Ruby Explorations V

Mark Keane...CSI...UCD

De Basics IV

Part IV: A Few Basics More...

A: modules, mixins & inheritance

B: back to how Ruby works...

C: itunes revisited...v.2

D: eine kleine logik musik...

E: hash tables

Part A:

modules, mixins and inheritance

Utilities is ODD

- in iTunes v.1, utilities.rb (and reader.rb to a less extent) is odd
- it is a class without real object instances, just really collections of methods
- so, we called utilities.rb a **module**, to indicate this
- modules are a whole new ballgame...but first let's consider them on their own

Utilities is ODD

```
class Reader

 def read_in_songs(csv_file_name)
   songs = []
   CSV.foreach(….
```

reader.rb

```
module Util
  def self.fetch(string_item, out =
[])
    ...
  end
```

utilities.rb

Why Modules ?

- some call a module, a *degenerate abstract class*; a class that can't instantiate itself, but has methods
- so, why would you want such things...
- well, in any program there will be fns/methods we want to use all over the place (even between apps): think of **sin**, **cos** and **tan** in maths or very general utilities
- modules, carry these, but note to be useful they need (i) to be quite general, (ii) to be introducable...

Consider **isa?**..

- in iTunes v1, we had this crappy **isa?** predicate in song.rb, album.rb, and actor.rb
- in each file, it does the same sort of thing:

```
def isa?
    instance_of?(Album/Actor/Song...)
end
```

- it's also poor, cos' it throws an error, if given anything other than the obj_types it is designed to handle
- maybe it wants to be a module ?

The Pred module

```
module Pred
 def isa?(target_class)
     instance_of?(target_class)
 end
end
```

predicate.rb

- this will take any obj in our prog (song, album..)
- it will test that the inst. is of the given class
- but still throws errors on obj-types other than program objs

```
song = $songs.first
p song.name
p "isa song?"
p song.isa?(Song)
```

```
"Superfast Jellyfish"
"isa song?"
true
```

```
p song.name
p "isa album?"
p song.isa?(Album)
```

```
"Superfast Jellyfish"
"isa album?"
false
```

```
 album = $albums.first
p album.name
p "isa album?"
p album.isa?(Album)
```

```
"Plastic Beach"
"isa album?"
true
```

Hold that thought...

- in any program there are methods we want to use all over the place (even between apps): think of **sin**, **cos** and **tan** in maths or very general utilities
- modules, carry these, but note to be useful they need (i) to be quite general, **(ii) to be introducable...**
- with **isa?** we have seen how we used a module to define a more ***general*** method that can handle many class instances
- we have not shown how they are **introducable**

...so let's digress into inheritance

Inheritance Ia

- one of the big benefits of OOP is *inheritance*
- a class can inherit the methods of another class
- why, would we want to do this ?
- consider, where the idea came from...

Inheritance in birds...

Laysan Albatross pair tending egg at breeding grounds on Sand Island at Midway Atoll NWR Photo by: David Patte USFWS

Semantic Networks

- consider people's knowledge of objects
- it does not make sense to store every attribute with every bird
- makes sense to build a hierarchy and inherit from top to bot
- called principle of cognitive economy

Inheritance Ic

- one of the big benefits of OOP is inheritance
- a class can inherit the methods of another class
- does not make sense to store every *method* with every *object*
- makes sense to build a hierarchy and inherit them from top to bot

Inheritance lla

- let's poke around
- Class.**superclass**
- every class you define is automatically a subclass of Object
- hence, **to_s** works automatically

```
>> String.superclass
=> Object

>> Object.superclass
=> BasicObject

>> BasicObject.superclass
=> nil

>> Float.superclass
=> Numeric

>> Numeric.superclass
=> Object

>> Object.superclass
=> BasicObject

>> BasicObject.superclass
=> nil

>> class Myclass; end
=> nil

>>Myclass.superclass
=> Object
```

