"ADVANCED RUBY" COURSE NOTES

INTRODUCTION

This document contains brief notes to accompany the *Advanced Ruby Programming* Course. They are intended to provide extra help when learning or revising the topics in each step. For a more in-depth explanation of the subjects covered in the course, it is recommended that students refer to the appropriate chapter in *The Book Of Ruby* (the PDF book supplied with this course). The 'Advanced Ruby' course is self-contained, but the book provides more detail.

THE SOURCE CODE ARCHIVE

You should download the complete code archive accompanying the Book and this course. This archive is available on the course page.

THE LITTLE BOOK OF RUBY

This course assumes that you are already comfortable with the fundamental features of Ruby programming. If you need to revise the basics, use my free eBook, *The Little Book Of Ruby*. This eBook and its accompanying source code archive may be downloaded from the *Extras* chapter on the course page.

THESE NOTES

These course notes provide some useful information relating to each of the ten steps in this course. When code samples from the archive are given in these notes, the Ruby file name is shown above the code like this:

A_sample_ruby_program.rb

Program code is shown like this:

```
if x == 1 : puts( 'ok' ) end    # This is a comment
```

STEP 1

See: Chapter 6 of The Book Of Ruby

CONDITIONAL STATEMENTS

if..then

Here is a simple example of an **if** test…

if_then.rb

```
if x==1 then
   puts( 'ok' )
end
```

When an `if` test and the code to be executed are placed on separate lines, the **then** keyword is optional. When the test and the code are placed on a single line, the `then` keyword is obligatory:

```
if x == 1 then puts( 'ok' ) end    # with 'then'
if x == 1 puts( 'ok' ) end         # syntax error!
```

In Ruby 1.8, a colon character (`:`) was permitted as an alternative to **then**. This syntax is not supported in Ruby 1.9 or Ruby 2.0:

```
if x == 1 : puts( 'ok' ) end    # This works with Ruby 1.8 only
```

if..elseif

You can use multiple `elsif` sections after `if`..

days2.rb

```
if i == 1 then puts("It's Monday" )
elsif i == 2 then puts("It's Tuesday" )
elsif i == 3 then puts("It's Wednesday" )
elsif i == 4 then puts("It's Thursday" )
elsif i == 5 then puts("It's Friday" )
elsif (6..7) === i then puts( "Yippee! It's the weekend! " )
else puts( "That's not a real day!" )
end
```

and . . or . . not

Ruby has two different syntaxes for testing Boolean (*true/false*) conditions:

1) `and`, `or`, and `not`.
2) `&&` (and), `||` (or), and `!` (not).

Be careful, though: The two sets of operators aren't interchangeable. They have different *precedence*, which means that when multiple operators are used in a single test, the parts of the test may be evaluated in different orders depending on which operators you use. As a general rule, don't mix them and, if in doubt, use parentheses to avoid ambiguity:

boolean_ops.rb

```
if (( 1==3 ) and (2==1)) || (3==3) then
   puts('true')
else
   puts('false')
end
```

Negation

The normal negation operator is the exclamation mark (`!`). It means "not". The negation operator can be used at the start of an expression; as an alternative, you can use the `!=` ("not equals") operator between the left and right sides of an expression:

```
!(1==1)       #=> false
1!=1          #=> false
```

unless

Ruby also can also perform `unless` tests, which are the exact opposite of `if` tests:

unless.rb

```
unless aDay == 'Saturday' or aDay == 'Sunday'
   daytype = 'weekday'
else
   daytype = 'weekend'
end
```

Think of `unless` as being an alternative way of expressing "if not."

Case Statements

When you need to take a variety of different actions based on the value of a single variable, multiple `if..elsif` tests are verbose and repetitive. A neater alternative is provided by a **case** statement.

case.rb

```
case( i )
   when 1 then puts("It's Monday" )
   when 2 then puts("It's Tuesday" )
   when 3 then puts("It's Wednesday" )
   when 4 then puts("It's Thursday" )
   when 5 then puts("It's Friday" )
   when (6..7) then puts( "Yippee! It's the weekend! " )
   else puts( "That's not a real day!" )
end
```

STEP 2

See: Chapter 7 of The Book Of Ruby

CLASSES, METHODS AND SINGLETONS

Class Methods

Syntax of class method:

```
<ClassName>.<methodName>
```

Example:

```
class MyClass
  def aMethod     # instance method
    puts "hello"
  end

  def MyClass.aClassMethod # class method
    puts "goodbye"
  end
end

ob = MyClass.new
ob.aMethod
MyClass.aClassMethod    # calling class method
```

Class Variables, Instance Variables, Instance Variables of a Class

This is a class variable: `@@class_variable`
This is an instance variable: `@instance_var`

