A Ruby Primer
By Scott Haines

“Ruby is designed to make programmers happy”
Yukihiro Matsumoto.

The Ruby programming language takes a little getting used to, but give it a little of your time and it will reward you greatly. Unlike many conventional languages that have hit the mass market, Ruby is a completely open language. This means that you can change its core behavior at runtime, allow it to fix itself overtime, etc. Many people got excited by Ruby for its *Reflective* capabilities, and use it for metaprogramming, while in the other camp, people are using Ruby to build fast, functional, and scalable websites, game portals and even iPhone game engines via macruby (http://www.macruby.org/).

Getting Started

All of the examples provided in this primer and the class to follow are located on github.com at https://github.com/newfront/hacker-dojo-rails3-code-base

If you are only interested in the Ruby Primer class files, they can be found at https://github.com/newfront/hacker-dojo-rails3-code-base/tree/master/rails_primer_code/

1.1 Using Interactive Ruby Environment (irb)

Check to make sure irb is installed on your computer. It should have been packaged along with Ruby, but hiccups occur, so it is best to make sure we are all on the same page.

Go to your console (terminal) and issue the following command to check for irb.

>> irb -v
=> irb 0.9.5(05/04/13)

1.2 Hello World

If you are familiar with any college course in programming, you have the "oh so expected" Hello World output as your first step into the new language you will be studying. Don't worry this will be as pain free as possible.

Enter the following into irb.

>> puts "hello world"
hello world
=> nil

(*note. What does the => nil mean? This takes us into the very Core of the Ruby language itself and is for another time and another place, it is a reference to the Object Type. Since everything in Ruby must be of some type, then a single one-time use output like puts is of the Object Class Nil. For more on the Ruby Language's idiosyncrasies please refer to the Master Himself **Yukihiro "Matz" Matsumoto** The Ruby Programming Language, O'Reilly Media **SBN-10:** 0596516177)

So what just happened? You issued a simple command telling the Ruby language to take the string "hello world" and print that to the terminal, alternatively you could have also used >> print "hello world", the only difference is **puts** will automatically append a newline character to your output, this is nice if you want to read a lot of output in an orderly fashion. Using **print** will allow you to output to the same line in the terminal.

(*note. Moving forward I will be describing the command-line prompt as the terminal.)

1.3 Ruby Variables

There are many types of variables in the Ruby language, all with their own pros and cons. Below are the basic types of variables.

Instance Variable

Instance variables begin with the @ followed by a word such as **@foobar**. (* instance variables have local **scope** and are available from within the class method that is calling them.)

Class Variables

Class variables begin with the @@ and are followed by a word such as **@@foobar**.

(* class variables are local to the entire class whose methods make up your Rails application or sub routines. Class variables have **global** scope within their object instantiation, wherein they are available from any method within the class.)

Global Variables

Global Variables are available to any class within your project. A global variable begins with *$* and is followed by a word such as ***$foobar***. These variables persist within the entire application but remember to use caution when setting these, as they will persist for the entire time that the server is running.

1.4 Ruby is an Object-Oriented Language

If you don't know what a **Class** is referring to above, this is referring to the Object Oriented notion of Objects. A Class is a blueprint of your Object, and when you create a new object you *instantiate* that object. Below we will work with a simple Ruby class called Greeter, this shows you the power of classes, and also the ease of working with classes in ruby.

Ex. (Greeter.rb, run_greeter.rb)

You can pull all of the examples out of the github directory for this course. Also, all code and examples have embedded rDoc (ruby documentation).

Run the sample in IRB.

>> irb
>> require "Greeter.rb" => true
>> greet = Greeter.new("Scott") => #<Greeter:0x10159aa68>
>> greet.say_hello
Hello, Scott => nil

>> greet.say_hello!
Hello, Sir Scott. I understand it is urgent. => nil

How does this all work?

In the code above, we actually got some interesting things accomplished. First, we required a code base outside of our main program (in this case irb), and we were able to *instantiate* a new Greeter object.

If you look through the Greeter.rb code, you will see at the top there is some comments to help understand the code on the page, although this code isn't terribly difficult to follow (in fact it is as basic as it really gets), this style of documentation can be real great for large projects. Ruby's rDoc will take the code and your comments and help create useful documentation for your code automatically.

Next interesting thing we see is the use of a class variable. @@name, holds reference to the name that the Object is *instantiated* with. In our example, this is my name "Scott".

Lastly, we have our two class methods. 1. say_hello, and 2. say_hello!. These are simple to follow class methods, class methods are also known as Object functions. However, people will typically try to use the same vocabulary, so **method** is better when talking to people about your code.

say_hello

Takes our Class variable, and assigns it to a preformatted string using the #{} pattern.

say_hello!
This is identical to the function above, however because Ruby best-practices state that you should use the **!** as a sign of warning or importance, I show a different tone in the response of the Greeter class.

How to find the class of a Ruby Object

In practice, sometimes you want to figure out the class of an object that shows up in your code, or the code base you are working with. A quick way to do that is to use the Ruby method **class**.

>> greet.class
=> Greeter

2 Strings

A great place to begin our journey into Ruby is with the String Class (http://ruby-doc.org/core/classes/String.html).

Strings make up the text we view when we render html into visual output on someone's computer screen, at their very nature, a String is no different than the text I am typing into this word processor. Strings can be words, references to numbers, serialized Objects (What does that mean? You will find out soon enough.), and much more.

(***note. We actually already created our first string in our Hello World example before. However, we had no way to reference the string itself, so it became of type nil.)

