Ruby Explorations IX

Mark Keane...CSI...UCD

Pausing for breath...

A: step back...some OOP theory

B: let's upgrade things

C: Ruby dealing with databases

D: more chips off the old...lambda

D: look-back and questions...

Part A:

Step back...some OOP theory

OOP

we have now seen a lot of OOP

recall, the lost years...when we were young

step back and consider some of the theoretical issues, now that we have some experience

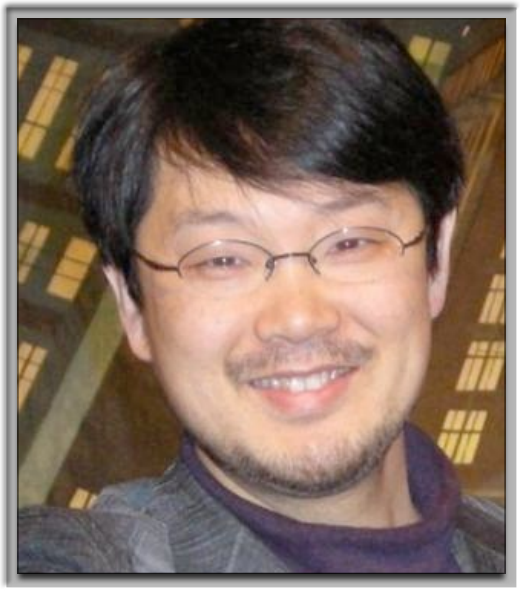
develop an appreciation for the raw beauty of Ruby...before muddying the waters with Rails

REM:
Ruby History

created by Yukihiro Matsumoto in 1993

a language balancing functional and imperative programming

Tubular Bells of programming...



REM:
My Puppy

this is my puppy

her name is Ruby

REM:
Aside on Progamming

imperative pg: computation is statements that change a program state; an algorithm with explicit steps or procedures (e.g. C, BASIC)

declarative pg: logic of computation without flow of control, what the program should achieve not how it achieves (e.g. Prolog)

functional pg: computation as the evaluation of mathematical functions avoiding state and mutable data (e.g. Scheme, Lisp)

OOP: Main concepts

OOP is a programming paradigm that uses “objects” -- data structures consisting of data fields and method and their interactions -- to design apps and program, has six key features:

data abstraction

polymorphism

inheritance

encapsulation

modularity

duck typing

Data Abstraction: Definition

wiki def: simplifying reality by modelling classes of object appropriate to the problem and working at the appropriate level of inheritance for a given aspect of the problem

forces us to see groups of things, an error is not an individual thing, but one of a class of things

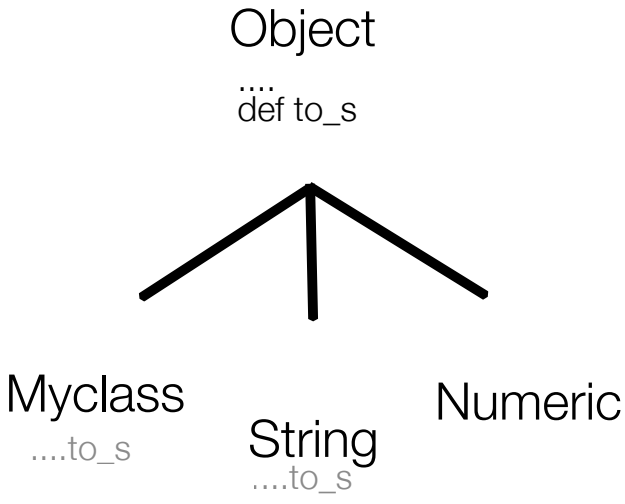
so, animal-methods are dealt with at the animal-object level, and dog-methods are dealt with at the dog-object level

Data Abstraction: Eg

at the object level, Ruby defines all the things you might want to do to Object per se

at the number level, you define the specific things you want to do with Numbers

at the string level, you define the specific...