A class can also have instance variables *of the class object*. Whereas in the class method, I can print the value of the class's instance variable because it belongs to the class *object* itself. But any attempt to print the instance variable of an object that is created from the class fails.

```
class MyClass
  @@class_variable = 1
  @instance_variable_of_class = 100

  def initialize
    @instance_var = 5000
  end

  def aMethod
    p( @@class_variable )
    p( @instance_var )
    p( @instance_variable_of_class )
  end

  def MyClass.aClassMethod
    p( @@class_variable )
    p( @instance_var )
    p( @instance_variable_of_class )
  end
end

ob = MyClass.new
puts( "---ob.aMethod---" )
ob.aMethod
puts( "---MyClass.aClassMethod---" )
MyClass.aClassMethod
```

Singletons

Ruby lets you create methods that belongs to a single object rather than to a class or to all the objects created from a class. The most common type of singleton methods are class methods. Class is an object. The class methods are methods that belong to that single class object.

Example:

```
werewolf = Creature.new( "growl" )

def werewolf.howl
  if FULLMOON then
    puts( "How-oo-oo-oo-oo!" )
  else
    talk
  end
end

werewolf.howl
```

You can also add methods by creating a nameless class and adding it to a specific object.

Example:

```
ob = Object.new

# singleton class
class << ob
  def blather( aStr )
    puts("blather, blather #{aStr}")
  end
end

ob.blather( "weeble" )
```

STEP 3

See: Chapter 8 of The Book Of Ruby

PASSING ARGUMENTS AND RETURNING VALUES

Basic Syntax

You may use or omit parentheses when defining and when calling methods.

Example:

```
def m1
  puts "hello"
end

def m2( anArg )
  puts anArg
end

def m3( anArg, anotherArg )
  puts( anArg )
  puts( anotherArg )
end

def m4 anArg, anotherArg
  puts anArg
  puts anotherArg
end

m1
m1()
m2 "hello"
m2( "goodbye" )
m3 "hi", "welcome"
m3( "good day", "good evening" )
m4( "good morning", "good night" )
m4 "toodle-pip!", "tinkerty-tonk!"
```

RETURN VALUES

All methods return value – the last expression evaluated by the function. The `return` keyword is optional.

Example:

```
def method1
  a = 1
  b = 2
  c = a + b
end

def method2
  a = 1
  b = 2
  c = a + b
  return b
end

def method3
  "hello"
end

def method4
  a = 1 + 2
  "goodbye"
end

def method5
end

p( method1 )       #=> 3
p( method2 )       #=> 2
p( method3 )       #=> "hello"
p( method4 )       #=> "goodbye"
p( method5 )       #=> nil
```

If no value is explicitly returned, the result of last expression evaluated by the function is returned. The `return` keyword is optional.

When no object is specifically returned or no expression is evaluated, a nil value is returned.

If you return an argument list separated by commas, Ruby will in fact return these as an array.

You can optionally provide default values for arguments like this:

```
def aMethod( a=10, b=20, c=100, *d )
```

Here the `*d` argument creates an array of any 'trailing' values.

You can assign multiple values to multiple variables using 'parallel assignment' like this:

```
s1, s2, s3 = "Hickory", "Dickory", "Dock"
```

ENCAPSULATION AND INFORMATION HIDING

Encapsulation refers to the grouping together of an object's "state" (its data) and the operations that may alter or interrogate its state (its methods). Information hiding refers to the fact that data is sealed off and can be accessed only using well-defined routes in and out—in object-oriented terms, this implies "accessor methods" to get or return values. In procedural languages, information hiding may take other forms; for example, you might have to define interfaces to retrieve data from code "units" or "modules" rather than from objects. In object-oriented terms, encapsulation and information hiding are almost synonymous— true encapsulation necessarily implies that the internal data of an object is hidden. However, many modern object-oriented languages such as Java, C#, C++, and Object Pascal are quite permissive in the degree to which information hiding is enforced (if at all).

In addition to hiding (making private) the data inside an object, it is also a sound principle to ensure that changes to the implementation of any methods do not have side-effects on variables and data outside the object that owns that method. For example, if you send a variable `x` to a method `y()` and the value of `x` is changed inside `y()`, you should not be able to obtain the changed value of `x` from outside the method—*unless the method explicitly returns that value*.

It turns out that there are occasions when arguments passed to a Ruby method can be used like the "by reference" arguments of other languages (that is, changes made inside the method may affect variables outside the method). This is because some

Ruby methods modify the original object rather than yielding a value and assigning this to a new object. For example, there are some methods ending with an exclamation mark that alter the original object. Also the << string concatenation operator. This gives a programmer the opportunity to use the changed value of arguments sent to a method rather than using the explicitly returned vales. If the implementation of that method is altered, this could have unintended side-effects on any code that calls that method.