Example.

Let's use an **instance variable** and associate that to our new String.

Open up **irb**, and type the following.

>> @name = "John"
=> "John"

Well that was easy. Let's now reference this string and write **Hello, John** as output inside **irb**.

(*note. Remember we used the **puts** command to output "Hello World" before, lets use it again, but this time we will inject the variable **@name** into the output.)

>> puts "Hello, #{@name}"
=> "Hello, John" => nil

Here are some simple little tricks for testing for the presence of **String**.

>> @name.is_a? String
=> true

In Ruby, you can use the is_a? method to test the presence of a particular class object. In this case we wanted to test for **String**, and this was true.

2.1 Popular String Functions

2.1.1 Formatting Strings

capitalize
This will create an Uppercase first letter in your String. Returns a new string.

capitalize!
Same as above. However, this modifies the original string permanently.

Example
>> @str = "there she blows"
=> "there she blows"

>> @str.capitalize
=> "There she blows"

>> @str
=> "there she blows"

>> @str.capitalize!
=> "There she blows"
>> @str
=> "There she blows"

downcase
Creates a lowercase formatted copy of a string.

Example

>> @j = “HELLO”
>> @j.downcase
=> “hello”

upcase
Creates an uppercase formatted copy of a string.

Example

>> @str = "Hacker Dojo"
=> "Hacker Dojo"

>> @str.upcase
=> "HACKER DOJO"

2.1.2 Concatination

If your not familiar with this term it basically means to merge or add. Depending on the use case. As we will see later in the section on Arrays.

Example

>> @n = “John”
=> “John”

>> @n << “ Smith”
=> “John Smith"

Alternatively

You can also simply use String addition, the plus sign (+) will add the two Strings to each other. This creates a permanent change in that String.

>> @name = “Scott”
>> @name_last = “Haines”
>> @name = @name + @name_last

>> @name => “Scott Haines”

2.1.3 String Replacement

Can be as simple or extreme as you want. Basically, gsub can be used with or without regular expressions, search for a String or Pattern. (If you are not familiar with Regular Expressions don’t worry, they are not needed for most tasks, but they will definitely speed up any work you do where you are looking for patterns within large regions of text (blobs)).

gsub

Will return a copy of the string with the substitution applied.

Example

>> @str = “Hacker Dojo”
=> “Hacker Dojo”
>> @str.gsub(/\DOJO/i,"scott")
=> "Hacker scott"

>> @url = “http://ruby.org”
>> @url.gsub(“ruby”,”google”)
=> “http://google.org”

>> @url = http://www.google.com?www=google
=> http://www.google.com?www=google

>> @url.gsub(/([Gg]oogle(?=.))/,"scott")
=> http://www.scott.com?www=google

Regular Expressions were used in the last two examples. The first example uses a simple string pattern search that is case insensitive (denoted by the /i), while the second pattern is a bit more tricky. It uses a case insensitive search on the first letter of **G**oogle, and will only match if the word google is followed by a period. This way we can preserve our matches if we are using the same word elsewhere, like I do with www=google.

(* For a primer on Ruby Regular Expressions please check out (http://www.tutorialspoint.com/ruby/ruby_regular_expressions.htm), or take a peek at an intro to Perl book. (Learning Perl, Third Edition by Tom Phoenix and Randal L. Schwartz. Orielly)

2.1.4 String Searching

include?
This method will let you know if the word (String) you are looking for exists in a blob of text.

Example

>> @s = "There is a dog in the back yard"
=> "There is a dog in the back yard"
>> @s.include? "dog"
=> true

Substring
Strings can be broken down by the individual letter index. This is useful when you are looking for specific sections of a String.

>> @s = “Here is a String”

>> @s[0,10]
=> “Here is a”

Count
This function will let you know the length of a string. This method is useful in the case of input validation. If a password is say less than 9 characters in length, send back an error.

Example

>> @str = “Hacker Dojo”

=> “Hacker Dojo”

>> @str.count @str
=> 11

Above you will see that you can use the String Object to count the full string length of itself. Try just using **count** out of context, it will throw an exception since the method needs to be joined to an Object that is of type String.

empty?
The string is empty, let’s test to see if that is true.

>> @j = “”
>> @j.empty?
=> true

eql?
Is string A equal to string B. Again this method is real useful when dealing with validation on the server-side.

>> @j = “Word”
>> @m = “Word”
>> @j.eql?(@m)
=> true

2.2 Exercises

1. Create the following String. **Why I love Ruby**, and store it in memory as an instance variable.
2. Append the word **Programming** to the String from Exercise 1, but only append it within the actual output in **irb**.
3. Create a duplicate of the String from exercise 1, and see if the two variables are equal.
4. Create a paragraph of text in **irb**, and store it as an instance variable. Now create another instance variable to use as a search pattern. (hint: **gsub**). Now, create a third instance variable to use as your replacement text for your string replacement. Lastly, run your string replacement in **irb** and see if you are able to output the correct paragraph of text in the command line window.
5. Think up some good uses for string replacement and experiment with some of these, see what you can come up with.

3 Collections. Arrays, Hashes and a look at Mashie Gem

I hope you are still reading along at this point. If so, then Ruby is starting to excite you. I know at this point your probably wondering, “Why haven’t we written any tangible piece of code. Do I have to use **irb** to create an actual website or application with Ruby on Rails?”

The answer is no, you don’t have to use **irb** forever, but it is a great tool to keep in your arsenal.