Polymorphism: Definition

wiki def: is the ability of objects belonging to different data types to respond to calls of methods of the same name according to the appropriate type-specific behaviour

allows us to do the same thing with different objects, because we often want to do this...

we may want an **isa?** or **+**, but recognise that it may do very different things internally for different objects

Polymorphism: Eg

+ adds things together, we all know this

3 **+** 4 gives us 7

“foo” **+** “bar” gives us “foobar”

[:a, :b] **+** [:c, :d] gives us [:a,:b,:c,:d]

and it doesn’t work for hash tables...why ?

the local meaning of **+** is resolved by the objects in the specific context in which they are called

we retain our intuition about how + works

Inheritance: Definition

wiki def: inheritance is a process in which a class inherits all the state and behaviour of another class; sub-classes or children are more specialised versions of the parent class

allows us to be very economic about writing code

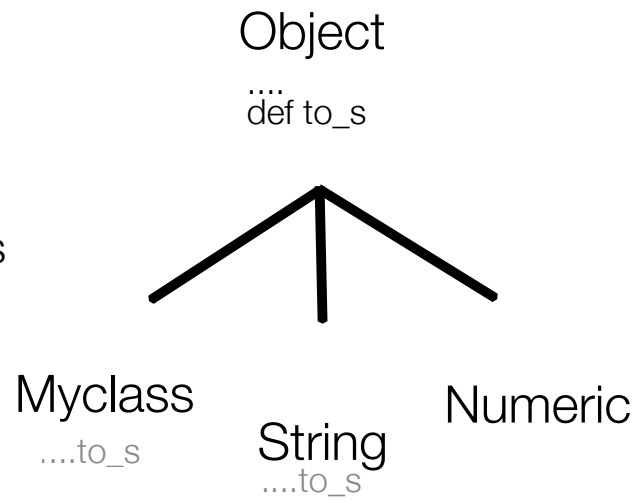
allows us to retain what is common to two objects and yet use what is different; we can define a bunch of methods for birds and then a few specific ones for kiwis

Inheritance: Eg

classic case is **to_s**

but, we will see nice cases in Rails where objects are created as subclasses of say ActiveController, ActiveRecord

slick way to import a huge amount of functionality



Encapsulation: Definition

wki def: encapsulation conceals the details of a class from the objects that sends messages to it

allows us to pass messages between objects without worrying about what goes on inside the object

the methods add_vat may add 21% vat to an item we buy, but I don't need to see this detail (cost * .21) and the programmer can change the internals without affecting my request to add_vat

Encapsulation: Eg

virtual attributes: when you create an object with an attribute, it looks like a fixed aspect of the object

but, they could be virtual and use encapsulation

```
markkean% !! ruby bookprice.rb
Book called: echoes bones
25
bookprice.rb:23: protected method
`real_price'
called for #<Book:0x27b8c
@real_price=15, @name="echoes bones">
(NoMethodError)
```

```ruby
class Book
 attr_accessor :name, :real_price

 def initialize(pl, per)
     @name = pl
     @real_price = per
 end

 def to_s
   "Book called: #{@name}"
 end

 def price
     self.real_price + 10
 end

 protected :real_price
end

bones = Book.new("echoes bones", 15)
puts bones
puts bones.price
puts bones.real_price
```

bookprice.rb

Modularity: Definition

wiki def: a design technique that separates software into separate parts/modules

a module represents a separation of concerns

modules are incorporated into a program via interfaces

a module interface expresses the elements that are provided and required by the module

the elements defined in the interface are detectable by other modules

Duck Typing

in most of our prog, typing is checked at the door of the class, to a degree (e.g., in **obj.method** calls)

above is implicit but can be explicit with **is_a? ; "stringoaaah".is_a? (String) => true**

Ruby-style supports duck-typing, v.flexible typing

Duck-typing

responds_to? allows you check whether any given **obj** can be called with any **method**

looks like cross-class usage and seems a bit lax; but makes sense when you consider the flexibility in Ruby including methods via modules/inheritance

```ruby
class Duck
  def quack
    'Quack!'
  end
  def swim
    'Paddle paddle paddle...'
  end
end

class Goose
  def honk
    'Honk!'
  end
  def swim
    'Splash splash splash...'
  end
end

class DuckRecord
  def quack
    play
  end

  def play
    'Quaaaack!'
  end
end

def make_it_quack(duck)
  duck.quack
end

def make_it_swim(duck)
 duck.swim
end
```

$ puts make_it_quack(Duck.new)
‘Quack!’
$ puts make_it_quack(DuckRecord.new)
‘Quaaaack!’