Try this simple example to appreciate the potential problems that might arise if a programmer relies on changes to the arguments passed to a method rather than (as the author of the method intended) using the return values. Create a new Ruby program, say a file called *test.rb*, and enter this code:

```
class X
    def change( anArg )
        anArg.reverse!()
        return anArg.reverse!() << "!!!"
    end
end


ob = X.new
greeting = "Hello"
puts( ob.change( greeting ) )
puts( greeting )
```

Notice that the `change()` method uses both the exclamation-marked `reverse!` method and the << concatenation method. Both these change the 'receiver object' (the object to their left) rather than returning a new object. In the code we have two examples of how this method may be used:

a) As it was intended to be used – the programmer sends an argument, `greeting`, to `change()` and displays the returned value:

```
puts( ob.change( greeting ) )
```

And this is the result:

```
Hello!!!
```

b) As it was *not* intended - the programmer uses the changed value of the variable, `greeting`, rather than using the value returned by the method:

```
puts( greeting )
```

And this is the result:

```
Hello!!!
```

As the results are the same, you may think there is no problem here. But what if the author of the method decides to change that method's implementation? He or she may assume that there is no problem in doing this because the value returned by the method remains the same. Make this simple change. In the final line of the method, remove the exclamation mark from `reverse!()`. It should now look like this:

```
return anArg.reverse() << "!!!"
```

The code of programmer (a) – who uses the return value, remains unaffected:

```
puts( ob.change( greeting ) )
```

And this is the result:

```
Hello!!!
```

But the code of programmer (b) is profoundly affected by even this tiny change:

```
puts( greeting )
```

This is now the result:

```
elloH
```

Be aware of the dangers of using methods that modify the object on which they are used (the '*receiver*' object) rather than returning new objects. They may accidentally

expose the internal implementation details of your classes so that code outside your objects may be able to rely on 'side effects' of your code.

STEP 4

See Chapter 9 of The Book Of Ruby

EXCEPTION HANDLING

Handling errors

If your code may cause an error, protect it inside an Exception-handling block. Here is the basic syntax:

```
begin
      # Some code which may cause an exception
rescue Exception
      # Code to recover from the exception
end
```

rescue

You may have multiple **rescue** blocks to handle different errors. Put the most specific Exception-handling blocks first. Any that are not handled by those will be handled by the more generic **rescue** blocks later on.

Example:

```
begin
      # Some code which may cause an exception
rescue TypeError, NoMethodError
      # Code to recover from TypeError or NoMethodError exception
rescue Exception
      # Code to recover from all other exceptions
end
```

ensure

Use an `ensure` block to contain code that executes whether or not an exception occurs. Use an `else` block to contain code that only executes when no exception occurs:

Example:

```
begin
      # Some code which may cause an exception
rescue Exception
      # Code to recover from the exception
else
      # Code executes when no exception occurs
ensure
      # Code always executes
end
```

STEP 5

BLOCKS, PROCS AND LAMBDAS

See Chapter 10 of The Book of Ruby

Summary

- Blocks are not just 'chunks' of code (as the term is used in some other languages)
- Blocks are nameless methods – you can pass arguments into a block
- Blocks are used in iterators, to do some operation a certain number of times
- They may be delimited by curly brackets or by `do` and `end`.

Blocks are executable code contained between either curly braces `{` and `}` or `do` and `end`. They can be thought of as nameless functions. They are often used as iterators. For example, arrays, integers and ranges may use methods to pass a series of values into a block. These values are received in the form of 'block parameters' which are placed between a pair of upright bars.

Example:

```
3.times do |i|
    puts( i )
end

3.times { |i|
    puts( i )
}

[1,2,3].collect{|x| x*2}
```

Creating block objects

If you want to create Blocks as objects, use the `Proc` class or the `proc()` or `lamba()` methods.

Example:

proc_create.rb

```
a = Proc.new{|x| x = x*10; puts(x) }      #=> Proc
b = lambda{|x| x = x*10; puts(x) }        #=> Proc
c = proc{|x| x.capitalize! }              #=> Proc
```

Methods that operate on blocks such as the `each` and `collect` methods don't take blocks as arguments in the normal way. It would be an error to try to pass a block to a method in parentheses, for example. The block must directly follow the method name. Even so, you can use blocks with your own methods.

Yielding blocks

The way to do it is to just put the block after the method-call – just as you would with Ruby's `each` and `collect` methods. The problem now arises that, since the method being called, does not receive the block like a normal argument – in its argument list – how does it execute the code in the block? the answer is, it yields control to that block using the `yield` keyword.

