

Introduction to the Ruby Language

ENDAX MANUALS

Content is freely available for non commercial, non Profit, educational purposes provided the content is not modified and the author / publisher are given full credit for their work. Permission to reproduce for any other use in any format is strictly forbidden without the copyright holders permission.

©2008 ENDAX

Ruby

Everything in Ruby is an object, so called *primitives*, everything.

Ruby is dynamically typed which, of course, means that specific data types aren't determined until run time. This of course has its advantages and disadvantages.

Advantages are such obvious reasons such as code reduction as well some more subtle yet powerful reasons such as Mixins (explained shortly) that allow functionality to be shared by many classes painlessly at runtime. In fact, Ruby supports code closure which allows you literally pass code around and avoid having to write truckloads of Classes and methods to do the same things in Java, etc.

The main disadvantage is that you may or may not catch type errors until run time. Of course bear in mind that you're developing web applications and a simple type mismatch will likely result in a BUG report in QA in the worst case rather than sending your Mars Lander spiraling off toward the Great Andromeda Galaxy, i.e. the advantages far outweigh the disadvantages.

The following is a very brief introduction to Ruby that covers, first the basics and then what are some smart, and fairly unique (compared to current, popular languages), smart features of Ruby. You can follow along by starting up the interactive ruby shell with the following:

```
irb
```

This provides you with an interactive shell where you can create variables, classes, etc. All of the following examples were tested with `irb`.

Variables and Classes

Variables come in a few varieties based on their scope and are explained below:

Locals are simply the variable name:

```
title
```

Instance variables for a specific object are prefixed with an `@` sign:

```
@title
```

Class variables are a kind of global variable across all instances (Objects) of a class and unlike instance variables they have to be initialized before they are used; they are prefixed with two @@ signs:

```
@@title
```

And finally, Globals are prefixed with a \$ sign:

```
$title
```

Ruby classes have some unique features that separate them from other languages. Some basics are as follows:

All Ruby classes have to start with a capital letter:

```
class Something
end
```

Creating a new class is a matter of using the new operator:

```
s = Something.new
```

If you intend to initialize a class with some variables you must define a initialize method for the class:

```
class Something
  def initialize(stuff, more_stuff)
    @stuff = stuff
    @more_stuff = more_stuff
  end
end
```

Then you can initialize with the new method:

```
s = Something.new 'real stuff', 12
```

Ruby classes also have their own version of accessors, for example read and write accessors can be defined as follows:

```
class Address
  attr_accessor :number, :street, :zip, :country
end
```

Reader and Writer accessors can be defined as well accordingly:

```
class Note
  attr_reader :text
  attr_writer :edited_by
  def initialize(text)
    @text = text
  end
end
```

Inheritance is supported as expected and as follows:

```
class YellowNote < Note
  def initialize(text)
    super(text)
    @color = 'yellow'
  end
end
```

Ruby is technically a single inheritance language, but there is a very powerful concept called a Mixin that allows for multiple inheritance like behavior but without the dangers sometimes associated with multiple inheritance in languages such as c++.

Encapsulation is an important concept in OOP and is supported in Ruby with some fairly standard qualifiers:

```
class Tree
  attr_reader classification
  private
  def grow
    ...
  end
end
```

If there is no qualifier then it defaults to public. Also any method defined after a private (or protected, public) declaration will remain private unless otherwise specified. There is an alternative way to specify as well:

```
class Tree
  def grow
    ...
  end
  private :grow, :age
  public :trim
end
```

One final note about Class Methods; they are static methods that available as part of the class and don't necessarily need to be instantiated into Objects. They are generally used as constructors or utility methods, for

example:

```
class TimeMachine
  def TimeMachine.transcript(to)
    ...
  end
end
```

Take note of the `TimeMachine.travel` definition (you can use `self.travel` too), this is what makes the method static; additionally there is an alternative way to accomplish this, though rarely documented:

```
class TimeMachine
  class << self
    def transcript(to)
      ...
    end
  end
end
```

And to use:

```
# no TimeMachine.new needed here
TimeMachine.transcript 'Battle of Hastings'
```

Finally, Ruby supports the ability to define operators similar to c++, for example:

```
class StuffWrapper
  def initialize()
    @stuff = Array.new
  end
  def <<(append_value)
    @stuff << append_value
  end
  def +(append_list)
    @stuff += append_list
  end
end
```

This allows me to use basic operators on my class once I have defined them.