Think about **irb** as your Ruby notepad, if you have an idea and want to quickly test it out, then this is the quickest way to do that. It also plays a fundamental role in your Ruby on Rails life cycle, fairly soon we will get introduced to the Rails Console, and all this time in **irb** will pay off.

Now, back to the topic of Ruby Collections, with a peek at the Array and Hash Classes.

3.1 What is an Array?

http://ruby-doc.org/core/classes/Array.html

An Array is a simple mechanism for storing collections of data. Indexes are used to point Ruby to a specific piece of data within the Array Object.

So what does this mean? Let's take a look at a real life example.

3.1.1 Use Case

You are building an application and have no real need for a database, since you want to hard-code everything. Say you have a list of places, countries, states – basically data that you don't see changing anytime soon. This is the perfect use for an Array.

3.1.2 Example

Here is my hypothetical list of places I want to tell people about.
Georgia, Florida, Massachusetts, England, and California.

```
>> @places = Array.new
=> []
>> @places.is_a? Array
=> true
>> @places << "Georgia"
=> ["Georgia"]
>> @places << "Florida"
=> ["Georgia", "Florida"]
>> @places << "Massachusetts"
=> ["Georgia", "Florida", "Massachusetts"]
>> @places << "England"
=> ["Georgia", "Florida", "Massachusetts", "England"]
>> @places << "California"
=> ["Georgia", "Florida", "Massachusetts", "England", "California"]
```

Now we have our list all stored in a convenient and easy to use instance variable. Now say I want to get a specific chunk of data out of this Array, well that is where our array index comes into play.

Arrays are odd at first. This is because their index begins with a 0. So typically what people expect to be the first result, is actually the second and so on. This of course would cause serious issues if a mistake like this somehow went into a production (live) environment.

>> @places[0]
=> "Georgia"

As you can see the first element in the Array is at index (or space) zero. If you want to see the full size of the Array (as in number or elements contained) then you can use the **size** method for the Array.

>> @places.size
=> 5

Arrays will play a key role in a lot of the work you do as a Ruby programmer and a Rails developer. You would be wise to get to know them very intimately, but to also know that the Array is not useful in all occations.

3.2 Array Nesting

You can store Arrays within Arrays, think of this as you would those little eggs with the nice painted outsides that get smaller and smaller. You have a single Array (egg) and within that main Array you can store more and more.

>> @places = Array.new
=> []
>> @places[0] = Array.new
=> []
>> @places[1] = Array.new
=> []
>> @places[0] << "Sunny"
=> ["Sunny"]

>> @places[1] << "Florida"
=> ["Florida"]
>> @places
=> [["Sunny"], ["Florida"]]
>> @places[0] << "Cold"
=> ["Sunny", "Cold"]
>> @places[1] << "Massachusetts"
=> ["Florida", "Massachusetts"]
>> @places
=> [["Sunny", "Cold"], ["Florida", "Massachusetts"]]

Now think of how we can use what we learned before to print this out into a sentence in **irb**. Think of a Sunny Place? I am thinking of Florida.

>> puts "The weather in #{@places[1][0]} is real #{@places[0][0]}"
The weather in Florida is real Sunny
=> nil

What about a Cold Place? Maybe Massachusetts.

>> puts "The weather in #{@places[1][1]} is real #{@places[0][1]}"
The weather in Massachusetts is real Cold
=> nil
You see now the power of the Array. Once you have a reason to use it, life becomes simple, but that was still a more complex and seemingly mismanaged way of storing sets of Associated Data, why would anyone want to remember that a specific index is related to another set of data. 0 is not a good representation of “Weather”, as 1 is not a good representation of the “Names” of the places with the associated relative climates. Luckily, Ruby has a better way of creating Human friendly pointers to your data. These are called Hashes (or Dictionaries).

3.3 Common Array Methods

3.3.1 Concatenation with Arrays

>> a = ["Scott","Mountain View","Class","Ruby"]

=> ["Scott", "Mountain View", "Class", "Ruby"]

>> b = Array.new
=> []
>> b.push("Code")
=> ["Code"]
>> b.push("Language")
=> ["Code", "Language"]

>> c = a
=> ["Scott", "Mountain View", "Class", "Ruby"]
>> c = c + b
=> ["Scott", "Mountain View", "Class", "Ruby", "Code", "Language"]
>> c
=> ["Scott", "Mountain View", "Class", "Ruby", "Code", "Language"]

In the example above, we create a new Array called (a), we don't explicitly tell Ruby that it is going to be an Array, but let Ruby understand what we mean by our use of the **[ ]** symbols. Ruby got the hint and created a new Array for us and populated it on *instantiation***.**

Next, we create another Array, and we use the very formal Array.new, and then we run a series of **push** commands to the Array. Push is used to append a new element to an Array at the next available index.

Lastly, we merge the two Arrays into a new copy of Array **a** on variable **c.**

>> c = a; c = c + b

Alternative A.
>> c.concat(b)

Alternative B.
>> c << b

Either way you choose is fine. Keep in mind that if it can be done with language specific methods, then those have been tested and the internal algorithms are going to be very speedy and optimized.

3.3.2 Summary of most used Array Methods

<< (bitwise left shift operator), used in the context of an Array will append or concatenate that object onto the end of an Array. (c << b)

== (equality operator), used in the context of an Array, this will let you know if the Array on the Left, is the same as the Array on the Right. (c == b => false)

clear Removes all elements from an Array (c.clear => [])

at(index) Find the array element at a specific index. (c[0] => “Scott”)

delete(obj) Will completely delete an element from an Array.