4blocks.rb

```
def aMethod
      puts('--- In aMethod ---' )
      yield
end

aMethod{ puts( "Good morning" ) }


def caps( anarg )
      puts('--- In caps method ---' )
      yield( anarg )
end

caps( "a lowercase string" ){ |x| x.capitalize! ; puts( x ) }

puts( "And now a block within a block..." )
# a block within a block
["hello","good day","how do you do"].each{
      |s|
      caps( s ){ |x| x.capitalize!
            puts( x )
      }
}
```

Curly bracket-delimited blocks have higher precedence than `do..end` delimited blocks. Se the sample program: *precedence.rb.*

STEP 6

See Chapters 11 and 17 of The Book Of Ruby

SYMBOLS, THREADS AND FIBERS

> **NOTE**: *Chapter 11 of The Book Of Ruby* (Symbols) is provided as a free download – refer to Step 6 on the course page.

Threads and Fibers

A thread is like a little self-contained program that runs inside a bigger program. Or to be bit more technical, your main program runs in a process. The process may contain smaller units of execution called threads. Each program runs at least one thread – the main thread – and if you run more than one thread your program is said to be multi-threaded.

Your operating system can run processes and threads and some computer languages let you access the operating system threads – called 'native threads' – in order to control the way in which multiple threads cooperate with one another. Some languages give you direct access to native threads. Ruby does not.

This means that Ruby threads are not as efficient as threads in some other languages. But they are portable. Ruby threads are run by the Ruby system itself and so your multi-threaded programs shouldn't rely on the vagaries of a specific operating system.

You can display the identifier and status of the *main* thread like this:

```
p( Thread.main )
```

Thread Status

Threads can have the status *sleep* – when a thread is waiting for something to happen or *aborting* when it's about to end and false when it terminates. These are the possible status values of a thread:

`run`	When the thread is executing
`sleep`	When the thread is sleeping or waiting on I/O
`aborting`	When the thread is aborting
`false`	When the thread terminated normally
`nil`	When the thread terminated with an exception

Mutexes

Sometimes threads may try to access some sort of shared resource. That can cause problems. Because one thread might modify a value in a way that is unexpected to - and can cause problems for – the other thread. Here I create a Mutex object called `semaphore`. Now inside my Thread objet I call `semaphore.synchronize`.

mutex.rb

```
require 'thread'

$i = 0
semaphore = Mutex.new

def addNum(aNum)
      aNum + 1
end

somethreads = (1..3).collect {
      Thread.new {
            semaphore.synchronize{
                  1000000.times{ $i = addNum($i)  }
            }
      }
}

somethreads.each{|t| t.join }
puts( $i )
```

Fibers

Ruby 1.9 and 2.0 also have another sort of unit of execution called a Fiber. This is a bit like a thread and a bit like a block. Unlike Threads, the execution of Fibers is not controlled –not scheduled – by Ruby. You have to do that yourself. There are several short programs to illustrate Fibers in the code archive for Chapter 17.

Notice the block syntax – `do` and `end`. To start a fiber running you call resume. To yield control to code running outside the fiber you call `yield`. When the fiber terminates, I can't resume it and a `FiberError` occurs. To avoid this problem you can test if a Fiber is alive using the `alive?` method.

fiber_alive.rb

```
require 'fiber'

f = Fiber.new do
      puts( "In fiber" )
    Fiber.yield( "yielding" )
      puts( "Still in fiber" )
      Fiber.yield( "yielding again" )
      puts( "But still in fiber" )
end

puts( "a" )
puts( f.resume )
puts( "b" )
puts ( f.resume )
puts( "c" )
puts( f.resume )
puts( "d" )

if (f.alive?) then
      puts( f.resume )
else
      puts("Error: Call to dead fiber" )
end
```

STEP 7

See Chapter 12 of The Book Of Ruby

MODULES AND MIXINS

Modules act as code repositories that can be included into classes when you need some common, shared features. But you cannot create a descendent from a module. Nor can a module have ancestors. And nor can you create objects directly from modules. Only classes can have instances. A module is a simple chunk of code that can be 'slipped into' other classes that need it.