Inheritance IIb

- hence, **to_s** works automatically
- **to_s** works for any class you create because it is inherited from Object
- typically, we re-define it to have a specific variant for our class objs

```
>> class Myclass ; end
=> nil

>> Myclass.class
 => Class

>>Myclass.superclass
=> Object

>> foo = Myclass.new
=> #<Myclass:0x11b494>

>> foo
=> #<Myclass:0x11b494>

>> foo.to_s
=> "#<Myclass:0x11b494>"
```

every obj get a
shadowy **to_s** for free

Inheritance III

- Ruby supports single class inheritance; you can only inherit from a single superclass (but there are mixins)
- so, if I define a class X with certain methods x, y, z
- ...and if I define a class Y, saying its subclass of X
- then, Y will automatically inherit methods x, y and z
- note, variables *look-like* they are inherited, but the effects are complicated and subtle

Inheritance is for the birds...

```
class Bird
 attr_accessor :name, :wings, :legs, :beak, :flies, :feathers
 def initialize(name, wings,legs,beak)
     @name = name
     @wings = wings
     @legs = legs
     @beak = beak
     @flies = true
     @feathers = true
 end

 def beak_size
     @beak
 end

 def can_fly?
     if @feathers && @flies then true
         elsif !@flies then false
     end
 end
end

class Seagull < Bird
 def eats_garbage?
     instance_of?(Seagull)
 end
    def beak_size
     @beak + @beak
 end
end
```

inherit.rb

```
class Bird
 attr_accessor :name, :wings, :legs, :beak, :flies, :feathers
 def initialize(name, wings,legs,beak)
     @name = name
     @wings = wings
     @legs = legs
     @beak = beak
     @flies = true
     @feathers = true
 end

 def beak_size
     @beak
 end

 def can_fly?
     if @feathers && @flies then true
         elsif !@flies then false
     end
 end
end

class Seagull < Bird
 def eats_garbage?
     instance_of?(Seagull)
 end
    def beak_size
     @beak + @beak
 end
end
```

method
inherited by subclass

method
inherited by subclass

inherits from Bird

method
specific to seagull

specialisation
of method

inherit.rb

```
class Bird
 attr_accessor :name, :wings, :legs, :beak, :flies, :feathers
 def initialize(name, wings,legs,beak)
     @name = name
     @wings = wings
     @legs = legs
     @beak = beak
     @flies = true
     @feathers = true
 end
...
end

class Seagull < Bird
...
end

class Kiwi < Bird
 attr_accessor  :cute
 def initialize(name, wings,legs,beak)
     @name = name
     @wings = false
     @legs = legs
     @beak = beak
     @flies = false
     @feathers = true
     @cute = true
 end
 def opens_shoe_polish_tins?
     instance_of?(Kiwi)
 end
end
```

inherit.rb [cont.]

```
class Bird
 attr_accessor :name, :wings, :legs, :beak, :flies, :feathers
 def initialize(name, wings,legs,beak)
     @name = name
     @wings = wings
     @legs = legs
     @beak = beak
     @flies = true
     @feathers = true
 end
...
end

class Seagull < Bird
...
end

class Kiwi < Bird
 attr_accessor  :cute
 def initialize(name, wings,legs,beak)
     @name = name
     @wings = false
     @legs = legs
     @beak = beak
     @flies = false
     @feathers = true
     @cute = true
 end
 def opens_shoe_polish_tins?
     instance_of?(Kiwi)
 end
end
```

vars apparently inherited by kiwi

inherits from Bird

looks like re-initialized inherited vars

new attribute

method specific to kiwi

inherit.rb [cont.]

```
jonas = Bird.new("jonas", 2, 2, "long")
seagull2 = Seagull.new("henry", 2, 2, "squat")
kiwi1 = Kiwi.new("kiwi", 2, 1, "long_and_thin")
```

```
p jonas
p seagull2
p kiwi1
```

```
#<Bird:0x2646c @beak="long", @legs=2, @feathers=true, @wings=2, @flies=true, @name="jonas">

#<Seagull:0x26430 @beak="squat", @legs=2, @feathers=true, @wings=2, @flies=true, @name="henry">

#<Kiwi:0x263f4 @beak="long_and_thin", @legs=1, @feathers=true, @cute=true, @wings=false, @flies=false, @name="kiwi">
```

```
p jonas.beak_size
p seagull2.beak_size
p kiwi1.beak_size
# "long"
# "squatsquat"
# "long_and_thin"

p jonas.can_fly?
p seagull2.can_fly?
p kiwi1.can_fly?
# true
# true
# false

#p jonas.eats_garbage?  undefined method `eats_garbage?' for #<Bird:0x266d8>
p seagull2.eats_garbage?
# true
#p kimwi1.eats_garbage?  undefined method `eats_garbage?' for #<Kiwi:0x267c8>

# p jonas.opens_shoe_polish_tins? undefined method `opens_shoe_polish_tins?' for #<Bird:0x26750>
# p seagull2.opens_shoe_polish_tins? undefined method `opens_shoe_polish_tins?' for #<Seagull:0x2678c>
p kiwi1.opens_shoe_polish_tins?
# true
puts "But, kiwis are really cute !: #{kiwi1.cute} => But, kiwis are really cute !: true
```

