept and the `#module_function` method you can associates methods and variables directly with the module.
Mixin *versus* module method

```ruby
module Special
  def mixin_method
    puts('Hi, I was mixed in!')
  end
  def how?
    puts('Really, really special')
  end
  module_function(:how?)
end

class Foo
  include Special
end

bar = Foo.new
bar.mixin_method
 #=> Hi, I was mixed in!
bar.how?
 #=> NoMethodError exception
Special.how?
 #=> Really, really special.
```
More module namespacing

• Modules provide a convenient way to encapsulate methods and values that don't really have a 'self' upon which to act

Math.class
=> Module
Math.constants
=> [:DomainError, :PI, :E]
Math.methods(false).sort
Modules as Mixins

- Modules provide a way to keep your code DRY without having to subclass all over the place.
- Every method or accessor declared in a module (and not flagged as special with `#module_function` or being declared in the eigenclass) will be inherited by any class or module that mixes it in with an `include` statement.
- N.B.: `#require` and `#load` (yes, they're methods) deal with files; `include` deals with named objects. Don't try to `include` a file; it won't work.
Reflection

- Everything is an object, so it may have variables and methods. Different categories of objects use different method names to bare their inner being to you. The following lists are **partial**.
- Pretty much all objects (classes, modules, instances) support some common methods:
  - #methods
  - #send
  - #hash
  - #inspect
  - #method
  - #methods
  - #object_id
  - #respond_to?
  - #to_s
Class Reflections

- Classes have class variables and constants, and methods relating to them
  - #ancestors
  - #included_modules
  - #class_variables
  - #class_variable_defined?
  - #class_variable_get/set
  - #constants
  - #const_defined?
  - #const_get/set
  - #const_missing
  - #name
Reflecting upon Instances

- Instance object have their own set of standard methods
  - #class
  - #clone
  - #dup
  - #eql?
  - #equal?
  - #instance_variables
  - #instance_variable_defined?
  - #instance_variable_get/set
  - #nil?
Exercise

1. How can you determine from what class an arbitrary instance was instantiated?

2. How can you determine what subclasses and/or modules are contributing to an instance's attributes and methods?

3. How can you tell whether an arbitrary object can handle a particular method call?

4. How can you tell whether an object possesses a particular instance variable?

5. Extra credit: How can you change the value of an object's instance variable if it doesn't provide a writer method?

6. Extra credit: How can you determine the arity of an arbitrary method of an object?
Eigenclasses

- The *eigenclass* of an object is essentially a class within a class (or module) declaration defining or altering the object's metadata.

- It's most often seen when declaring class methods:

```ruby
class Foo
  def self.IFF
    return self.name
  end
  def IFF
    return ('%0x' % self.object_id)
  end
end

foo = Foo.new
bar = Foo.new
foo.IFF => "2ad541821c98"
bar.IFF => "2ad54195d990"
Foo.IFF => Foo
foo.class.IFF => Foo
bar.class.IFF => Foo
```
Eigenclases

• In the previous example, methods were declared into the eigenclass by prefixing them with 'self.' This works because during the declation, self is defined as the class (or module) being declared.

• Another syntax you will see is this:

```ruby
class Foo
  class << self
    def IFF
      return self.name
    end
  end
  def IFF
    return '%0x' % self.object_id
  end
end
```
**Eigenclasses with 'class << self'**

- One of the advantages of this way of defining the eigenclass is that it lets you declare class variables using the canonical `attr_accessor`, `attr_reader`, and `attr_writer` invocations, as well as defining class methods.
Open Classes

• Ruby has open classes (and modules), which means that can modify their behavior – even the standard classes.

• This is done simply by opening the class, entering your additions or changes, and closing it.

• Because classes are singletons, any such changes are instantly available to all instances of the class.
Modifying Classes

173.sound_off
=> NoMethodError exception
class Integer
  def sound_off
    return 'Woof!'
  end
end
173.sound_off
=> "Woof!"
29992.sound_off
=> "Woof!"
1.5.sound_off
=> NoMethodError exception
class Float
  def sound_off
    return 'Meow!'
  end
end
1.5.sound_off
=> "Meow!"
Math::PI.sound_off
=> "Meow!"