Defining Modules

A module definition is similar to a class definition. It may define module methods with the syntax: *<module name>.<method name>* and it may define instance methods. Here module `M` has the module method `M.greet` and the instance method `greet`:

```
class M

      def greet
            puts "greet instance method"
      end

      def M.greet
            puts "M.greet"
      end
end

ob = M.new
M.greet
ob.greet
```

Mix in Instance Methods

Instance methods can be mixed into classes by specifying the module name after `include`, like this:

modules4.rb

```
module MagicThing
      attr_accessor :power
end

module Treasure
      attr_accessor :value
      attr_accessor :owner
end

class Weapon
      attr_accessor :deadliness
end

class Sword < Weapon
  include Treasure
  include MagicThing

  attr_accessor :name
end

class Jewel
      include Treasure
      attr_accessor :stone
end

s = Sword.new
s.name = "Excalibur"
s.deadliness = "fatal"
s.value = 1000
s.owner = "Gribbit The Dragon"
s.power = "Glows when Orcs appear"
```

```
s.value = 1000
s.owner = "Gribbit The Dragon"

j = Jewel.new
j.stone = "Sapphire"
j.value = 500
j.owner = "The Treasure Master"

puts(s.name)
puts(s.deadliness)
puts(s.value)
puts(s.owner)
puts(s.power)
puts
puts(j.stone)
puts(j.value)
puts(j.owner)
```

STEP 8

See Chapters 13, 14 and 15 of The Book Of Ruby

FILES, YAML AND MARSHAL

This step explains some features of File handling with Ruby as well as serializing data in human readable format with YAML or as byte-streams with Marshal.

File Handling

The File class descends from The `IO` class and it also provides access to methods from the `FileTest` module. In some cases, you can use either the `IO` or the `File` class to manipulate files. For example, I can use the `foreach()` class method of `IO` or of `File` to open a named text file and pass into a block each of the lines in that file.

io_test.rb

```
puts('IO.foreach...')
IO.foreach("testfile.txt") {|line| print( line ) }

puts("\n\nIO.readlines...")
lines = IO.readlines("testfile.txt")
lines.each{|line| print( line )}
```

File modes

You can create a `File` object with `File.new` and, optionally, a second argument to indicate the file mode. These are the file modes:

`"r"`	Read-only, starts at beginning of file (default mode)
`"r+"`	Read-write, starts at beginning of file
`"w"`	Write-only, truncates existing file to zero length or creates a new file for writing
`"w+"`	Read-write, truncates existing file to zero length or creates new file for reading and writing
`"a"`	Write-only, starts at end of file if file exists; otherwise, creates a new file for writing
`"a+"`	Read-write, starts at end of file if file exists; otherwise, creates a new file for reading and writing
`"b"`	(DOS/Windows only) Binary file mode (may appear with any of the key letters listed earlier)

Example:

file_ops.rb

```
# File.exist? will return true if a specific file or
# directory exists (note, you must use two slashes \\
# in directory paths inside a string

puts( "Testing File.exist? ..." )
if File.exist?( "C:\\" ) then
      puts( "Yup, you have a C:\\ directory" )
else
      puts( "Eeek! Can't find the C:\\ drive!" )
end

if File.exist?( "Z:\\" ) then
      puts( "Yup, you have a Z:\\ directory" )
else
      puts( "Eeek! Can't find the Z:\\ drive!" )
end

# to check if a specific name is a directory rather
# than a data file, use File.directory?
def dirOrFile( aName )
      if File.directory?( aName ) then
            puts( "#{aName} is a directory" )
      else
            puts( "#{aName} is a file" )
      end
end

puts( "\ndirOrFile..." )
dirOrFile("file_ops.rb")
dirOrFile("C:\\")

# An example of using File.exist? and
# File.directory? to take differing actions
# if a file is a data file or a directory
def dirOrFile2( aName )
if File.exists?( aName ) then
```

```
      puts( "#{aName} found" )
      if File.directory?( aName ) then
            puts( "#{aName} is a directory" )
      else
            puts( "#{aName} is a file" )
      end
else
      puts( "#{aName} can't be found!" )
end
end

puts( "\ndirOrFile2..." )
dirOrFile2("file_ops.rb")
dirOrFile2("C:\\")
dirOrFile2( "gribbitsnorp.xyz" )
```

YAML

In Ruby, the most common human-readable format for saving structured data to file is YAML. You need to 'require' YAML in order to use it. You can use the `y()` method to inspect and display YAML-format objects:
Example:

yaml_test1.rb

```
require 'yaml'

class Treasure
      def initialize( aName, aValue )
            @name = aName
            @value = aValue
      end
end


# y is a shortcut to print out object in yaml format
y( ['Bert', 'Fred', 'Mary'] )
y( { 'fruit' => 'banana', :vegetable => 'cabbage', 'number' => 3 } )
t = Treasure.new( 'magic lamp', 500 )
y( t )
```

When you want to save data you can use the `YAML.dump` method to create a YAML-format string of data. This method can take as a second argument, an IO object such as a disk file. So to save to disk, you open a file object for writing, using either `File.new` or `File.open` as here – then dump your YAML data and close the file.

yaml_dump2.rb

```
require 'yaml'

f = File.open( 'friends.yml', 'w' )
YAML.dump( ["fred", "bert", "mary"], f )
f.close

File.open( 'morefriends.yml', 'w' ){ |friendsfile|
    YAML.dump( ["sally", "agnes", "john" ], friendsfile )
}

File.open( 'morefriends.yml' ){ |f|
    $arr = YAML.load(f)
}

myfriends = YAML.load(File.open( 'friends.yml' ))
morefriends = YAML.load(File.open( 'morefriends.yml' ))
puts( myfriends )
puts
puts( morefriends )
puts
p( $arr )
```

There may be times when you want to save only a subset of data. You can do this by specifically defining the variables to be saved by YAML in an optional method named `to_yaml_properties` which you can place inside the class definition of some objects you plan to save. See the *limit_y.rb* sample program.