Delete Example

>> c
=> ["Scott", "Mountain View", "Class", "Ruby", “Code”,”Language”]

>> c.delete(“Scott”)
=> ["Mountain View", "Class", "Ruby", "Code", "Language"]

each Iterate through the Array and run a block of code on each piece within the Array.

Each Example
>> c.each {|x| puts "Element #{x}"}
Element Mountain View
Element Class
Element Ruby
Element Code
Element Language

empty? Test to see whether an Array has any objects within itself.
>> c.empty? => false
>> c.clear
>> c.empty? => true

eql? Test to see if two Arrays are identical.
>> a
=> []
>> a.push("noise")
=> ["noise"]
>> a.push("sweet")
=> ["noise", "sweet"]
>> c = a
=> ["noise", "sweet"]
>> a.eql? c
=> true

include? Does this Array have a particular element within it.
>> c.include? “sweet” => true

inspect Show verbose information on the Object.
>> c.inspect =>"[\"noise\", \"sweet\"]"

length How long is the Array from the first to the last indices.
>> c.length => 2

pop Removes the last element from the Array.
>> c
=> ["noise", "sweet"]
>> c.pop
=> "sweet"
>> c
=> ["noise"]

shift Removes the first element from an Array.

slice!(*args) Will remove the matching index or range.

Slice! Example
>> slice_arr = ["A","B","C","D","E","F","G"]
=> ["A", "B", "C", "D", "E", "F", "G"]

>> slice_arr.slice!(1..5)
=> ["B", "C", "D", "E", "F"]

>> slice_arr
=> ["A", "G"]

Above you see the 1..5, that is Ruby's way of saying Range from x – y, so we say, delete from the Array elements 1,2,3,4,5. Since an Array begins with an index of 0, we save "A", and we are left with just "G" after the Array range from 1..5 is deleted from the Array.

3.4 Hashes
http://ruby-doc.org/core/classes/Hash.html

The Hash Class is basically just another way to store data, just like with an Array. The beauty of the Hash is that it uses keys (pointers) instead of indexes. So now we can create a Hash called @places and actually store "cold","sunny" as our pointers to get back the associated values for that Place.

3.4 .1 Hash Example

>> @places["sunny"] = "Florida"
=> "Florida"
>> @places["cold"] = "Massachusetts"
=> "Massachusetts"

>> @places
=> {"cold"=>"Massachusetts", "sunny"=>"Florida"}

Now if we repeat our sentences from before. First for Florida, then for Massachusetts, it is much easier to understand the first time through.

>> puts "The weather in #{@places["sunny"]} is #{"sunny".capitalize}"
The weather in Florida is Sunny
=> nil

>> puts "The weather in #{@places["cold"]} is #{"cold".capitalize}"
The weather in Massachusetts is Cold
=> nil

3.4.2 Useful Tips for using Hashes

has_key?
This is a Boolean (returns true or false) to let you know that the Hash has the key you are about to use.

Consider this statement.

>> @key = "sunny"
>> if @places.has_key? @key
>> puts "I like the weather in #{@places[@key]}"
>> end
I like the weather in Florida
=> nil

We will get more into statement in the next section on Loops and Iteration, for now think about how this example could be used in the real world. Let's say that you have a registration system (and for some reason you are manually keying in this data!) but you want to check to see if a username has been added. Below is the ruby code to create a Hash for the "user" and store some data about that user in a new Hash Object within that user.

>> @users = Hash.new

=> {}
>> @users["buddy"] = Hash.new
=> {}
>> @users["buddy"]["username"] = "buddy"
=> "Buddy"
>> @users["buddy"]["name"] = "Buddy Holiday"
=> "Buddy Holiday"
>> @users["buddy"]["friends"] = "5"
=> "5"
>> @users
=> {"buddy"=>{"name"=>"Buddy Holiday", "friends"=>"5",
"username"=>"buddy"}}

>> if @users.has_key? "buddy"
>> puts "Reject this new user's username. It has already been taken"
>> end
Reject this new user's username. It has already been taken
=> nil

The example above is by no means something that should actually be used in the context of a website that is going to get real traffic, that approach is a nightmare waiting to happen. However, it makes a good example and a real world use case. It is also useful to note at this time, that the methods you learned for Arrays will actually work as well with Hashes since the Hash and Array are both classes use the Ruby Enumerable module.

3.5 Hashie (gem) https://github.com/intridea/hashie

This following section is optional but will introduce you to the wonderful world of the ruby gem (using RubyGems). I take it you have RubyGems installed, if not, please refer to the **class introduction** pdf.

3.5.1 Get the Hashie Gem

>> gem list
=> shows you the gems you currently have installed

>> sudo gem install hashie
>> password:
=> gems now is installing the hashie gem

The Hashie gem takes the Hash one step further and makes it even more easy to work with. Now instead of key:value pairs (eg. “sunny” => “Florida”) we can use dot syntax to work with our values.