Variables are not inherited !

this is very subtle, please tighten your safety belt

```
class Bird
 attr_accessor :name, :wings, :legs, :beak, :flies, :batty
 def initialize(name, wings,legs,beak)
     @name = name
     @wings = wings
     @legs = legs
     @beak = beak
     @flies = true
     @feathers = true
 end

end

class Seagull < Bird
  def initialize(name, wings,legs)
     @namer = name
     @winger = wings
     @legser = legs
  end
end

class Kiwi < Bird
 :nothing
end
```

inherit_vars.rb

```
jonas = Bird.new("jonas", 2, 2, "long")
seagull2 = Seagull.new("henry", 2, 2)
kiwi = Kiwi.new("K", 3, 3, "very long")

puts "stage 1:"
p jonas
p seagull2

puts "stage 2:"
jonas.batty = "yes"
seagull2.batty = "yes"
p jonas
p seagull2
```

```
$ruby inherit_vars.rb

stage 1:

#<Bird:0x007f98e4084178 @name="jonas",
@wings=2, @legs=2, @beak="long",
@flies=true, @feathers=true>

#<Seagull:0x007f98e4084128
@namer="henry", @winger=2, @legser=2>

stage 2:

#<Bird:0x007f98e4084178 @name="jonas",
@wings=2, @legs=2, @beak="long",
@flies=true, @feathers=true,
@batty="yes">

#<Seagull:0x007f98e4084128
@namer="henry", @winger=2, @legser=2,
@batty="yes">
```

```
jonas = Bird.new("jonas", 2, 2, "long")
seagull2 = Seagull.new("henry", 2, 2)
kiwi = Kiwi.new("K", 3, 3, "very long")

. . .
puts "stage 3:"
seagull2.flies = true
p jonas
p seagull2
p kiwi
```

```
$ruby inherit_vars.rb
…
stage 3:
#<Bird:0x007f98e4084178 @name="jonas",
@wings=2, @legs=2, @beak="long",
@flies=true, @feathers=true,
@batty="yes">

#<Seagull:0x007f98e4084128
@namer="henry", @winger=2, @legser=2,
@batty="yes", @flies=true>

#<Kiwi:0x007f98e40840b0 @name="K",
@wings=3, @legs=3, @beak="very long",
@flies=true, @feathers=true>

$
```

Variables are not inherited !

- methods are inherited from super- to sub-classes (including initialization methods, right !)
- so, it *looks like* instance variables are inherited from the superclass (bird) to subclass (seagull); but they aren't
- all objects have instance variables that are simply created in an instance, as values are assigned to them
- NO inheritance of variables or values occurs; instance variables have no role in the inheritance mechanism
- but, <u>methods</u> may be inherited that do a set of assignments to same-named variables...

Pause...
Deep Breath...
New Topic...

...let's return from our digress into inheritance

REM: Hold that thought...

- in any program there are methods we want to use all over the place (even between apps): think of **sin**, **cos** and **tan** in maths or very general utilities
- modules, carry these, but note to be useful they need (i) to be quite general, **(ii) to be introducable...**
- we used a module with a more *general* method of **isa?** that can handle many class instances
- but, we have *not* shown how they are **introducable**

Modules & Mixins

```
module Pred
  def isa?(target_class)
      instance_of?(target_class)
  end
end
```

predicate.rb

- single-class inheritance looks like a limitation, but its fixed by **modules** can introduce its methods into any class (using **include** at the start)

song.rb

```
require 'predicate'
class Song
    include Pred
        attr_accessor :name, ...
        def initialize(name, albu...
            @name = name
            @album = album
            @time = time
            @artist = artist...
```

- in these **mixins** all the module's methods become available to that class

- this gets a lot more complicated..baby...a story for another day (e.g. Kernel and Enumerable)

Part B:

how Ruby works...