YAML format divides data into 'documents' starting with three dashes and individual objects starting with a single dash like this:

```
---
- !ruby/object:CD
  artist: The Groovesters
  name: Groovy Tunes
  numtracks: 12
- !ruby/object:PopCD
  artist: Dolly Parton
  genre: Country
  name: Greatest Hits
  numtracks: 38
```

For more information on YAML format, refer to http://www.yaml.org

Marshal

An alternative way of saving and loading data is provided by Ruby's Marshal library.

This has a similar set of methods to YAML to enable you to save and load data to and from disk but it saves byte-streams so the data files are not human readable.

Example:

marshal1.rb

```
f = File.open( 'friends.sav', 'w' )
Marshal.dump( ["fred", "bert", "mary"], f )
f.close

File.open( 'morefriends.sav', 'w' ){ |friendsfile|
    Marshal.dump( ["sally", "agnes", "john" ], friendsfile )
}

File.open( 'morefriends.sav' ){ |f|
    $arr = Marshal.load(f)
}

myfriends = Marshal.load(File.open( 'friends.sav' ))
morefriends = Marshal.load(File.open( 'morefriends.sav' ))

p( myfriends )
p( morefriends )
p( $arr )
```

To omit variables on saving, write a method called `marshal_dump` specifying the variables to be saved in an array, like this:

```
def marshal_dump
    [@variable_a, @variable_b]
end
```

Version numbers

You should be sure that any Marshal data read into your program has a compatible version number with the version of Marshal currently in use. See the program *version_m2.rb* for an example.

STEP 9

See Chapter 16 of The Book Of Ruby

REGULAR EXPRESSIONS

Regular Expressions provide ways for you to find and, if necessary, change text by matching patterns – for example, you could find all words that start with "R" and end with "y" to match words such as Ray, Rotary, Romany and, of course, Ruby.

Example:

```
str ="A ray of light shone on the rotary blades of the helicopter.
The old Romany woman looked up, and her ruby ring glittered in the
sunlight."

x = str.scan(/(\b[rR].*?y) / )
p x
```

Produces this output:

```
[["ray"], ["rotary"], ["Romany"], ["ruby"]]
```

See the Ruby API documentation for the Regexp class:
http://www.ruby-doc.org/core-1.9.3/Regexp.html

Groups and Captures

You can use a regular expression to match one or more substrings. To do this, you should put part of the regular expression between parentheses. Here I have two groups (sometimes called *captures*): The first tries to match the string "hi", and the second tries to match a string starting with "h" followed by any three characters (a dot means "match any single character," so the three dots here will match any three consecutive characters) and ending with "o":

groups.rb

```
/(hi).*(h...o)/ =~ "The word 'hi' is short for 'hello'."
```

After evaluating groups in a regular expression, a number of variables, equal to the number of groups, will be assigned the matched value of those groups. These variables take the form of a `$` followed by a number: `$1`, `$2`, `$3`, and so on. After executing the previous code, I can access the variables `$1` and `$2` like this:

```
print( $1, " ", $2, "\n" ) #=> hi hello
```

Processing Files

You may use Regular Expressions to process the contents of text and code files. For example, this program creates two files based on an input Ruby file. *comments.txt* contains all the comments but no code. *nocomments.txt* contains all the code but no comments:

regexp_2.rb

```
file_out1 = File.open( 'comments.txt', 'w' )
file_out2 = File.open( 'nocomments.txt', 'w' )

File.foreach( 'regex1.rb' ){ |line|
      if line =~ /^\s*#/ then
            file_out1.puts( line )
      else
            file_out2.puts( line )
      end
}

file_out1.close
file_out2.close

puts( "Done" )
```

STEP 10

See Chapter 20 of The Book Of Ruby

DYNAMIC RUBY AND METAPROGRAMMING

In most compiled languages and many interpreted languages, writing programs and running programs are two completely distinct operations: The code you write is fixed, and it is beyond any possibility of further alteration by the time the program is run. That is not the case with Ruby. A program—by which I mean *the Ruby code itself*—can be modified while the program is running. It is even possible to enter new Ruby code at runtime and execute the new code without restarting the program. The ability to treat data as executable code is called *metaprogramming*.