3.5.2 Hashie Usage Example

>> require "hashie"
=> true
>> @places = Hashie::Mash.new
=> <#Hashie::Mash>
>> @places.sunny = "Florida"
=> "Florida"
>> @places.cold = "Massachusetts"
=> "Massachusetts"
>> @places
=> <#Hashie::Mash cold="Massachusetts" sunny="Florida">
>> @places.cold
=> "Massachusetts"
>> @places.inspect
=> "<#Hashie::Mash cold=\"Massachusetts\" sunny=\"Florida\">"

3.5.3 Places Hash to Hashie Retrospect

>> @users = Hashie::Mash.new => <#Hashie::Mash>

>> @users.buddy = Hashie::Mash.new => <#Hashie::Mash>

>> @users.buddy.name = "Buddy Holiday" => "Buddy Holiday"

>> @users.buddy.username = "buddy" => "buddy"

>> @users.buddy.friends = 5 => 5

>> @users
=> <#Hashie::Mash buddy=<#Hashie::Mash friends=5 name="Buddy Holiday" username="buddy">>

3.5.4 Wrap Up on Hashie

Ultimately it is up to you on your approach to using Ruby and Rails as well. There are tons of simple plugins and gems to keep your work load low, but sometimes you get into a bigger pickle when you haphazardly take the "easy way out". General rule of thumb, if your going to use a plugin or gem, read about it first and make sure it is reputable.

3.5.6 Exercizes

1. Create an Array on a new instance variable.

2. Populate your new Array with a collection of your Interests / Hobbies.

3. Iterate through your Array using the **each** method and print out to the terminal a nice list of your hobbies.

4. Create a new Hash Object called @my_hobbies and populate this hash with your Hobbies using the symbol "hobbies".

5. Write a sentence to the terminal where you pick a specific hobby from the Hash of hobbies and use the symbol "hobbies" as text in your sentence.

4 Operators

Operators are tokens or symbols in the Ruby programming language that represent an operation, such as Addition, Subtraction, Comparison and more.

4.1 Math Operators

Addition +
>> 2 + 2 => 4

Subtraction –
>> 4 - 2 => 2

Multiplication *
>> 2 * 3 => 6

Division /
>> 4 / 2 => 2

>> include Math
>> 16 / Math.sqrt(4) => 8.0

Modulo (remainder) **%**
>> 2 % 2 => 0
>> 3 % 2 => 1

Exponents **
>> 2 ** 3 => 8

Arithmetic Sequences
>> (2*2)4** => 256

4.2 Boolean Operators (useful for comparison)

And &&
>> if a && b …

OR ||
>> if a || b …

4.3 Equality

== (is equal to)

=== (is exactly equal to, this Object is the same as itself)

!= (not equal to)

4.4 Conditional Operator
?:

>> x = 90
>> x < 90 ? "true" : "false"
=> "false"

The example above is sometimes a difficult concept to grasp. The Conditional Operator looks to see if the expression **x < 90** is true or false. If the variable **x** is less than **90** then continue this block of code using the first expression after the **?**, in this case that is just a simple String “true”, however, since we specified that **x** is equal to **90** this will run the second expression, which is the String “false”.

>> x < 90 ? "The variable X is less than 90" : "The variable X is greater than or equal to 90"
=> "The variable X is greater than or equal to 90"

4.5 Conditional and Loop Modifiers

defined? Test to see if a variable or type exists

not Same as Boolean Not (!=)

and or (Same as Boolean **&&** and **||**)

if unless while until Conditional and Loop modifiers

rescue Exception Handling modifier

4.6 Assignment

= (equals)
>> k = 2

****= (exponential)**
>> k = 2; k **= 2 => 4

***= (value multiplied by the right side)**
>> k=2; k *=6 => 12

/= (divide by the right side)
>> k = 16; k /= 4 => 4

+= (plus the right hand side)
>> k = 2; k += 6
=> 8

-= (minus equals value)
>> k = 14; **k -= 6**
=> 8

Wrap Up
Ruby provides many different options for doing comparisons and there are more Operators than are covered above, these should help you along your way. If you want to know more about the Ruby Operators, you can refer to the Ruby Docs or to the great book “The Ruby Programming Language”, Chapter 4 part 6 on Operators.

5 Working with **Dates** and **Time**

No language would be complete without a series of methods and classes for dealing with Dates and Time. People tend to skip working with dates and time since is easy to get by with the basics, and typically people only need to figure out simple stats like what the current date is in M-D-Y format (Month, Date Year).

If you are like most people, then feel free to skim this section, but if you are interested in learning the skills to say create a Membership system on a website that regulates user's ability to access certain pieces of a website unless their subscription is up to date, or perhaps your intent is to keep track of the total time a user has been active on your website, or one of the many other million reasons to learn more than the basics, than indulge yourself in Ruby's Time class.

6.1 Time

Time in Ruby is robust and chock full of many useful helper methods that all wrap themselves around the server time (computer time) on the machine running the Ruby interpreter.

6.1.1 How to work with Time

Time in Ruby takes a call to the main Time class. This returns a reference to Time, and from there the options are endless.

Here are the basic methods.

```
>> current = Time.now
Fri Nov 05 20:26:12 -0700 2010

>> current.year
2010

>> current.month
11

>> current.day
```

Given a reference to a Time object, retrieval of your basic information is a breeze. Consider the following. You have built your shiny new website and decided that you wanted to greet a User and also let them know the current Day in the most human readable manor possible.

6.1.2 Ugly Example

```
>> @today = Time.now
>> @user = “Joe Thompson”
>> printf “Welcome, %s. Did you know that it is %s” %
[@user,@today.to_s[0,10]]
Welcome Joe Thompson. Did you know that it is Fri Nov 05
```

The example above is “ugly” because it could have been using Ruby’s Time methods to create friendly representation of the day’s Date.