$ puts make_it_swim(Duck.new)
‘Paddle paddle paddle...’
$ puts make_it_swim(Goose.new)
‘Splash splash splash...’

duck.rb

Duck Typing

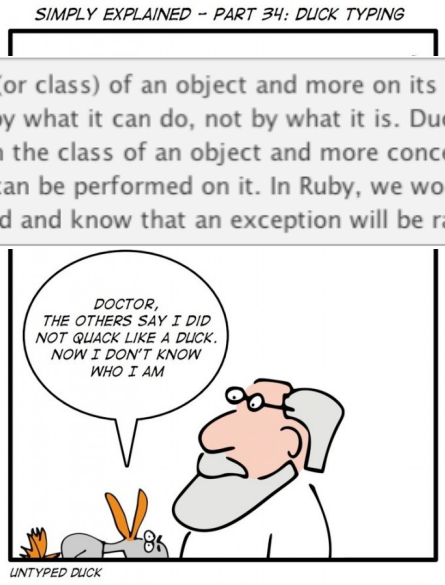
in strongly-typed languages you could not apply **quack** to all sorts of objects

Ruby is more promiscious

if it walks like a duck and quacks like a duck then the object is, a **duck**

an object is more defined by what you can do with it than by its class-type *per se*

Duck Typing



In Ruby, we rely less on the type (or class) of an object and more on its capabilities. Hence, Duck Typing means an object type is defined by what it can do, not by what it is. Duck Typing refers to the tendency of Ruby to be less concerned with the class of an object and more concerned with what methods can be called on it and what operations can be performed on it. In Ruby, we would use respond_to? or might simply pass an object to a method and know that an exception will be raised if it is used inappropriately.

Part B:

let's upgrade things...for Rails

This is a bit historical...

Installing Rails...

Macs now ship with a ruby, rails etc (in which case a **gem update** or **gem update --system** may be sufficient...macports is a good way to go

Also, do not forget the very useful

gem pristine --all

PC-Upgrade: Ruby Gems

Type: gem –v It will return you the gem version.

Then type: *gem list* to see list of installed gems.

Now to update type: *gem update --system*

This should update to latest version of gems.

If any issues go to www.rubygems.org and get one click installer for latest version

this may be wrong...fo PC

PC-Upgrade: Using Rails 4.0

Installing Rails:

Type: *gem install rails* (Once installed type: *rails -v*)

To get SQLite installed go to *www.sqlite.org/download*

From Precompliled Binaries for Windows section download sqlite-dll-win32-x86.zip

Unzip it and copy all files into ruby/bin directory.

Now type: *gem install sqlite3-ruby*

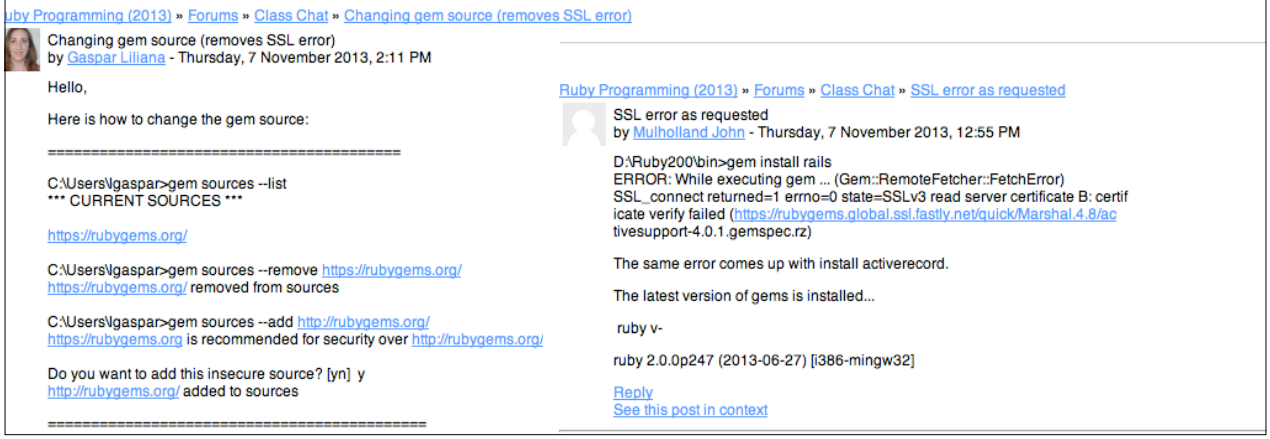
This will install sqlite3 gem to be used by rails applications.