Ruby is a...

REM:

- scripting language; not a static, compiled language like C or Java
- so, Ruby programs are lists of statements to be executed (or scripts)
- statements are executed sequentially; except where flow of control is diverted (e.g., by **if**...)
- note, class/module definitions, methods and method calls are all statements in Ruby
- Ruby steps through the statements evaluating each

Understanding...

- it helps to understand the evaluation model in such functional languages
- if you understand evaluation, you will understand Ruby
- ...will help you make sense of weird bugs...when you inevitably feed the interpreter some garbage code...

As Ruby rumbles forward...

- ...it is evaluating three main types of entities:
  1. objects/variables on their own
     (constants, class v, method v and local v)
  2. methods (class and instance)
  3. classes (classes proper, modules...)

Evaluation of *variables*

- if the entity is a variable then Ruby evaluates it returning the value of that variable (PI, myname, _fup)
- if the entity is a literal -- i.e. a value that appears directly in the code like a no, string, symbol or regular expression -- then Ruby still evaluates it, returning itself
- true, false, nil are treated similarly (though they are really part of Ruby's keyword set)
- Keywords are treated specially by the Ruby parser

RDoc Documentation

www.ruby-doc.org/docs/keywords/1.9/

Home&Abroad | Finance | Net&Functions | MyCal | Google Scholar | ISI | Health&Exer | Popular | Hacking | Wikis | » Other Bookmar

Files	Classes	Methods
keywords.rb		BEGIN (keywords.rb) END (keywords.rb) __ENCODING__ (keywords.rb) __END__ (keywords.rb) __FILE__ (keywords.rb)

keywords.rb

Path: keywords.rb
Last Update: Thu Oct 22 23:00:56 -0700 2009

RDoc-style documentation for Ruby keywords (1.9.1).

David A. Black

June 29, 2009

Yes, I KNOW that they aren't methods. I've just put them in that format to produce the familiar RDoc output. I've been focusing on the content.

If anyone has a good idea for how to package and distribute it, let me know. I haven't really thought it through.

Also, if you spot any errors or significant omissions, let me know. Keep in mind that I'm documenting the keywords themselves, not the entities they represent. Thus there is not full coverage of, say, what a class is, or how exceptions work.

Changes since first release:

- Added `__END__` (thanks Sven Fuchs)
- Added 'retry' to retry example (thanks mathie)
- Corrected description of when 'rescue' can be used (thanks Matt Neuburg)
- Added `else` in rescue context (thanks Rob Biedenharn)

Methods

BEGIN END __ENCODING__ __END__ __FILE__ __LINE__ alias and begin break case class def defined? do else elsif end ensure false for if in module next nil not or redo rescue retry return self super then true undef unless until when while yield

Public Instance methods

...along with, sort of: **=begin =end**

Steps in *method* lookup*

obj_any.method_any

- it searches the class of **obj_any** for an instance method named **method_any**; if no method is found in the class then...
- it searches the instance methods of any modules included in the class of **obj_any** for **method_any** (backwards); if no method is found then...
- the search moves up the hierarchy to the superclass and repeats the same searches...exhaustively
- if all these searches fail then then **method_missing** is invoked

* ignoring eigenclasses

Steps in *method* lookup*

message = “hello”; message.show_it

- it searches the **String** class of **“hello”** for an instance method named **show_it**; if no method is found in the class then...
- it searches the **Comparable** and **Enumerable** modules of the **String** class for **show_it** (backwards); if no method is found then...
- the search moves up the hierarchy to the superclass **Object** but it has no **show_it** method either
- so, it searches the **Kernel** module included in **Object** but no luck there so it finds the **method_missing** method in **Kernel** and does it

* ignoring eigenclasses

Steps in class *method* lookup*

ClassA.method_any

- ClassA is a constant that refers to an object that is an instance of **Class**; so class methods are *singleton methods* of the object **ClassA**; when we invoke above
- Ruby searches for any class methods defined called **method_any**; if no method is found then...
- it searches for any <u>*instance methods*</u> in the class called **method_any**; if that fails
- the search moves to the methods that Class inherits from **Module**, **Object** & **Kernel** ...exhaustively
- if all these searches fail then then **method_missing** is invoked

I know this is counter-intuitive

*not ignoring eigenclasses

Class **Object**

In: class.c

Ruby version:

Methods

!~ === =~ __id__ class clone define_singleton_method display dup enum_for eql? extend freeze frozen? inspect instance_of? instance_variable_defined? instance_variable_get instance_variable_set instance_variables is_a? kind_of? method nil? object_id public_method public_send remove_instance_variable respond_to? respond_to_missing? send singleton_class singleton_methods taint tainted? tap to_enum to_s trust untaint untrust untrusted?

