Learning Ruby

Ruby Code Blocks

• Code blocks in Ruby are not like “blocks of code” in other programming languages
• Blocks are typically associated with method calls
• Blocks can implement "iterators"
• Blocks can be passed around from method to method
Block Examples Seen So Far

3.times { print "Ho! " }

1.upto( 5 ) { |i| puts i }

my_a.each { |element| puts element if element.length > 4 }

classic_rock.each_with_index { | song, where |
  puts "#{song} performs '#{where}'."
}

How Code Blocks Work

• The code within { .. } or do .. end is passed to the method (as if it were another parameter)
• The method then arranges to call the block of code as required
• It is possible to write custom methods that take advantage of this Ruby feature
• The "yield" method will call an associated code block from within a method
More Block Examples

```ruby
[ 'a', 'e', 'i', 'o', 'u' ].each { |ch| puts ch }

( 'f'..'t' ).each { |ch| print ch }

p [2, 3, 5, 7, 11,13,17,19].find_all { |prime| prime*prime > 30 }  # [7, 11, 13,17, 19]
returns an array containing all elements for which block is true
p calls the inspect method (printing arrays nicely)

kids = %w( joseph aaron aideen )
%w is non-interpolated array of words, separated by whitespace
kids.each { |child| puts "Child is #{child}\n" }
```

Dictionary Moment

**in·ter·po·late**

–verb (used with object)
1. to introduce (something additional or extraneous) between other things or parts; interject; interpose; intercalate.
2. Mathematics. to insert, estimate, or find an intermediate term in (a sequence).
3. to alter (a text) by the insertion of new matter, esp. deceptively or without authorization.
4. to insert (new or spurious matter) in this manner.
Ruby Control Flow

If else end

puts 'I am a fortune-teller. Tell me your name:'
name = gets.chomp
    # OR name= gets
    # name.strip!
if name == 'Chris'
    puts 'I see great things in your future.'
else
    puts 'Your future is... Oh my! Look at the time!'
    puts 'I really have to go, sorry!'
While loops

command = ''
while command != 'bye'
    puts command
    command = gets.chomp
end
puts 'Come again soon!'

Methods

• We’ve seen several methods so far: puts, gets, chomp, to_s, and even +, -, *, and /
• Usually use dot notation
• 5 + 5 is really just a shortcut way of writing 5.+ 5
• ‘pit’ * 5
String methods

lineWidth = 50
puts('Old Mother Hubbard'.center(lineWidth))

lineWidth = 40
str = ' --> text <-- '
puts str.ljust lineWidth    #parens are optional

Random Numbers

rand    --> float greater than or equal to 0.0 and
         less than 1.0.
rand(5)  --> int in range 0..4
srand 1776  gives the random number
generator a seed
puts(Math::PI)  #:: is scope resolution
Defining own methods

def sayMoo numberOfMoos
    puts 'mooooooo...'*numberOfMoos
end

# Note local variables
def doubleThis num
    numTimes2 = num*2
    puts num.to_s+' doubled is '+numTimes2.to_s
end

doubleThis 44
puts numTimes2.to_s # ERROR undefined variable

---

Defining own methods

def sayMoo numberOfMoos
    puts 'mooooooo...'*numberOfMoos
end

numTimes2 = 0 # defined
def doubleThis num
    numTimes2 = num*2 # local scope
    puts num.to_s+' doubled is '+numTimes2.to_s
end

doubleThis 44   -> 44 doubled is 88
puts numTimes2.to_s   -> 0
Makes local copy

def littlePest var
  var = nil
  puts 'HAHA! I ruined your variable!'
end
var = 'You can\'t even touch my variable!’
littlePest var
puts var

Local variables are local to the code construct in which they are declared. For example, a local variable declared in a method or within a loop cannot be accessed outside of that loop or method. Local variable names must begin with either an underscore or a lower case letter.