NB PC issues over 32/64 bit versions

PC-Upgrade: Using Rails 4.0

there were two pc errors...
(1) error in installing devkit may arise because of going back to 32-bit version of Ruby 2.0; this may mean re-installing 64-bit version to move on. Means nokogiri will no longer work

(2) second error was with downloading the rails gem; got this SSL error which arose from the wrong source being used; not sure how this arises.



uby Programming (2013) » Forums » Class Chat » Changing gem source (removes SSL error)

Changing gem source (removes SSL error)
by Gaspar Liliana - Thursday, 7 November 2013, 2:11 PM

Hello,

Here is how to change the gem source:

==========

C:\Users\lgaspar>gem sources --list
*** CURRENT SOURCES ***

https://rubygems.org/

C:\Users\lgaspar>gem sources --remove https://rubygems.org/
https://rubygems.org/ removed from sources

C:\Users\lgaspar>gem sources --add http://rubygems.org/
https://rubygems.org is recommended for security over http://rubygems.org/

Do you want to add this insecure source? [yn] y
http://rubygems.org/ added to sources

==========

Ruby Programming (2013) » Forums » Class Chat » SSL error as requested

SSL error as requested
by Mulholland John - Thursday, 7 November 2013, 12:55 PM

D:\Ruby200\bin>gem install rails
ERROR: While executing gem ... (Gem::RemoteFetcher::FetchError)
SSL_connect returned=1 errno=0 state=SSLv3 read server certificate B: certif
icate verify failed (https://rubygems.global.ssl.fastly.net/quick/Marshal.4.8/ac
tivesupport-4.0.1.gemspec.rz)

The same error comes up with install activerecord.

The latest version of gems is installed...

ruby v-

ruby 2.0.0p247 (2013-06-27) [i386-mingw32]

Reply
See this post in context

I checked all sites on Windows, and encountered three issues as followed. I struggled with how to make sqlite3 work for RoR on Windows, and the only solution worked for me is the answer 1 for question 1, which involves installing ruby devkit and a special version of sqlite3. For all sites I tested, you will have problem 3, but the solution is very simple.

1. How do I install sqlite3 for Ruby on Windows?

***** https://www.ruby-forum.com/topic/4413168
http://stackoverflow.com/questions/15480381/how-do-i-install-sqlite3-for-ruby-on-windows

2.Cannot load such file — sqlite3/sqlite3_native (LoadError) on ruby on rails:

Find your sqlite3 gemspec file. One example is /usr/local/share/gem/specifications/sqlite3-1.3.7.gemspec or 'C:\Ruby21\lib\ruby\gems\2.1.0\specifications'.
Change s.require_paths=["lib"] to:
s.require_paths= ["lib/sqlite3_native"]

3.TZInfo::DataSourceNotFound error starting Rails v4.1.0 server on Windows:

In Gemfile, add:
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw]
then update bindle

Regards,
Claire

MacUpgrade I: Safe (Macports)

Check what macport has installed already...

```
port installed ruby19 (or ruby19)
```

The following ports are currently installed:
ruby20 @2.0.0-p247_1 (active)

you may need to track down and destroy old versions in bin , lib and gems

```
sudo port uninstall ruby19
sudo port clean ruby19
```

Then, make sure gems dirs are gone..

```
$ rm -rf /opt/local/lib/ruby
$ rm -rf /opt/local/lib/ruby19
```

MacUpgrade II:

Then start installation....

sudo port install ruby20 +nosuffix

ruby -v => 2.0.0

sudo port install sqlite3

sudo gem install rack

sudo gem install rake

sudo gem install rails

sudo gem install sqlite3

gem list (to check they are all there)

...
rack (1.5.2)
rack-test (0.6.2)
rails (4.0.0)
railties (4.0.0)
rake (10.1.0, 0.9.6)
rdoc (4.0.1, 4.0.0)
sprockets (2.10.0)
sprockets-rails (2.0.0)
sqlite3 (1.3.8)
sqlite3-ruby (1.3.3))
...