Included Modules

Kernel

Constants

TOPLEVEL_BINDING	= rb_binding_new()	
STDIN	= rb_stdin	constants to hold original stdin/stdout/stderr
STDOUT	= rb_stdout	
STDERR	= rb_stderr	
ARGF	= argf	
DATA	= f	
ARGV	= rb_argv	
ENV	= envtbl	
RUBY_VERSION	= MKSTR(version)	
RUBY_RELEASE_DATE	= MKSTR(release_date)	
RUBY_PLATFORM	= MKSTR(platform)	
RUBY_PATCHLEVEL	= INT2FIX(RUBY_PATCHLEVEL)	

Module **Kernel**

In: object.c

Ruby version:

`BasicObject` is the parent class of all classes in Ruby. It's an explicit blank class. `Object`, the root of Ruby's class hierarchy is a direct subclass of `BasicObject`. Its methods are therefore available to all objects unless explicitly overridden.

`Object` mixes in the `Kernel` module, making the built-in kernel functions globally accessible. Although the instance methods of `Object` are defined by the `Kernel` module, we have chosen to document them here for clarity.

In the descriptions of Object's methods, the parameter *symbol* refers to a symbol, which is either a quoted string or a `Symbol` (such as `:name`).

Methods

Array Complex Float Integer Rational String __callee__ __method__ ` abort at_exit autoload autoload? binding block_given? callcc caller catch chomp chop eval exec exit exit! fail fork format gets global_variables gsub iterator? lambda load local_variables loop open p print printf proc putc puts raise rand readline readlines require select set_trace_func sleep spawn sprintf srand sub syscall system test throw trace_var trap untrace_var warn

Class **Module**

In: class.c

Ruby version:

Methods

< <= <=> == === > >= alias_method ancestors append_features attr_accessor attr_reader attr_writer autoload auto
class_eval class_exec class_variable_defined? class_variable_get class_variable_set class_variables const_defined? const_get
const_missing const_set constants constants define_method extend_object extended freeze include include? included
included_modules instance_method instance_methods method_added method_defined? method_removed method_undefined
module_eval module_exec module_function name nesting new private private_class_method private_instance_methods
private_method_defined? protected protected_instance_methods protected_method_defined? public public_class_method
public_instance_method public_instance_methods public_method_defined? remove_class_variable remove_const remove_method
undef_method

and finally,
an interesting aside...

A remaining mystery in **def**

An Aside

- most of the time we define instance methods in a class that are invoked using object instances: “str”.size
- sometimes we define class methods to do some task Person.new (or those weird hidden Kernel functions)
- but we have also defined methods at the top-level (as we have done in files) that just take arguments and are not within a class (so, they are not invoked via objects) e.g., hail(mark)

REM:THIS MAY NOW MAKE SENSE

Top-level Methods

An Aside

- ...are instance methods of **Object** (but self is not **Object**, self is **main**)
- ...are always private (don't ask...)
- why ?
  1. since they are methods of **Object** they can (in theory) be used with any object
  2. since they are private they must be invoked like functions with no explicit receiver

REM:THIS MAY NOW MAKE SENSE

Part C:

Cleaning up iTunes...v2...sort of

Entering reflection...