6.1.3 Ruby’s Saving Grace

```
>> @today = Time.now
>> @user = “Scott Haines”
>> puts “Welcome Back %s. Today is %s” % [@user, @today.strftime("%A,
%B %wth %Y")]
“Welcome Back Scott Haines. Today is Friday, November 5th 2010”
```

The strftime takes the value of the Time object, turns it into a String and applies Formatting to that string. If you have worked in other programming languages than you know what a saving grace this function is. With PHP you would have to create a new Date object, and if you didn’t happen to find a Formatting class, or know how to format the Date, then you would have to create a series of lookup Arrays. Say one for days of the week (formal) and another for days of the week (informal), another to cross reference the year, is a leap year – well have to add some logic for that, etc.

Bottom line, Ruby makes it fun to work with the Time.

6.2 More from the Time class

```
# current hour
>> @today.hour
20

# current minute
>> @today.min
32

# current second
>> @today.sec
10

# current day of the week (number)
>> @today.wday
5

# current day within the year (number out of 365 or 366 if leap year)
>> @today.yday
309

# view an Array of the current Date
>> @today.to_a
[10, 32, 20, 5, 11, 2010, 5, 309, true, "PDT"]
```

6.3 Real World Example

Let's say you have a website and you have created a registration system with Premium upgrades to each account. You store reference to the creation date timestamp of a new Premium subscription in a database that is associated on a one-to-one basis with each of the users in your system.

So just to paint a picture of this we have a User who is associated to a Membership of a particular level. When that User makes a successful payment, you update the Membership table in your database for that User with the timestamp of their last successful payment.

Here goes, a simple class that takes a reference to a User's last successful payment, a Membership period to test against the current timestamp (say 30 days, 90 days, 120 days), and this will produce a simple "Your In", or "Pay Up Deadbeat".

Open up **Membership.rb** and follow along.

Membership.rb defines the main Membership class as well as an example sub-class called LetMeIn. LetMeIn is just there mostly for show, to start to expand upon what is possible in Ruby.

Membership has one main method called **am_i_good?**. What this method does, is a.) define a User if this class has not been instantiated, otherwise, the user defaults to nil and is instead pulled in from the Class variable @@user (which is set on initialization).

```
@@user = user if user.is_a? Hashie::Mash
```

The second thing we do is pull the current timestamp with **Time.now**. Afterwards, we get a reference to the User's last successful payment, which was defined in the Hashie::Mash object on instantiation.

```
last_real_payment = @@user.membership.payment_date
```

(*Note. This Class depends on a HardCoded Hashie Object, it is up to you to define how you want to work with data and this is only one of many options)

Then we create a second instance variable to hold the User's last payment date (String) as a Time object. We rely on the Ruby built-in Helper class called **parsedate** to do so.

Lastly, we get to the logic. We are presenting a simple logic to test if the User should be allowed into your Premium content area, we state "if the User's last payment date is within the subscription period, then let them in, otherwise, tell them to Pay Up".

```
if ((last_payment_as_time.to_i + @@expiration_period*60*60*24) >= time.to_i)
   puts "Welcome In"
else
   puts "Payup Deadbeat"
end
```

And there we have it. A simple class for testing whether or not a User can access a particular set of contents on your website.

6 Loops, Iteration, Mapping

A computer program at is very core is a task. Whether that task is simple or extremely advance, it starts and ends when it has completed its task. Most conventional programming languages make this easy for the programmer and Ruby has made it very simple but also provided many different ways to attack a problem and come to an optimized solution.

We will take a look at the way Ruby approaches looping, iteration and mapping data. Using the information we learned from the section on Operators, we can now create conditions in our Ruby code.

6.1 Conditions

6.1.1 Conditional **if**

```
if expression
  run this code
end
```

Example.

```
x = 2
if x < 10
 x += 1
end
```

What we see above is a very straight forward, If **x** is not greater than or equal to **10** then add **1** to **x** until it is equal to **10**.

6.1.2 Conditional **else**

```
If expression
 run this code
else
 run this code
end
```

Example

```
word = “Scott”

if word == “Scott”
  puts “hi Scott”
else
  puts “Sorry. You are not Scott”
end
```

In the example above, we look at a basic Conditional Block of code. We set the variable **word** to **Scott** and then we run a test against the variable **word.** If **word** is equal to **Scott** then say **hi Scott** otherwise say **Sorry. You are not Scott**.

It is worth mentioning hear why I chose to test the equality to true and our else case as false. In Ruby and other languages there is the concept of failing closed. So consider this example. The word variable is a test against say a password in a secure database, If I write the same code in the following way we have some issues.

```
If word != “Scott”
  run code to take the user back to the main login page
else
  run code to take the user to the secure credit card number lookup section
end
```

The example above is dumb, what if there is an issue and word is **nil**. What if something happens and this first statement didn’t run properly, you would be leaving your secure site open to any number of “whoops” cases.

6.1.3 Conditional **elsif**

```
age = 26

if age > 26
   puts “Welcome to the above 26 section”
elsif age == 26
  puts “Wow, you are 26”
```

```
else
  puts “Doesn’t look like your 26 yet”
end
```

Above we add the **elsif** to our Ruby expression vocabulary.
elsif literally means else if (otherwise if this is true, do this instead).

```
If age is greater than 26
  welcome the user to the above 26 section
elsif age is conditionally equal to 26
  express joy in the fact that the user is 26
else
  let the user know you know they are not 26
end
```

The ability to test many different “conditions” is the first key step in creating code that can solve large tasks, and validate any number of unique inputs or situations.

6.1.4 Conditional **unless**

The **unless** keyword is fairly straight forward, **unless** the following is true, do nothing or do something else.