The Horror....

sqlite3 (1.3.8)
sqlite3-ruby (1.3.3)
..

None of last year's programs worked...said I needed sqlite3 (1.3.5) but when I tried to **gem install sqlite3** it threw and error saying could not compile it...

- Problem lay in gcc in Xcode 4.5 (now 7.1)

  need to install the latest version of Xcode

  need to install gcc-4.2 from Apple site

  need to install it via Xcode Preferences>Downloads>Command Line Tools

  then fix the links...gcc was pointing to llvm-gcc-4.2

sudo ln -s /usr/bin/llvm-gcc-4.2 **/usr/bin/gcc-4.2**

Part C:

Ruby dealing with databases

Active Record Gem...

part of Ruby's attraction, esp with Rails, is that it works seamlessly with databases

sqlite is bundled from Ruby1.9 onwards and operates off it via the **ActiveRecord** Library

we can put **ActiveRecord**, alongside **Nokogiri** as another useful gem

rem: **gem list** and **gem install activerecord;** or maybe just do **gem install rails**

What ActiveRecord does...

this library allows you to set up a database, define its fields etc

create, destroy and edit its records

query the database

all using ruby-like methods

we will hide the database part

Countries: Open db

```
require 'active_record'

ActiveRecord::Base.establish_connection(
    :adapter => "sqlite3",
    :database  => "memory")
...
```

neatdb.rb

require the libraries

open a connection to sqlite

send user-name and password

Countries: Define tables

```
ActiveRecord::Schema.define do
    create_table :countries do |table|
        table.column :name, :string
        table.column :continent, :string
        table.column :size, :integer
    end

    create_table :regions do |table|
        table.column :country_id, :integer
        table.column :region_size, :integer
        table.column :name, :string
    end
end
...
```

neatdb.rb

two tables

region references the country it is part of

Countries: Create associations

```
class Country < ActiveRecord::Base
    has_many :regions
end

class Region < ActiveRecord::Base
    belongs_to :country
end
...
```

neatdb.rb

Country and **Region** are subclasses of ActiveRecord

describes relationships between models (tables)

here, **has_many** and **belongs_to** define a 1:n

http://en.wikibooks.org/wiki/Ruby_on_Rails/ActiveRecord/Associations

Countries: Create records

```
country = Country.create(:name => 'Ireland',
                         :continent => 'Europe',
                         :size => 84421)
country.regions.create(:region_size => 20000, :name => 'Leinster')
country.regions.create(:region_size => 22000, :name => 'Munster')
country.regions.create(:region_size => 12421, :name => 'Connaght')
country.regions.create(:region_size => 30000, :name => 'Ulster')

country = Country.create(:name => 'Belgium',
                         :continent => 'Europe',
                         :size => 30528)
country.regions.create(:region_size => 21000, :name => 'Walloon')
country.regions.create(:region_size => 9000, :name => 'Flemish')
country.regions.create(:region_size => 528, :name => 'Brussels')
```

neatdb.rb

Countries: Create records

throwaway local var

gets country id

:symbol => val

```
country = Country.create(:name => 'Ireland',
                         :continent => 'Europe',
                         :size => 84421)
country.regions.create(:region_size => 20000, :name => 'Leinster')
country.regions.create(:region_size => 22000, :name => 'Munster')
country.regions.create(:region_size => 12421, :name => 'Connaght')
country.regions.create(:region_size => 30000, :name => 'Ulster')

country = Country.create(:name => 'Belgium',
                         :continent => 'Europe',
                         :size => 30528)
country.regions.create(:region_size => 21000, :name => 'Walloon')
country.regions.create(:region_size => 9000, :name => 'Flemish')
country.regions.create(:region_size => 528, :name => 'Brussels')
```

throwaway local var

key/id created for all

neatdb.rb

Query-ing I

db creating table

```
markkean% ruby neatdb.rb
-- create_table(:countries)
   -> 0.1219s

-- create_table(:regions)
   -> 0.0021s
```