- there are a few things in iTunes v.2
- **isa?** can go into **Pred** module and be introduced as a mixin (with other checking methods)
- we could look at trying to get rid of global variables
- you really want everything to be objects and clean interfaces with some encapsulation

Pred as a Mixin

```
module Pred
  def isa?(target_class)
      instance_of?(target_class)
  end
end
```

predicate.rb

song.rb

```
class Song
    include Pred
    attr_accessor :name, ...
    def initialize(name, albu...
        @name = name
        @album = album
        @time = time
        @artist = artist...
```

actor.rb

```
class Actor
    include Pred
    attr_accessor :name, ...
    def initialize(name, owns...
        @name = name +...
        @lib = owns_lib
        @id = name.obje...
```

album.rb

```
class Album
    include Pred
    attr_accessor :name, ...
    def initialize(name, tracks...
        @name = name
        @tracks = tracks
        @length = length
        @artist = artist...
```

also need to do:
require_relative 'predicate' first in itunes.rb

```ruby
class DataBit
  attr_accessor :songs, :albums, :actors, :hashes
  def initialize()
    @songs = []
    @albums = []
    @actors = []
    @hashes = Hash.new
  end

  def songs=(x)
    @songs = x
  end

  def albums=(x)
    @albums = x
  end

  def actors=(x)
    @actors = x
  end

  def hashes=(x)
    @hashes = x
  end

end
```

data.rb

```
require 'actor','album','library','song','reader','utilities','csv',
'error'

reader = Reader.new
songs_file = 'songs.csv'                #in RubyMine
owners_file = 'owners.csv'              #in RubyMine

puts "\nProcessing Songs from file: #{songs_file}"
$songs = reader.read_in_songs(songs_file)

puts "Processing Ownership from file: #{owners_file}"
$hash_owners = reader.read_in_ownership(owners_file)

puts "Building all owners..."
$actors = Actor.build_all()

puts "Updating songs with ownership details..."
$songs.each{|song| song.owners = $hash_owners[song.id]}

puts "Building All Albums..."
$albums = Album.build_all()
```

OLD

Top-level

itunes.rb

```
require 'actor','album','library','song','reader','utilities','csv',
'error', predicate, data

reader = Reader.new
data = DataBit.new()
songs_file = 'songs.csv'                #in RubyMine
owners_file = 'owners.csv'              #in RubyMine

puts "\nProcessing Songs from file: #{songs_file}"
data.songs = reader.read_in_songs(songs_file)

puts "Processing Ownership from file: #{owners_file}"
data.hashes = reader.read_in_ownership(owners_file)

puts "Building all owners..."
data.actors = Actor.build_all(data)

puts "Updating songs with ownership details..."
data.songs.each{|song| song.owners = data.hashes[song.id]}

puts "Building All Albums..."
data.albums = Album.build_all(data)

# Print out all songs
puts "\nPrinting full details of all songs..."
data.songs.each{|song| p song}
```

NEW

Top-level

itunes.v2.rb

```
class Song
  include Pred
 attr_accessor :name, :album, :artist, :time, :owners, :id
 def initialize(name, album, artist, time, owners, id)
     @name = name
     @album = album
     @time = time
     @artist = artist
     @owners = owners
    @id = id
 end

 def to_s
     puts "<< #{@name} >> by #{@artist} in their
          album #{@album} is owed by #{@owners} .\n"
 end


 def play_song
     no = rand(10)
     no.times {print "#{@name} do be do..."}
     puts "\n"
 end

end
```

song.rb

```
class Album
  include Pred
 attr_accessor :name, :tracks, :length, :artist, :owners, :id
 def initialize(name, tracks, length, artist, owners)
     @name = name
     @tracks = tracks
     @length = length
     @artist = artist
    @owners = owners
     @id = name.object_id
 end

 def to_s
     puts "The album #{@name} by #{@artist}. \n"
 end

 def self.make_album(name,tracks, length, artist, owners)
     Album.new(name, tracks, length, artist, owners)
 end

 def self.build_all(data, albums = [])
    album_names = data.songs.collect{|song| song.album}
    album_names.uniq.each do |album_name|
        albums << self.build_an_album_called(data, album_name)
    end
    albums
  end
. . .

end
```

album.rb

```
class Actor
 include Pred
 attr_accessor :name, :id
 def initialize(name)
     @name = name
     @id = name.object_id
 end

 def to_s
     puts "Actor #{@name} has ID: #{@id}.\n"
 end

 def self.build_all(data, actors = [])
     actor_names = data.hashes.values.clean_up
     actor_names.each {|name| actors << Actor.new(name)}
     actors
 end

 def buys_song(song)
     song.owners <<  (" " + @name)
  end

 def what_songs_does_he_own(data)
      data.songs.select{|song| song.owners.include?(@name)}
  end

end
```