Examples

```
@name = “Ruby”

unless @name == “Ruby”
  puts “You made it”
end
```

--

```
@name = “Ruby”
```

```
unless @name == “Ruby”
  puts “You made it”
else
  puts “You failed”
end
```

6.1.5 Conditional **case**

Cases are not unique to Ruby. In Javascript and PHP you see the following.

```
switch(condition){
  case “something”:
    some code
  break;

  default:
    default code
  break;
}
```

In Ruby however, you can run these case conditions with a little more grace and ease. Consider the following, We want to create a custom Welcome message for a user based on their past activities on a website, maybe “Welcome Super-User <username>” for a regular user with more than 20 days logged into a website, or “Welcome Back, <username>” for someone who hasn’t been on a website for a while and logs back in, or perhaps “Welcome <username>, click here to tour our site” for a first time registered user.

Case Example

Please take a look at custom_user_welcome_messages_5-1-5.rb. You can see we built a Ruby Class at the top of the actual file, and are *instantiating* that same class within the Ruby file. It is up to you to decide how you want to separate your code, but it is easier when creating large Ruby applications if you separate your Classes by function (what does this code do).

As you can see, once we have the Class created, working with it is simple.

```
n = SayHello.new("Scott",3)
n.greet_me

>> Welcome Super-User Scott. We love you

n = SayHello.new("Scott",2)
n.greet_me

>> Welcome Back! Scott
```

6.2 Loops in Ruby

The ability to loop through a series of expressions based off of an unlimited number of various conditions is a simple but powerful mechanism for creating real-world applications.

Looping in Ruby is similar in logic to most programming languages, and there a just a few key fundamentals to learn before you can jump into creating and tackling new and interesting problems with the Ruby language.

6.2.1 The **for..in** Loops

Example.

```
>> @ages = [10,11,15,20,24,26,90]
=> [10, 11, 15, 20, 24, 26, 90]

>> for @i in 0..@ages.length do
?> puts "In the Stack #{@i}"
>> end
In the Stack 0
In the Stack 1
In the Stack 2
In the Stack 3
In the Stack 4
In the Stack 5
```

In the Stack 6
In the Stack 7

The code above takes an Array of possible ages, and uses the length of that Array to iterate through the Array and just print out how many times this expression has looped.

A better example actually shows you what information is currently at a particular index within that Array as we are looping through it. Let's now take the same code from above, but Change the output within the Loop to say "The Index in the @ages Array is currently **@i** and that corresponds to **@ages[@i]**"

```
>> for @i in 0..@ages.length do
?> puts "The index in the @ages Array is currently #{@i} and that corresponds to #{@ages[@i]} in the @ages Array"
>> end
```

The index in the @ages Array is currently 0 and that corresponds to 10 in the @ages Array
The index in the @ages Array is currently 1 and that corresponds to 11 in the @ages Array
The index in the @ages Array is currently 2 and that corresponds to 15 in the @ages Array
The index in the @ages Array is currently 3 and that corresponds to 20 in the @ages Array
The index in the @ages Array is currently 4 and that corresponds to 24 in the @ages Array
The index in the @ages Array is currently 5 and that corresponds to 26 in the @ages Array
The index in the @ages Array is currently 6 and that corresponds to 90 in the @ages Array
The index in the @ages Array is currently 7 and that corresponds to in the @ages Array

This is much nicer, when you see the output on the screen you feel like you have a better idea of what is going on as the loop iterates through the @ages Array.

6.2.2 Looping meets Conditions

We are going to modify the for in loop example from above to include a conditional test for a specific age within the @ages Array, if this condition is met, we are going to change the text that is written to the terminal.

Please open up the file **looping_and_conditions_5-2-2.rb**. You will see a chunk of code that takes a @target (which is an arbitrary number) and then uses this number as a condition to write out a custom message to the terminal.

```
>> ruby looping_and_conditions_5-2-2.rb
Still waiting for my conditions to be met
Still waiting for my conditions to be met
Still waiting for my conditions to be met
Still waiting for my conditions to be met
Still waiting for my conditions to be met
Excellent. These conditions have been met
```

You will see that we use the **break** keyword in this example. This is done to stop the loop and return back to the code itself. This is best so that you can optimize your code. Say you had 2 million names and you were going to iterate through those names to try and find a particular person. If that person was the second result, you would be wasting precious CPU cycles on looking for something you already have.

6.2.3 The **while** Loop

This is probably one of the most popular loops because of its simplicity.

```
while condition do
  run code
   alter condition
end
```

Example

```
x = 13

while x < 26 do
  puts “Adding to your variable #{x.to_s}”
  x += 1
end
```

```
>> x = 13
=> 13

>> while x < 26 do
?> puts "Adding to your variable #{x.to_s}"
>> x += 1
>> end

Adding to your variable 13
Adding to your variable 14
Adding to your variable 15
Adding to your variable 16
Adding to your variable 17
Adding to your variable 18
Adding to your variable 19
Adding to your variable 20
Adding to your variable 21
Adding to your variable 22
Adding to your variable 23
Adding to your variable 24
Adding to your variable 25
```

6.2.4 The **until** Loop

Just as we have the **while** loop, which waits for a condition to be met, we also have the **until** loop, which also waits for a condition to be met. The **until** loop works by only evaluating the expression following the **until condition** if that is not true.

```
>> x = 10
>> until x == 20 do
?> puts “Adding to X.”
>> x += 1
>> end
Adding to X
Adding to X
Adding to X
Adding to X
Adding to X
Adding to X
Adding to X
Adding to X
Adding to X
Adding to X
```

(*Note. Running loops in irb gives you a good idea of how Ruby works. Think about how this changes your understanding of the language. You know now that a loop or expression that is appears below is only executed after the loop has finished running its own tests. You can consider this to be blocking the Ruby interpreter from continuing until the Loop has completed.)