db creating table

```
p Country.all
```

country id

```
#<ActiveRecord::Relation [#<Country id: 1, name: "Ireland",
continent: "Europe", size: 84421>, #<Country id: 2, name:
"Belgium", continent: "Europe", size: 30528>]>

p Region.find(7)

#<Region id: 7, country_id: 2, region_size: 528, name:
"Brussels">
```

country record

Query-ing II

```
markkean% ruby neatdb.rb

p Region.find(:all)
```

country id

```
#<ActiveRecord::Relation [#<Region id: 1, country_id: 1,
region_size: 20000, name: "Leinster">, #<Region id: 2,
country_id: 1, region_size: 22000, name: "Munster">,
#<Region id: 3, country_id: 1, region_size: 12421, name:
"Connaght">, #<Region id: 4, country_id: 1, region_size:
30000, name: "Ulster">, #<Region id: 5, country_id: 2,
region_size: 21000, name: "Walloon">, #<Region id: 6,
country_id: 2, region_size: 9000, name: "Flemish">, #<Region
id: 7, country_id: 2, region_size: 528, name: "Brussels">]>
```

region id

```
p Region.first

#<Region id: 1, country_id: 1, region_size: 20000, name:
"Leinster">
```

region record

Query-ing III

```
markkean% ruby neatdb.rb

out = Country.all
out.each {|place| p place.name}
```

field accessor

```
"Ireland"
"Belgium"
```

ready-made find accessors

```
p Region.find_by_name("Walloon")

#<Region id: 5, country_id: 2, region_size: 21000, name:
"Walloon">
```

ready-made find accessors

```
p Region.find_by_region_size(9000)

#<Region id: 6, country_id: 2, region_size: 9000, name:
"Flemish">
```

region record

Query-ing IV

search for these attributes

```
markkean% ruby neatdb.rb
p Country.where("continent = 'Europe'")
<ActiveRecord::Relation [#<Country id: 1, name: "Ireland",
continent: "Europe", size: 84421>, #<Country id: 2, name:
"Belgium", continent: "Europe", size: 30528>]>
#
```

search, using diff syntax

```
p Country.where("continent = 'Europe' AND size = 30528")
#<ActiveRecord::Relation [#<Country id: 2, name: "Belgium",
continent: "Europe", size: 30528>]>


p Region.where("name like '%ster'")
#<ActiveRecord::Relation [#<Region id: 1, country_id: 1,
region_size: 20000, name: "Leinster">,
#<Region id: 2, country_id: 1, region_size: 22000, name:
"Munster">,
#<Region id: 4, country_id: 1, region_size: 30000, name:
"Ulster">]>
```

What if I do it again...

you may have noticed that if you try to run **neatdb.rb** a second time,it will throw an error about the tables already existing

if you want to clear tables and re-set them you need something slightly different

Several things can be done...

Quick fix is just to test for whether the tables exist...

But, note this will keep creating new records on each iteration into your DB (unless you explicitly destroy all records and start anew each time..)

An alternative is to explicitly drop tables and then rebuild them...

Countries: Define tables

```
if (Country.table_exists? || Region.table_exists?)
  puts "table exists"
  else  ActiveRecord::Schema.define do
          create_table :countries do |table|
            table.column :name, :string
            table.column :continent, :string
            table.column :size, :integer
        end
          create_table :regions do |table|
            table.column :country_id, :integer
            table.column :region_size, :integer
            table.column :name, :string
    end
  end
end
```

neatdb.rb

Several things can be done...

Quick fix is just to test for whether the tables exist...

But, note this will keep creating new records on each iteration into your DB (unless you explicitly destroy all records and start anew each time..)