actor.rb

Utilities

```
module Util
   #will fetch object give string that is its name
   def self.fetch(data, string_item, out = [])
       all = data.songs + data.albums + data.actors
      found = all.select{|obj| string_item == obj.name}
       if found.size == 0
         then MyErr.new("not_found_error", string_item, "fetch").do_it
       elsif   found.size > 1
         then  MyErr.new("multiple_answer_error", string_item, "fetch").do_it
       elsif  found.size == 1
         then found.first
       end
   end
end
```

utilities.rb

- slightly prettier...but not much

Exiting reflection...

- there are a few things to do in iTunes v.2
- **isa?** is one
- replacing globals with classes, objects

Advice: Avoid Global,Class Variables

Part C:

Eine kleine logik musik...

If...And...Or

```
def charades(att1, att2, att3)
 if att1 == "green" && att2 == "tall" && att3 == "woody"
     then puts "Is it a tree?"
 elsif att1 == "green" && att2 == "tall" && !(att3 == "woody")
     then puts "Is it a big green thing?"
 elsif (att1 == "green" || att1 == "red") && (att2 == "tall")
     then puts "Is it a traffic light?"
 elsif att1 == "green" || att2 == "tall" || att3 == "woody"
     then puts "Is it a plant?"
 else puts "Haven't a clue..."
 end
end
```

logik.rb

If...And...Or

```
def charades(att1, att2, att3)
 if att1 == "green" && att2 == "tall" && att3 == "woody"
     then puts "Is it a tree?"
 elsif att1 == "green" && att2 == "tall" && !(att3 == "woody")
     then puts "Is it a big green thing?"
 elsif (att1 == "green" || att1 == "red") && (att2 == "tall")
     then puts "Is it a traffic light?"
 elsif att1 == "green" || att2 == "tall" || att3 == "woody"
     then puts "Is it a plant?"
 else puts "Haven't a clue..."
 end
end
```

logik.rb

If...And...Or

&& and &
are quite different

```
def charades(att1, att2, att3)
 if att1 == "green" && att2 == "tall" && att3 == "woody"
     then puts "Is it a tree?"
 elsif att1 == "green" && att2 == "tall" && !(att3 == "woody")
     then puts "Is it a big green thing?"
 elsif (att1 == "green" || att1 == "red") && (att2 == "tall")
     then puts "Is it a traffic light?"
 elsif att1 == "green" || att2 == "tall" || att3 == "woody"
     then puts "Is it a plant?"
 else puts "Haven't a clue..."
 end
end
```

! is NOT

brackets estab.
precedence

double | | is OR

logik.rb

If...And...Or

```
puts "Running charades..."
p charades("green","tall","woody")
p charades("green","round","plastiky")
p charades("red","round","woody")
p charades("green","tall","stoney")
p charades("red","tall","stoney")
p charades("red","small","stoney")
p charades("red","small","woody")
```

```
# Running charades...
# Is it a tree?
# nil
# Is it a plant?
# nil
# Is it a plant?
# nil
# Is it a big green thing?
# nil
# Is it a traffic light?
# nil
# Haven't a clue...
# nil
# Is it a plant?
# nil
```

When...

```
def wh_charades(att1, att2, att3)
 case
    when att1 == "green" && att2 == "tall" && att3 == "woody"
         puts "Is it a tree?"
    when att1 == "green" && att2 == "tall" && !(att3 == "woody")
         puts "Is it a big green thing?"
    when (att1 == "green" || att1 == "red") && (att2 == "tall")
         puts "Is it a traffic light?"
    when att1 == "green" || att2 == "tall" || att3 == "woody"
         puts "Is it a plant?"
    else puts "Haven't a clue..."
 end
end
```

logik.rb

When...is visually easier

```
def wh_charades(att1, att2, att3)
 case
     when att1 == "green" && att2 == "tall" && att3 == "woody"
          puts "Is it a tree?"
     when att1 == "green" && att2 == "tall" && !(att3 == "woody")
          puts "Is it a big green thing?"
     when (att1 == "green" || att1 == "red") && (att2 == "tall")
          puts "Is it a traffic light?"
     when att1 == "green" || att2 == "tall" || att3 == "woody"
          puts "Is it a plant?"
     else puts "Haven't a clue..."
     end
end
```