eval

The `eval` method provides a simple way of evaluating a Ruby expression in a string. For example, at a prompt the user could enter a method name such as `upcase` and ask a running Ruby program to execute that method:

eval2.rb

```
print("Enter a string method name (e.g. reverse or upcase):")
# user enters: upcase
methodname = gets().chomp()
exp2 = "'Hello world'."<< methodname
puts( eval( exp2 ) ) #=> HELLO WORLD
```

Special types of eval

There are some variations on the `eval` theme in the form of the methods named `instance_eval`, `module_eval`, and `class_eval`. The `instance_eval` method can be called from a specific object, and it provides access to the instance variables of that object. It can be called either with a block or with a string. See *instance_eval.rb*. The `module_eval` and `class_eval` methods to retrieve the values of class variables (but bear in mind that the more you do this, the more your code becomes dependent on the implementation details of a class, thereby compromising encapsulation). See *classvar_getset.rb*.

Adding and Removing Methods

You can use the `send` method to invoke the method named as the first argument (a symbol), passing to it any other arguments needed. here I call the `define_method` method to create a new method, `m`, containing the code of a block. Now when I call `addMethod` here, I create a method called `xyz` containing this code. Now I can call the method I've just created.

dynamic.rb

```
class X
      def a
            puts("method a")
      end

      def addMethod( m, &block )
            self.class.send( :define_method, m , &block )
      end
end

ob = X.new
ob.instance_variable_set("@aname", "Bert")
ob.addMethod( :xyz ) { puts("My name is #{@aname}") }
ob.xyz

ob2 = X.new
ob2.instance_variable_set("@aname", "Mary")
ob2.xyz
puts( ob2.instance_variable_get( :@aname ) )


X::const_set( :NUM, 500 )
puts( X::const_get( :NUM ) )
```

And this is how to remove a method:

rem_methods1.rb

```
puts( "hello".reverse )

class String
      remove_method( :reverse )
end

puts( "hello".reverse ) # reverse has been removed
```

Testing if a method exists

If you are adding and removing methods, there's one other useful trick you may need to know about. How to deal with an attempt to call a method that doesn't exist. You write a method called `method_missing` and when the user tries to call a non-existing method the call will be redirected to the `method_missing` method. It receives the method name as a symbol and so can display a descriptive error message as here.

```
def method_missing( methodname )
      puts( "Sorry, #{methodname} does not exist" )
end

xxx
```

Writing Programs at Runtime

It is possible substantially to modify the running program at runtime. This program lets the user enter Ruby code and evaluate it whenever a blank line is entered:

writeprog.rb

```
program = ""
input = ""
line = ""

until line.strip() == "q"
      print( "?- " )
      line = gets()
      case( line.strip() )
      when ''
            puts( "Evaluating..." )
            eval( input )
            program += input
            input = ""
      when 'l'
            puts( "Program Listing..." )
            puts( program )
      else
            input += line
      end
end

puts( "Goodbye" )
```

For more in-depth information on Metaprogramming refer to the book '*Metaprogramming Ruby*' by Paolo Perrotta.

FAQ

CAN SPECIFIC VALUES BE RETURNED BY A CONSTRUCTOR?

This subject was prompted by an eagle-eyed student who noticed a piece of sample code in which a value appears to be returned from the initialize *method.*

Question: I think you make a fundamental error (in the lecture on *'Encapsulation and information hiding'*). A constructor, i.e. (`initialize()`) should never do "a return of anything" explicitly! `X.new` should return an instance/object of the class `X`. In this case `ob` and that should be that instance/object, nothing else.

Answer: I agree. You should not return user-defined values from `initialize`. In my lecture I try to show how some data is 'naturally' hidden inside objects while other data can be accessed in various tricky and error-prone ways, thereby bypassing data-hiding to create imperfect encapsulation. In one simple example, I show this code to demonstrate that an instance variable such as `@xyz` is 'naturally' hidden and cannot be accessed by external code:

```
class X

 def initialize
   return @xyz = 500
 end

end
```

So, if I create an `X` object called `ob`, that object will be an instance of the `X` class rather than an integer, even though `initialize` specifically returns an integer:

```
ob = X.new
p ob
```

When I print `ob`, this is what is shown (the object including its object ID and its `@xyz` variable):

```
#<X:0x2a969c0 @xyz=500>
```

But note that it *is legal* (correct syntax) in Ruby for `initialize` to return a value. That's because, in spite of the fact that `initialize` is called immediately after object construction, it is just a normal method and syntactically it permits return

values to be declared. However, Ruby does not make use of the return value in this special case. That is because the return value of the constructor `new` is used (so the new object is returned) and any value from the `initialize` method which is called after construction is discarded.