defined? tells the scope

arg = 5
numTimes2 = 0
def doubleThis num
  numTimes2 = num*2
  p defined? num ->"local-variable"
  p defined? arg ->nil
  puts num.to_s+' doubled is '+numTimes2.to_s
end
Ruby Methods are Easy!

```ruby
def addThem ( first, second )
  first + second
  # The value of the last statement in the method is value returned, but you can also use "return"
end # of addThem.

puts addThem( 23, 6 )  # 29

def addThemAll ( first, *rest )
  tot=first
  rest.each{|val|  tot +=val}
  tot  # or return tot
end
p addThemAll( 1, 2, 3, 4, 5, 6, 7 )  # 28
rest is of type Array. The "*"parameter must be the last item
```

Scope

- **Local variables** (begin with either an underscore or a lower case letter) and are local to the code construct in which they are declared. For example, a local variable declared in a method or within a loop cannot be accessed outside of that loop or method.
- **Global variables** (begin with $) are accessible from anywhere in the Ruby program, regardless of where they are declared.
- A **class variable** (begins with @@) is shared amongst all instances of a class.
- **Instance variables** (begin with @) are similar to Class variables except that their values are local to specific instances of an object.
Blocks Scope - closures

```ruby
sum = 0
(1..10).each { |i| sum += i}
p sum -> 55 # available as known before

(1..10).each { |i| last=i}
p last -> undefined local variable or method ‘last’
```

Yield – calls provided block

```ruby
def yup
  yield
  yield
  yield
end
yup {puts "where is the beef?")
```
At Seats: what is output?

def test
    msg = "OF COURSE"  # local variable to test
    puts "You are in the method"
    yield
    puts "You are again back to the method"
    yield
end
msg = "HOPEFULLY"  # different local variable

Output - Notice Scope

• You are in the method
• You are in the block
• You are again back to the method
• You are in the block
Yield – reusing code (block can have parameters)

def triple(max)
    yield 3*max
end

triple(8) { |x| print x} ->24

Yield – reusing code what is output?

def cubes(max)
    i=1
    while i < max
        yield i**3
        i += 1
    end
end
cubes(8) { |x| print x, "\n"}

sum = 0
cubes(8) { |y| sum += y}
print "\nsum=",sum

product = 1
cubes(8) { |z| product *= z}
print "\nproduct=",product

Scope of block passed is calling block not called block.
sum is NOT known inside cubes but is known inside code block.
def cubes(max):
    i = 1
    while i < max
        yield i**3
        i += 1
    end
end

cubes(8) { |x| print x, "", }

sum = 0
cubes(8) { |y| sum += y}
print "\n\n\nsum=",sum

product = 1
cubes(8) { |z| product *= z}
print "\n\n\nproduct=",product

def compute( what )
    if block_given?
        yield( what )
    else
        what
    end # of if.
end # of compute

puts compute( 22 ) 22
puts compute( 11 ) { |num| num*num } 121
puts compute( 4 ) { |num| num+num } 8
Implicitly passing and invoking blocks

```ruby
def foo(*args)
    yield(args.join(' '))
end

foo('oscar', 'Meyer', 'wiener'){|name|
    puts "Hello #{name}"}
```

- Hello oscar Meyer wiener

Explicitly passing, binding and invoking blocks

```ruby
def foo(*args, &blk)
    blk.call(args.join(' '))
end

foo('oscar', 'Meyer', 'wiener'){|name|
    puts "Hello #{name}"}
```

- Hello oscar Meyer wiener

The & binds the block to the variable blk making it available as a Proc object.
Explicitly passing, binding and invoking blocks via a block variable

def foo(*args)
    blk = args.delete_at(-1)
    blk.call(args.join(' '))
end

the_block = lambda { |name| puts "Hello #{name}" }

foo('oscar', 'Meyer', 'wiener', the_block) -> Hello oscar Meyer wiener

This is how each could be implemented via yield

self – current class “this”

class Array # Note how I can extend the functionality of built-in classes
def my_each()
    0.upto(length-1) do |i|
        yield self[i] \ call to provided code with my ith element
    end
    return self \ if want to use in assign statement or nest calls
end

puts "Array\n"
t = ["quit", "felt", "worry", "smith", "white", "abandon", "fish", "this", "oboe"]
t.my_each{|i| print i, "\n" } 
p t.my_each{|i| print "The element is ", i, "\n" }.sort! \ needs
    my_each to return the array
class NumericSequences
  def fib(limit)
    yield 0, 0
    yield 1, 1
    prev = 0
    curr = 1
    i=2
    while (i <= limit)
      new = prev + curr
      yield i, new
      prev = curr
      curr = new
      i = i+1
    end
  end
end

What does this do?