Modules

Modules, in one way, share a similar purpose to Namespaces or Packages in other languages and become a natural way to group functionality, but that is where the similarities end. It's important not to think of Modules as

Classes because they differ greatly (though you will soon see that a Class can *mix in* a module). Modules are **not** Classes and if you're coming from a Java / C# background where everything is an object (whether or not it should really be an object), this can be a little confusing.

Modules are a good way to group functionality that wouldn't otherwise naturally fit into a class. For example, I need to add some basic logging to my Classes:

```
module Logger
  def log_message(message)
    puts message
  end
end
```

I can then include the module in any subsequent Class:

```
class Person
  include Logger
end

class Place
  include Logger
end
```

This will allow me to use the module inside of the class, for example:

```
joe = Person.new
joe.log_message 'This is a log message..'
```

This functionality is known as a *mix in* but we will go into further detail regarding mixins shortly.

Strings

The String class in Ruby is one of the largest classes which contains just about every possible method ever developed in any language, and a full review of the methods is beyond the scope of this manual. First, there are some basic differences from what you may be used to. For example, single or double quotes mean different things. Single quotes mean a basic string:

```
puts 'A string with escaped single \'quote\''
```

Using double quotes support the expression escape for embedding the output of a variable in a string:

```
require 'date'
puts "Today's date is: #{DateTime.now}"
```

Additionally, it is possible to specify a *here document* which will print a number of lines:

```
puts <<-END_STRING
A long and multi-lined passage
that demonstrates how to create
a here document which will print
everything after the '<<-' until the
constant that is declared first..
END_STRING
```

You can of course, set to a variable, or use in any way you wish.

As mentioned before the String class has numerous methods, more than we can cover here but here are a few examples of the powerful methods for use with a String.

Ability to split strings based on regular expressions:

```
'a4string3with9some8numbers'.split(/[0-9]/)
=> ["a", "string", "with", "some", "numbers"]
```

Ability to extract values based on a regular expressions:

```
'a4string3with9some8numbers'.scan(/[0-9]/)
["4", "3", "9", "8"]
```

Ability to substitute based on regular expressions:

```
'a4string3with9some8numbers'.gsub(/[0-9]/, ' ')
=> "a string with some numbers"
```

Any many more options for string manipulation. See your favorite reference for more.

Ranges

Ranges are a data type of their own. It is best to think of them in terms of a sequence:

```
(0..9).to_a
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can set a range to a variable and operate on it accordingly as well:

```
zero_to_nine = 0..9
=> 0..9
zero_to_nine.max
=> 9
zero_to_nine.reject {|x| x < 7}
=> [7, 8, 9]
```

Finally Ranges come with their operator with three equal signs called a *case equality operator* that returns true or false if a give number falls into a range, for example:

```
(0..9) === 4
=> true
```

or

```
('a'..'l') === 'g'
=> true
```

and

```
(0..9) === 72
=> false
```

This is, of course, a very significant time saver when searching a range for the existence of a particular value. Ranges can be used in a variety of ways from representing a range of values as well as condition expressions, etc.

Symbols

There is nothing remotely complicated about Symbols; they are simply representations for a name. Again, just something you probably have never come across in other languages but an important part of using Ruby.

Symbols are created by simply stating them and can be used to a variety of purposes:

```
# hash, key as symbols
s = { :one => 1, :two => 2, :three => 3 }

# using
s[:two]
```

Collection Types

Ruby supports a variety of standard collections. For example to create an

Array

```
a = Array.new
```

Or simply define one with values:

```
b = ['a', 'b', 'c', 'd']
```

Arrays can be made into Sets with:

```
# need to require the set library for this
require 'set'

# add a redundant variable to the array
b << 'd'

# and create a Set from it
s = Set.new b
```

which will create a Set, a unique of variables in the set; or to just trim out redundant values from an array, do it all at once:

```
b = Set.new(b).to_a
# or use the uniq method
b.uniq!
```

Briefly, the ! operator means that the method changes the original value as opposed to simply returning it, for example:

```
b = b.uniq
b.uniq!
```

are the same thing.

Hashes are just as easily created with either:

```
h = Hash.new
```

Or by specifying some default values:

```
h = {'one' => 1, 'two' => 2, 'three' => 3}
```

And are accessed as you would expect

```
h['one']
```

Hashes come with an assortment of useful methods, too many to mention

in this introduction but for example:

```
# invert a hash
h.invert

# delete a member based on a condition
h.delete_if {|key, value| value >= 2 }
```

Exceptions

Ruby handles exceptions like most OOP languages with a few additional features. First the basics:

```
begin
  raise 'Exception!'
rescue
  # some code to handle the exception
end
```

Of course it's possible to specify which exception to catch:

```
begin
  # some code
rescue CustomError1, CustomError2 => custom
  puts "CustomError has occurred: #{boom}"
rescue StandardError => custom
  puts "CustomError has occurred: #{boom}"
end
```

Ruby also make the global variable `$!` available which contains the raised exception. Additional features include `ensure` and `else` blocks, for example:

```
some_file = File.open('README')
begin
  # some code
rescue
  # some code to handle the exception
else
  puts 'Process Successful'
ensure
  some_file.close unless some_file.nil?
end
```

Additionally, Ruby provides a `retry` statement that will repeat what is in the entire `begin` to `end` block but be sure to use with caution since it's very easy to get yourself in an infinite loop:

```
@login_keys = ['one', 'two', 'three']
@current = 0
```

```
begin
  remote_server.connect(login_keys[current])
rescue RemoteLoginError
  if @current < @login_keys.size
    @current += 1
    retry
  else
    raise
  end
end
```

Again, use caution when using `retry`; it can easily get you into a bad situation when not used properly.

One final thought on Exceptions, it's important not to think of Exceptions as *errors*. They are *exceptions* to the normal processing on an application and should be raised to a point that they are useful to someone, likely a developer. There is a *very bad* tendency to suppress them for the benefit of the end user. Exceptions are good. They let you know when and where a problem has occurred.

Requires

We've seen a number of different references to `require` so far and should take a moment to explain how these PATHs are found, created etc.

There are set of default paths that can be seen with the following:

```
ruby -e 'puts $:'
```

The `$:` variable holds all paths and it is possible to append to this if necessary:

```
$: << "/tmp"
```

while your programming is running or optionally using the `-I` command line switch to prepend directories to the `$:` load path.

Finally, basic paths work as well. Included in the load path is a reference to the current directory (a dot). This allows you to load modules, etc. that are also in the current directory or reference if the files are localized (for example in a `/lib` folder)

Smart Features: Statement Modifiers

Statement Modifiers are a smarter way to deal with simple conditional clauses. Instead of having to write 3 or 4 lines to create some desired outcome, it can be simplified into a single line where applicable.

Things like:

```
if x.valid?  
  puts 'Valid!'  
end
```

can be reduced to

```
puts 'Valid' if x.valid?
```

or

```
while x < 10  
  x+= 2  
end
```

can be reduced to

```
x+=2 while x < 10
```

Now it may seem backwards, but of course it's not. It's just different from what you're used to seeing. Like most of Ruby, once you start to use it like Ruby, it's becomes very intuitive.

Smart Features: block iterators

Block Iterators are one of the nicest features in Ruby and liberate us from the archaic `for i = 0` loop nonsense that has plagued us now for generations. Block iterators come in two forms, single and multiple line iterations.

Single line iterations are likely for shorter statements and can be performed on a single line:

```
a = ['a', 'b', 'c']  
a.each {|x| puts x.capitalize}
```

This outputs a capitalized version of each element in the array. Also for those times when you just really absolutely have to have an `i`:

```
a.each_with_index {|x, i| puts "#{i}: #{x}"}
```

outputs:

```
0: a
1: b
2: c
```

Multi-lined iterators are similar except for they are able to span multiple lines clearly:

```
a.each |x| do
  #we may have to write a number of lines
  puts x.capitalize
  #multi-lined iterators are good for this
  puts x.swapcase
end
```

Why are there two options for doing the same thing? Well you may to do a number of things inside your iteration block, and stuffing those all into a single line (which can be done with a semi-colon separating them) would lead to unreadable nonsense -- always keep in mind that programming languages are for *humans to communicate*; machines couldn't care less how you organize your sequence of instructions and creating cryptic blocks of unreadable code will undermine the purpose of a language in first place.

Smart Features: better control statements

You can now avoid the super long OR statements found in other languages e.g.,

```
if ((x == 'y') || (x == 'x') || ('x == 'z'))
  ...
```

Can be reduced to:

```
if ['y','x','z'].include? x
  ...
```

Again, the same result but a much cleaner, more concise and readable format for expressing it. Also, seemingly reversed in how it is expressed but that is only because your thinking in terms of other languages.

On a more general note, you should be leveraging ruby as a language to

avoid complicated, confusing constructions such as the long OR statements. In fact, you should never see very long repetitive statements in ruby. If you end up writing something like what is above then you are doing something wrong. Same thing goes for very large if / else if / else statements that span many lines. If you are writing these type of constructs found in C, etc. then you are not leveraging the polymorphic attributes of your OOP language or at least using statement modifiers to simplify the readability of the code.

Smart Features: Mixins

Introduced above, Mixins provide a very powerful way to reuse functionality and generally extend or plugin your classes with an array of preexisting code. What was described above is a fairly straightforward extension of a class with an existing set of functionality packaged as a Module. But what if you want to determine the functionality at run time? Of course, there is a way to do that as well.

Imagine a scenario where you want to output a specific format type based on the user's request. In other words, your application needs to be able to provide HTML, XML, or Yaml accordingly.

First provide the output details of each in 3 different modules:

```
module HtmlOutput
  def custom_output(a_class)
    #specifics for converting class to HTML
    ...
  end
end

module XmlOutput
  def custom_output(a_class)
    #specifics for converting class to XML
    ...
  end
end

module YamlOutput
  def custom_output(a_class)
    #specifics for converting class to Yaml
    ...
  end
end
```

And a mini-factory so we can use a symbol to specify the type of output that we want:

```
type_factory = { :html => HtmlOuput,  
                :xml => XmlOuput,  
                :yaml => YamlOuput }
```

Now to use, simply extend your class with the applicable module (XML in the following example):

```
@my_model.extend type_factory[:xml]
```

and output:

```
puts @my_model.custom_output
```

Of course since this is a Module, you can extend any class with the functionality and transparently reuse the different Modules accordingly.

This pattern is known traditionally as the GOF (Gang of Four) *Strategy* Pattern and arguably more intuitive now with the use of Mixins. Ruby provides a number of these Object Oriented patterns in the form of libraries, notably Vistor, Delegate, Observer, and Singleton. Further review is beyond the scope of the manual but it is important to know that this functionality is available.

Smart Features: Code Closures

In Ruby it is possible to pass actual code around and to execute it via the yield method, for example, define a method:

```
def execute  
  yield  
end
```

and then pass a chunk of code to it:

```
execute {puts 'Yield this'}
```

Of course those examples aren't entirely useful so let's implement another example that uses a & operator to specify to a method that what is being passed in a block of code.