An alternative is to explicitly drop tables and then rebuild them...

```
require 'active_record'

ActiveRecord::Base.establish_connection(
    :adapter => "sqlite3",
    :database  => "memory")

class Clean < ActiveRecord::Migration
   def self.up
    create_table :towns do |table|
        table.column :name, :string
        table.column :continent, :string
        table.column :size, :integer
     end
   end

   def self.down
     drop_table :towns
   end

end

class Town < ActiveRecord::Base
end

#Clean.down
Clean.up

p Town.create(:name => "ff", :continent => "fa", :size => 333)
```

cleandb.rb

```
$ ruby cleandb.rb
-- create_table(:towns)
   -> 0.0138s
#<Town id: 1, name: "ff", continent: "fa", size: 333>

# after you comment out Clean.down after 1st run

$ ruby cleandb.rb
-- drop_table(:towns)
   -> 0.0088s
-- create_table(:towns)
   -> 0.0027s
#<Town id: 1, name: "ff", continent: "fa", size: 333>
```

Part D:

more chips off the old...

Blocks, Anon Fns and Yield

we have seen how blocks are important in Ruby

with **each**, **collect** and **inject** we pass in blocks to do various tasks on a given object (e.g., arrays)

coming from Lisp, you would say blocks look a bit like spaghetti-western fns

so they can be passed around assigned to variables as anonymous fns and closures

REM: Yield

Anything in curly brackets or between **do...end** is a block

A block of code to be bandied around

like Lisp's anon-fns

but, can be more

```
def block_eg
  puts "this is the first message"
  yield
  puts "this is the middle message"
  yield
  puts "this is the last\n\n"
end

block_eg {puts "-------CUT HERE-------"}

def block_with_args
  puts "First we say this"
  yield("CUT", "HERE")
  puts "this is the bit we cut out"
  yield("CUT", "AGAIN HERE")
  puts "this is the last bit"
end

block_with_args {|a, b| puts "---#{a} #{b}---"}
```

the block

the block with args

block.rb

there is something too scary & magical about **yield**

Blocks & Lambda I

so, as you would expect, a block is an object

of what type? of type **Proc**

so, this works...

```
ablock = Proc.new {|x| puts x}
```

but, preferred way to create blocks is using **Kernel#lambda** method

```
ablock = lambda {|x| puts x}
```

let's make them a bit more comprehensible...

Blocks & Lambda II

remember, I said blocks sort-of allow you to swap arbitrary methods into the body of an existing method

if you think of blocks as objects that can be assigned to variables

then, it is a short step to start passing them as arguments to methods (reducing the magic of **yield**)

Ruby, can do this...using **&arg** and **call**

consider examples...

Eg 1a

```
def block_eg1(cutblock)
  puts “this is the first message”
  cutblock.call
  puts “this is the middle message”
  cutblock.call
  puts “this is the last\n\n”
end

ablock = lambda {puts “-------CUT HERE-------”}
block_eg1(ablock)
```

cutblock.rb

```
ruby cutblock.rb
this is the first message
-------CUT HERE-------
this is the middle message
-------CUT HERE-------
this is the last
```

Eg 1a

block bound to cutblock

block called

```
def block_eg1(cutblock)
  puts “this is the first message”
  cutblock.call
  puts “this is the middle message”
  cutblock.call
  puts “this is the last\n\n”
end

ablock = lambda {puts “-------CUT HERE-------”}
block_eg1(ablock)
```

the block

cutblock.rb

```
ruby cutblock.rb
this is the first message
-------CUT HERE-------
this is the middle message
-------CUT HERE-------
this is the last
```

Eg 1b

```
def block_eg1(&cutblock)
  puts “this is the first message”
  cutblock.call
  puts “this is the middle message”
  cutblock.call
  puts “this is the last\n\n”
end

block_eg1() {puts “-------CUT HERE-------”}
```

cutblock.rb

```
ruby cutblock.rb
this is the first message
-------CUT HERE-------
this is the middle message
-------CUT HERE-------
this is the last
```

Eg 1b

throws error without &

proc’s what is passed

block called

nb. 0 arg method

```
def block_eg1(&cutblock)
  puts “this is the first message”
  cutblock.call
  puts “this is the middle message”
  cutblock.call
  puts “this is the last\n\n”
end

block_eg1() {puts “-------CUT HERE-------”}
```

cutblock.rb

```
ruby cutblock.rb
this is the first message
-------CUT HERE-------
this is the middle message
-------CUT HERE-------
this is the last
```