Using yield to print Fibonacci numbers

class NumericSequences
  def fib(limit)
    yield 0, 0
    yield 1, 1
    prev = 0
    curr = 1
    i=2
    while (i <= limit)
      new = prev + curr
      yield i, new
      prev = curr
      curr = new
      i = i+1
    end
  end
end

Using yield to print Fibonacci numbers

g = NumericSequences.new 
  Creates instance of class

g.fib(10) { |i,x| print "The #{i}th Fibonacci number: #{x}\n" }

← allows two arguments to be used
Return multiple values

def give_back ( a, *b )
    return a
end # of give_back.

def give_back2 ( a, *b )
    return a, b
end

first, rest = give_back2( 1, 2, 3, 4, 5 )

Creating a class

class Die
    def roll
        1 + rand(6)
    end
end # Let's make a couple of dice...
dice = [Die.new, Die.new]
# ...and roll them.
dice.each do |die|
    puts die.roll
end
Better to have a variable to store the current value
class Die
  def initialize #When an object is created,
    # its initialize method (if it has one defined) is always called.
    # I'll just roll the die.
      roll
    end
  def roll
    @numberShowing = 1 + rand(6)
  end
  def showing
    @numberShowing
  end
end
die = Die.new
die.roll
puts die.showing

class Die
  attr_reader :numberShowing
  attr_writer :timesRolled
  def initialize #constructor
    @timesRolled = 0
  end
  def roll
    @timesRolled+=1
    @numberShowing = 1 + rand(6)
  end
  def getValue
    @numberShowing
  end
  def printIt
    puts "Die " + @numberShowing.to_s + " times rolled:" + @timesRolled.to_s
  end
end
die = Die.new
puts die.getValue
puts die.numberShowing # access local variables
die.timesRolled = 5
die.printIt  #-> Die 6 times rolled:5
Closure – method plus scope

The ability to take a block of code (code in between do and end), wrap it up in an object (called a proc), store it in a variable or pass it to a method, and run the code in the block whenever you feel like. So it's kind of like a method itself, except that it isn't bound to an object (it is an object), and you can store it or pass it around like you can with any object.

```ruby
msg = "NICE 
toast = Proc.new do
  puts 'Cheers!' + msg
end

def doit toast
  msg = "yes"
  toast.call
  toast.call
  toast.call
end
doit toast
```

Procs with arguments

```ruby
doYouLike = Proc.new do |aGoodThing|
  puts 'I *really* like '+aGoodThing+'!
end

doYouLike.call 'chocolate'
doYouLike.call 'ruby'  # the call is required?

Why not just use methods? Well, it's because there are some things you just can't do with methods. In particular, you can't pass methods into other methods (but you can pass procs into methods), and methods can't return other methods (but they can return procs). This is simply because procs are objects; methods aren't.
```
def doSelfImportantly someProc
    puts 'Everybody just HOLD ON! I have something to do...'
someProc.call
    puts 'Ok everyone, I\'m done. Go on with what you were doing.'
end

sayHello = Proc.new do
    puts 'hello'
end

sayGoodbye = Proc.new do
    puts 'goodbye'
end

doSelfImportantly sayHello
doSelfImportantly sayGoodbye

Passing a Class

def create_from_factory(factory)
    factory.new
end

obj = create_from_factory(Array)
Try at seats
"99 bottles of beer on the wall..." Write a program which prints out the lyrics to that beloved classic, that field-trip favorite: "99 Bottles of Beer on the Wall."

Extending built-in classes – truly object oriented

```ruby
class Integer
  def is_odd
    odd = self % 2 == 1
    end
end

p 10.is_odd #=>false
```

```ruby
class Integer
  def is_perfect_square?
    root = Math.sqrt(self).to_i
    #puts root.to_s
    root * root == self
  end
end

puts 10.is_perfect_square?
puts 9.is_perfect_square?
```