But let's now suppose that, for some special reason, the programmer wants something other than a new object returned from the constructor. As I've said, this is not generally a good idea but even so there may be very special occasions when it might be useful. You can do this by overriding the constructor itselk – that is, the class's `new` method. Here's a simple example:

```
class X
 def initialize
 end

 def X.new
   return "Hello world"
 end
end

ob = X.new
p ob
```

This time `ob` is not an instance of `X` at all. It is the string "Hello world". Obviously, in most cases it would not be at all desirable for a constructor to return anything other than an instance of the class itself. Nevertheless, the code shown above is entirely legal in Ruby. Here is an example of an overridden constructor that is a bit more useful:

```
class Purchase
 def initialize(amount, item)
   @amount = amount
   @item = item
 end

 def Purchase.new(amount, item)
   if amount < 1 then
     return nil
   else
     super
   end
 end
end
```

Let's assume this code handles the inventory database of a store. Every time a purchase is made a new `Purchase` object is created storing the amount paid and the name of the item. But I don't want to allow a new object (and entry in the database)

to be created when no payment has been received. So I've overridden the `new` constructor to return `nil` in that case. Now I can run this code:

```
ob1 = Purchase.new(5000, "mink stole")
ob2 = Purchase.new(0, "Rolls Royce Silver Shadow")

p ob1
p ob2
```

And this is what I see:

```
#<Purchase:0x2b05540 @amount=5000, @item="mink stole">
nil
```

So while it is very uncommon – and usually ***very bad practice*** – to specify values returned by a constructor, this example shows that it *can* be done. But if you do so, proceed with extreme caution!

Useful Resources

Ruby Interpreter Downloads

Simplified (or all-in-one) Ruby Installers

Windows
http://rubyinstaller.org

Windows/Mac/Linux
http://bitnami.org/stack/rubystack (this includes extra tools for Rails and database development)

All operating systems

http://www.ruby-lang.org

EDITORS AND IDES

You can edit Ruby source code files in any plain text editor. Just be sure to save your files with the '.rb' extension – e.g. **rubytest.rb**

There are various source code editors and integrated environments available which offer dedicated support for Ruby programming. At the simplest level there are editors such as SciTE, which does 'code colouring' to highlight Ruby keywords and variables in different colours. At the other extreme is an IDE such as 'Sapphire (Ruby In Steel)' from my company, SapphireSteel Software, which includes IntelliSense (code completion), a visual debugger and a variety of other integrated tools. To get started with Ruby, a plain text editor is sufficient. If you plan to write complex applications in Ruby, however, a more powerful editor or IDE will be beneficial.

Here are a few options:

CROSS PLATFORM

SciTE http://www.scintilla.org/SciTE.html
Free programmer's editor with support for syntax colouring and code formatting for a number of languages including Ruby. Runs on Windows and Linux.

Komodo Edit: http://komodoide.com/komodo-edit/
A free editor for multiple languages including Ruby. A more powerful commercial edition, Komodo IDE, is also available. Runs on Windows, Linux, Mac OS X.

RadRails http://www.aptana.com/products/radrails.html
Free cross-platform IDE that is tailored for Ruby On Rails users. Runs on Windows, Linux, Mac OS X.

RubyMine http://www.jetbrains.com/ruby/
Commercial cross-platform IDE that is tailored for Ruby On Rails users. Runs on Windows, Linux, Mac OS X.

WINDOWS

Sapphire http://www.sapphiresteel.com
Commercial IDE for Micrsoft Visual Studio.

MAC

TextMate http://macromates.com/
Commercial programmer's editor for the Mac. Favoured by many Mac-based Ruby programmers.

MORE

There are several more code editors that support the Ruby programming language. The fact of the matter is that new ones come and old ones disappear fairly frequently. To find up-to-date links, just use Google to search for "Ruby code editor".

RUBY DOCUMENTATION

The Ruby-Doc site provides links to a variety of useful documentation:
http://www.ruby-doc.org

Documentation of Ruby's core classes can be found here:
http://www.ruby-doc.org/core/

THE BOOK OF RUBY

A revised edition of The Book Of Ruby (the book on which this course is based) is available as a paperback from No Starch Press: http://nostarch.com/boruby.htm

THE AUTHOR

Huw Collingbourne the founder of Bitwise Courses – an eLearning company based in the UK. In addition, he is Director of Technology with SapphireSteel Software (http://www.sapphiresteel.com), makers of the *Sapphire* (and *Ruby In Steel*) IDE for programming Ruby in Visual Studio and the *Amethyst* IDE for programming the Adobe Flash Platform. He has been programing since the early 1980s and had written programming columns for numerous magazines. He is author of The Book Of Ruby.