6.3 Iteration

This is a topic that probably deserves much more than a simple explanation, but you will “get” the Ruby way with time. An iterator is a control structure that allows you to write complex code with very little complex code.

For example,

```
>> 12.times { |x| puts “hello iteration interval #{x.to_s}” }
hello iteration 0
hello iteration 1
hello iteration 2
hello iteration 3
hello iteration 4
```

hello iteration 5
hello iteration 6
hello iteration 7
hello iteration 8
hello iteration 9
hello iteration 10
hello iteration 11

Above we are using the **times** iterator. It takes an Integer (number) and will repeate the code block following that Interger exactly as many times as the Interger value. That is a mouthful. We add the |x| into the start of the code block, this allows us to see what is happening in each cycle of the loop. This same function could be do in an **until** loop or any other looping method, but in more lines.

(*Note. The **{** braces character is the same as saying **do**. We could have written the expression above as follows.

```
>> 12.times do |x|
?> puts "hello iteration #{x.to_s}"
>> end
```

However, as you can see. One line of code is better than three, albeit less human readable.)

Next we see how to **map** a series of values to the iterator loop.

```
>> ("a".."z").map {|letter| puts "This is the letter #{letter}"}
This is the letter a
This is the letter b
…
This is the letter z
```

This example uses the power of the ruby **Range** class coupled with the **map** iterator. Think about having to create an **Array** of the letters of the alphabet, and then have to loop through that Array to create the same expression.

```
(*Note. Looping through manually.
>> alphabet = [“a”,”b”,”c”,…”z”];
>> for @i in 0..alphabet.length do
?> puts “This is the letter #{alphabet[@i]}”
>> end
)
```

It is worth mentioning that earlier we encountered the **each** method while looking at **Array**s and **Hashes**. We said, for each the given Array/Hash elements, lets run the following code once.

```
>> (“a”..”z”).each {|x| puts “another alphabet example %s” % x}
another alphabet example a
another alphabet example b
…
another alphabet example z
```

Each of these iterator examples uses Ruby’s **yield** method. Basically, this works by saying “given these initial conditions, say repeat 4 **times**, I plan on running the following expression, say outputting my current position in this iterator – block cycle. After I have completed stating my current position, I will return to the see if I need to repeat this task again, if not then I will stop.

The iterator works much like the **until** or **for in** loops, however it is very tidy and with a little creativity can save you the agony of typing more than you have to.

Wrap Up

We started out looking at Variables in Ruby, moved onto working with Strings, then into dealing with Collections of Data.

We skimmed some topics like Conditions and Looping while looking at how to work with the Data inside our Arrays and Hashes.

We then moved on to look at Ruby Operators (Math Functions, Booleans, and Condition Operators).

After we had an understanding of the vocabulary needed to create expressions in Ruby (Operators), we followed that up by actually creating loops and conditional expressions in Ruby.

Where does the lead to?

It is my hope that the information presented up to this point would create an eye opening tour of the Ruby language. One that would graze just enough of the core concepts to really get the average person excited about learning more and help guide a new Rubyist in their next steps with the language.

In the next section we are going to take what we have learned so far and create a Chat Room. We will be running a Chat Server and create Chat Clients that will allow a user to connect to the Chat Server, write a message to everyone, or just a specific Chat Channel.

7 Real Time Chat

Real Time Chat is a combination of what we have learned so far, as well as some new code that you will soon understand. It is important to sit back and design your ruby applications before you dive into them, and this is what we will be doing now.

7.1 Designing an Application

We know that we want to create a Chat application. But before we concern ourselves with how this will be written from a code perspective, we must first settle down and write How this should work.

7.1.1 Modeling your Application

Let's break down the specific tasks we must accomplish.

1. Run a Chat Server
    a. Allow a User to Login

    b. Capture the User's information (username, socket, ip)

    c. Allow a User to create a Channel (think of this as a private information gateway)

    d. Keep track of what kind of Channel a User has created and logged into

    e. Keep track of who is allowed within a particular Channel

    f. Once logged into the Chat Server. Allow Messaging between end points.

2. Create a Chat Client

    a. How does a User log into the Chat Server?

b. How does a User add text to be sent to the Chat Server?

c. How does a User specify whom they want to messages to go to?

d. How does a User log off of the Chat Channel? Server?

3. How to notify User

a. Send User info about new Chat Messages

b. Let a User know when someone has left their Channel

c. Prompt a User to allow a new User to enter the Chat Channel

7.2 Defining your Classes and task Delegation

Chat.rb
This class will be in charge of creating the Chat Server and storing global variables such as Channels, Users as well as simple helper methods.

ChatChannel.rb
This class will be in charge of all Tasks that relate to a ChatChannel. These tasks include storing a reference to the current Channel (@channel_name), storing an Array of User Hashes (@users) that will help to notify a particular user when a message has come in. ChatChannel will also keep track of who is allowed in a particular Channel by marshaling the global $users_allowed Array.

ChatClient.rb
This will be the User's gateway into the Chat application. Here a User will login with their username (completely unique) and be welcomed to the Chat application. They will then be given an option to join a ChatChannel, or to just stay on the main ChatChannel. They will then view messages as they come in over the Socket.