begin

then-part

if-part

end

logik.rb

When...

```
puts "And Now the Wh version..."
p wh_charades("green","tall","woody")
p wh_charades("green","round","plastiky")
p wh_charades("red","round","woody")
p wh_charades("green","tall","stoney")
p wh_charades("red","tall","stoney")
p wh_charades("red","small","stoney")
p wh_charades("red","small","woody")
```

```
# And Now the Wh version...
# Is it a tree?
# nil
# Is it a plant?
# nil
# Is it a plant?
# nil
# Is it a big green thing?
# nil
# Is it a traffic light?
# nil
# Haven't a clue...
# nil
# Is it
```

Unless...

```
def unlesso(test)
 unless test
     puts "ok"
 else puts "yuk"
 end
end

puts "Unless test..."
puts unlesso(true)
puts unlesso(false)
foo = "anything"
puts unlesso(foo)
puts unlesso(!foo)
```

```
# Unless test
# yuk
# nil
# ok
# nil
# yuk
# nil
# ok
# nil
```

logik.rb

Part D:

Of course...I made a dweadful hash of his arm...

Hash Tables Ia

- arrays have build in indices
- its like every item has a number that points to it
- hash tables have specified indices, useful when you know a key to find something else

```
>> Array.new
=> []
>> foo = ["a","b","c","goo", "gaa"]
=> ["a", "b", "c", "goo", "gaa"]
>> foo[0]
=> "a"
>> foo[4]
=> "gaa"
>> foo[-1]
=> "gaa"
>> foo[-3]
=> "c"
```

Hash Tables Ia

- arrays have build in indices
- its like every item has a number that points to it
- hash tables have specified indices, useful when you know a key to find something else

```
>> Array.new
=> []
>> foo = ["a","b","c","goo", "gaa"]
=> ["a", "b", "c", "goo", "gaa"]
>> foo[0]
=> "a"
>> foo[4]
=> "gaa"
>> foo[-1]
=> "gaa"
>> foo[-3]
=> "c"
```

create new array

first element

fifth element

third from end

Hash Tables Ib

```
>> Hash.new
=> {}
>> foo = {:first => "a", :second => "b", :third => "c", :fourth => "goo",
   :flipped => "gaa"}
=>{:second=>"b", :first=>"a", :third=>"c", :fourth=>"goo", :flipped=>"gaa"}
>> foo[1]
=> nil
>> foo[:first]
=> "a"
>> foo[:flipped]
=> "gaa"
>> foo[:third]
=> "c"
```

Hash Tables Ib

```
>> Hash.new
=> {}
>> foo = {:first => "a", :second => "b", :third => "c", :fourth => "goo",
   :flipped => "gaa"}
=>{:second=>"b", :first=>"a", :third=>"c", :fourth=>"goo", :flipped=>"gaa"}
>> foo[1]
=> nil
>> foo[:first]
=> "a"
>> foo[:flipped]
=> "gaa"
>> foo[:third]
=> "c"
```

create new hash

symbol as index

arrow does mapping

symbol as index

element indexed by :first

element indexed by :flipped

element indexed by :third

REM

```
class Reader
. . .

 def read_in_ownership(csv_file_name, temp_hash = Hash.new)
   CSV.foreach(csv_file_name, :headers => true) do |row|
       song_id, owner_data = row[0], row[1]
       unless (song_id =~ /#/)
              temp_hash[song_id] = owner_data
            end
     end
     temp_hash
  end

end
```

recall csv i/o

hash table used later

hash it

return built hash

```
>> foo[:fff] = "p"
=> "p"
>> foo
=> {:fff=>"p"}
```

reader.rb

Ok...that's a wrap

```
class Bird
 attr_accessor :name, :wings, :legs, :beak, :flies, :feathers
 def initialize(name, wings,legs,beak)
     @name = name
     @wings = wings
     @legs = legs
     @beak = beak
     @flies = true
     @feathers = true
      @batty = “oh..yes!”
 end

    def is_it_batty?
         if @batty then p @batty end
    end
end


class Seagull < Bird
    def  beak_size
         @beak + @beak
    end
end
```

inherit_vars.rb

Ruby Core

(Jerome DALBERT)