```
class Engine  
  def start  
    #specifics for starting the engine  
    ...  
  end  
  def stop
```

```

        #specifics for stoping the engine
        ...
    end
end

class EngineSwitch
  def initialize(&perform_action)
    perform_action.call self
  end
end

```

The `perform_action` can also be stored as a variable, i.e. it doesn't have to be used right away.

Using:

```

my_engine = Engine.new
EngineSwitch.new { my_engine.start }
EngineSwitch.new { my_engine.stop }

```

This approach is clearly more appropriate because it allows the Engine to specify how to start and stop, and lets the EngineSwitch simply (as it should) trigger the start or stop accordingly.

Smart Features: A Functional Language?

Functional languages are not like anything you've probably seen up until this point as a developer (and likely never will) and are certainly not going to be covered here. That said though, they traditionally look at everything in terms of lists and use recursion to process them (as opposed to `for` / `while` / `each`) iterators. This may seem bizarre and I suppose it is at first, but it turns out this approach to list processing can be very helpful and, fortunately, can be found in small amounts in Ruby.

This first, and likely most useful, is the `map` (also known as `collect`) method. This allows to process each item in an enumeration, for example:

```
(1..5).map { |x| x*2 }
```

returns an array of:

```
=> [2, 4, 6, 8, 10]
```

Additionally you can modify the original array with the `!` version:

```

a = ['a', 'b', 'c']
a.map! { |x| x.capitalize }

```

This modifies the original variable to:

```
=> ["A", "B", "C"]
```

Another aspect of functional languages is the ability to shift, slice and generally cut some part of a list or add to it with little effort. The following are some example of how to do this in Ruby:

```
# remove the first element of the array and return it
a.shift

# put the A back in the front of the array
a.unshift 'A'

# grab the remaining array after the first element
a.slice(1..a.size)

# pop last element
a.pop

# push element to the end of array
a.push 'C'
```

Again, it is unlikely that you will find yourself using or needing a functional language, but the approach to list processing is helpful and it is very likely that you will be processing lists in general.

In summary, Ruby is a significant departure from what you're likely used to with it's dynamic typing, ability to extend and modify classes at runtime, and a host of other advanced feature that I'll reserve for a later date.

How to Avoid Writing Java/C# in Ruby

Certainly not a requirement of the language, but far too often you see Java classes with a getter and a setter for every single attribute. This, of course, completely defeats the original purpose of encapsulation in the first place and has become the defacto standard for Java programmers creating new classes. What's important here isn't so much the Java dogma as much as it liberating yourself of your old ways even though you've convinced yourself that they are right.

Don't make a `get_this` or `set_that` for every single attribute in your class. Instead think about the purpose of encapsulation, i.e. if I need to create a getter and a setter for this attribute that does nothing but get or set an attribute, maybe I should just make it public or use an `attr_reader`

or `attr_writer`? Maybe there's attributes in my class that I shouldn't be exposing via getters and setters? Also use the Ruby accessors for accessing these attributes as opposed to a slew of `get_this`, `set_that`, etc:

```
class EncapsulatedThing
  attr_reader :read_only_attribute
  attr_accessor :public_attribute
end
```

Don't dig into and start creating `for i = 0` loops everywhere, use the block iterators that are provided by the language. If you absolutely need a `i` value use the `each_with_index` method (and others) to accomplish the same result.

```
a.each_with_index {|x, i| puts "#{i}: #{x}"}
```

Use double quotes with embedded variables instead of +ing them all together in a massive mess, i.e don't do this:

```
puts 'This is a ' + weather + ' day: ' + date
```

Use:

```
puts "This is a #{weather} day: #{date}"
```

Don't write a truckload of print statements for a long block of strings:

```
# bad
puts '<html>'
puts ' <body>'
puts ' <h1>This is wrong</h1>'
puts ' </body>'
puts '</html>'
```

Use the *here document* operation:

```
# good
puts <<-END_HTML
  <html>
  <body>
  <h1>This is good</h1>
  </body>
</html>
END_HTML
```

Don't make a Class out of everything. One of the things that still makes me cringe when using Java is that *everything must be defined as a class*. This is OOP at it's political extreme. OOP was originally meant to allow us

to model objects based on reality; but if everything has to be a object, we are forced to create objects out of things that aren't really objects.

For example, I need a basic logging system. So do I model an object of a Logger? What is that exactly, a Guy object with a Notepad and Pen objects? A object that cuts down and mills trees? The point is, it's likely that you will come across some group of related functionality that can't be represented naturally with objects – use a Module when applicable.

Don't Repeat Yourself

Principles

DRY as it sometimes abbreviated, is essentially a philosophy to avoid excessive duplication in programming. It has many positive attributes such decreasing the need for later changes across a code base, increased clarity, and generally reduces inconsistency. It extends beyond simply code and can include practices for documentation, database schema, build scripts, and test plans as well. The core principle is that modifications of code do not inadvertently change areas that shouldn't have been affected by such change.

Practice

DRY implementations can be applicable in almost every application and must determined based on it's applicability and balanced against over obsessing the principle in the first place. A somewhat trivial example might be:

```
if something == true
  x = SomeClass.new
  x.title = 'This is my title'
  x.text = 'This is my text'
  x.something = true
else
  x = SomeClass.new
  x.title = 'This is my title'
  x.text = 'This is my text'
  x.otherthing = false
end
```

Can be DRY'd by removing the duplicate code and reducing the condition to the a single line:

```
x = SomeClass.new
```

```
x.title = 'This is my title'  
x.text = 'This is my text'  
x.otherthing = something ? true : false
```

Again, a nonsensical example, but a general rule of thumb is that if you have lots of redundant code, it is likely that you are not following the DRY principle.

Don't DRY-out

Keep this principle in mind but don't obsess over it. There are certainly valid occasions when implementing DRY will take absurdly longer than simply having duplicates. A generally good example of this is lightweight, inexpensive views such as RHTML pages. Spending hours and hours trying to squeeze every last drop of DRY out of your views will likely just cut into your production schedule and in the worst case, nullify any progress you may have made.

In my Java youth, I had a particular employee that only liked to work on things that were *exceptionally* interesting and when asked to add a few, redundant links to a small range of pages, spent an entire week developing a 'Link Factory' to manage the 3 or 4 set of links that was realistically about 10 minutes worth of work; and of course being Java, it required a complete recompile of the application for the smallest change.

You get the idea, don't obsess over principles or patterns. It's more critical that you understand when and where they are applicable than even how to use them in the first place. DRY is meant as a means to be more productive and clear, obsessing over it incurs the opposite.

Conventions

The following conventions are fairly universal in Ruby but of course, not enforced by the interpreter (except class names), i.e. these are recommended conventions.

Indentation should be two white spaces and *not a tab character, e.g.

```
def some_function  
  if true  
    puts 'It is true'  
  end  
end
```

A single white space should separate all statements

```
if x == y
  y += 2
  call_this x
end
```

The exception is when assigning a value in a method argument list:

```
def something(x, y=5, more_args={})
  ...
end
```

All classes are camel case, all methods and variables use underscores.

```
module SomeModule
  class SomeClass < AnotherBaseClass
    attr_accessor :some_variable
    def some_method(a_variable)
      @some_variable = a_variable
    end
  end
end
```

Iteration block format should be based on the size of the iteration block:

```
# short
(0..5).each {|x| x.strip!; puts x }

# long
some_array.each do |x|
  x.strip!
  some_call x
  puts x
  puts x += 'this'
end
```

Instantiations should use the new method:

```
some_array = Array.new
some_hash = Hash.new
```

Use boolean values and **not** 0 or 1 for true and false:

```
some_value = true unless another_value == false
```

All comments should have a space between the comment character and the text. Lines should not be longer than 80 characters long

```
# This is a comment that spans more than one line
# to show the multi-line comments
def some_method(x)
```

```
# this is a single line comment
w ||= x
end
```

Method calls should use parenthesis when the number of variables are small

```
some_method_call x, y
a_long_method_call(x, y, z, a, b, c)
a_short_but_complex_call(%w[1 2 3], x, {puts 'test'})
```

Again, these are recommended conventions and maybe used accordingly; that said, it may be possible that the shop you work has a coding standard and, of course, some convention may no longer be optional.