Eg 2

```
low_vat_rate = 10
high_vat_rate = 21

blocktest = lambda do |x|
 if x > 100 then p(x * low_vat_rate)
 elsif x < 100 then p(x * high_vat_rate)
 elsif x = 100 then
        puts "not sure what to do" end
 end

blocktest.call(20)
blocktest.call(100)
blocktest.call(500)

# ruby blocko.rb
# 420
# not sure what to do
# 5000
```

blocko.rb

Eg 2

block made

block called

block has arg

block uses vars

```
low_vat_rate = 10
high_vat_rate = 21

blocktest = lambda do |x|
 if x > 100 then p(x * low_vat_rate)
 elsif x < 100 then p(x * high_vat_rate)
 elsif x == 100 then
        puts "not sure what to do" end
 end

blocktest.call(20)
blocktest.call(100)
blocktest.call(500)

# ruby blocko.rb
# 420
# not sure what to do
# 5000
```

blocko.rb

Blocks, Lambda & Closures

Closure (lexical closure, function closure or function value) is a function together with a referencing environment for the non-local variables of that function

A closure allows a function to access variables outside its typical scope. Such a function is said to be "closed over" its free variables.

The referencing environment binds the nonlocal names to the corresponding variables in scope at the time the closure is created, additionally extending their lifetime to at least as long as the lifetime of the closure itself.

When the closure is entered at a later time, possibly from a different scope, the function is executed with its non-local variables referring to the ones captured by the closure.

Example 1

```
no = 0
closures = []
(0..7).each { |n|  closures << lambda { n + no } }
p closures
p closures.map { |c| c.call }
no = 3
p closures.map { |c| c.call }
```

closureo.rb

```
$ ruby closure0.rb
[#<Proc:0x007fa81904e430@closure0.rb:18 (lambda)>,
#<Proc:0x007fa81904e3e0@closure0.rb:18 (lambda)>,
#<Proc:0x007fa81904e390@closure0.rb:18 (lambda)>,
#<Proc:0x007fa81904e340@closure0.rb:18 (lambda)>,
#<Proc:0x007fa81904e2f0@closure0.rb:18 (lambda)>,
#<Proc:0x007fa81904e2a0@closure0.rb:18 (lambda)>,
#<Proc:0x007fa81904e250@closure0.rb:18 (lambda)>,
#<Proc:0x007fa81904e200@closure0.rb:18 (lambda)>]
[0, 1, 2, 3, 4, 5, 6, 7]
[3, 4, 5, 6, 7, 8, 9, 10]
$
```

Can Closures be Smoked?

Closures remind us that functional programming can pass blocks/functions/methods as arguments to other method/functions

Closures are handy for returning to early stages of processing in a program (callbacks) but only if you pass the values to local variables established in the scope of the closure

Closures have different properties to blocks, though they look like them

Passing closures as objs

```
def complement(fn)
  lambda {|arg| not fn.call(arg) }
end

is_even = lambda {|n| n % 2 == 0 }
is_odd  = complement(is_even)

p is_even.call(1)
p is_even.call(4)

p is_odd.call(1)
p is_odd.call(4)
```

passing.rb

```
# false
# true
# true
# false
```

Passing closures as objs

```
def complement(fn)
  lambda {|arg| not fn.call(arg) }
end

puts "\nAnd then the string one..."

is_string = lambda {|n| n.instance_of?(String) }
is_not_string  = complement(is_string)

p is_string.call("hello")
p is_string.call(:hello)
p is_not_string.call("hello")
p is_not_string.call(:hello)
```

passing.rb

```
# true
# false
# false
# true
```

Creating new methods dynamically

```
def compose (f, g)
  lambda {|args| f.call(g.call(args)) }
end

mult2 = lambda {|n| n*2 }
add1  = lambda {|n| n+1 }
mult2_add1 = compose(add1, mult2)

p mult2_add1.call(3)
```

composure.rb

```
# 7
```

Part D:

Half way through look back...and questions

Covered...

main Ruby types: strings, symbols, arrays, hashes

defining classes, modules and use of inheritance/mixins

looked at flow of control, iterative and logic constructs; variables and I/O

how to access the web to scrape info from pages (using Nokogiri and Regexp)

how to set-up and interface with a database

seen a number of largish programs: iTunes, SixDegrees

next...move on to Ruby on Rails