Why Ruby ?

it's enjoyable - "The curse of Ruby...."

it's succinct - "I hate Ruby..."

it's fun - the ruby community is very sociable and friendly

Why Rails ?

It's a complete stack

- Rails web framework
- RSpec Cucumber for testing
- Capistrano for deployment
- New Relic & Airbrake for monitoring

It's quick

Coding by convention means that there's no boiler plate code to write.
Large library of gems available:

Connectors for databases including MySql, SqlLite, SqlServer, Oracle, DB2, CouchDB and MongoDB.

- **Paperclip** manages uploading of file attachments and connects with ImageMagik to resize uploaded images.
- **Delayed_Job** makes method calls almost seamlessly asynchrous so that heavier tasks are managed in the background so responses return quickly.
- **Skynet** is a pure Ruby map reducer (like Hadoop) which manages tasks across many servers.
- **Will Paginate** handles finding and displaying of paginated results.
- **Bluecloth** and **Redcloth** handles Markdown so that text input can be converted to/from
- HTML.
- **Log4R**, logging
- **Nokogiri** parses HTML used for simple web scraping.
- **Mechanize** sits above Nokogiri, a headless browser so that you can navigate websites, fill forms, log-in etc.
- **Devise** manages user authentication
- **Rspec** for expressive testing of code
- **Factory Girl** for crating mock testing objects
- **Capistrano** for auto-deployment of code and database scripts
- **Bundler**, used in Rails 3 as a package manager for individual applications rather than the Ruby environment for that computer.

Last but not least there's the 'acts_as_hasselhof' plugin.

http://rubygems.org/

1. Ruby Language

The documentation is available online as Rdoc.

Split into core:

http://www.ruby-doc.org/core/

for things like String, Time, Date, Regexp, Object and Standard Library

http://www.ruby-doc.org/stdlib/

for things like IO, Net, Socket, Thread

It's worthwhile bookmarking these for reference

1.1 Running Ruby Applications:

1.1.1 Running Ruby Scripts

Ruby can be run from command line scripts or inside the IRB shell.

Exercise 1. Hello World

1. create a file called helloworld.rb with the line puts “hello world”
2. From a command prompt run ruby helloworld.rb

1.1.2 Running Ruby inside IRB

Ruby can also be run inside the interactive IRB Shell which is a great way to get a feel for the language and also to test simple things out. We'll be doing that in the next section.

1.2 Variables

Variables are declared simply by creating the variable name

Exercise 2.

```
x = 5
y = "6"
z = nil
```

Ruby variables may not be type safe but just try to add numbers, strings, nil:

```
x + y
y + z
x + z
```

You will get an error if you call a variable that doesn't exist: type a new variable name such as d and hit return

Everything is an Object even nil and numbers

get the class of objects x, y & z by typing

```
x.class
y.class
z.class
```

Now get the .superclass on the classes of x, y, z. Then keep going on Fixnum until you get Object.

```
5.class.superclass.superclass.superclass
“6”.superclass
nil.superclass
```

1.3 Some Ruby Basics

1.3.1 Symbols

'primitive' Objects used as references in code instead of more memory hungry strings.

```
:mysymbol
```

1.3.2 Hashes

lie at the heart of Ruby, Rails and the Rails ORB ActiveRecord. The lack of strongly typed classes lets us pass data around conveniently in hashes.

Exercise 3

Create a script called myhash.rb and run it from the command line

```
myhash = {:name => 'steve', :email => 'steve@steve.com', :eyes => "blue"}

puts "my name is #{myhash[:name]}"

myhash.map do |key, value|
    puts key
    puts value
end
```

1.3.3 Arrays

Exercise 4

create a script called myarray.rb and run it from the command line

```
myarray = [1, 2, 3, 'joe', true]
myarray.each do |value|
    puts "array value is #{value} with class #{value.class}"
end
```

1.3.4 Ranges

Analogous to a for loop - create and run myrange.rb

Exercise 5

create a script called myrange.rb and run it from the command line

```
(1..100).each do |num|
    puts num
end

('a'..'z').each do |char|
    puts char
end
```

note the syntax around .each, this is a closure & block, we'll cover that soon.

1.3.5 if, unless & case

Exercise 6 create a script called myif.rb enter the code below and run it from the command line providing arguments for x and y e.g. ruby myif.rb 3 17

Before you do:

NOTE Ruby uses elsif NOT elseif (this still catches me out).

NOTE ALSO unless is the negative of if but there is no elsunless, try working one out in your head and you'll see why...

```
x = ARGV[0].to_i
y = ARGV[1].to_i

if x < 5
   puts "less than 5"
elsif x > 20
   puts "#{x} greater than 20"
else
   puts "inbetween 5 & 20"
end

puts "x is less than 5" if x < 5

unless x > y
   puts "#{x} is less than #{y}"
end

puts "#{y} is less than #{x}" unless y > x

case x
    when 5
     puts 5
else
     puts “else”
end
```

1.3.6 begin rescue always

```
begin
      # -
rescue OneTypeOfException
      # -
rescue AnotherTypeOfException
      # -
else
      # Other exceptions
ensure
      # Always will be executed
end
```

1.3.7 while do, begin end while

```
i = 0
while i < 5
   puts i
   i += 1
   break if i == 2
end

i = 0
until i == 5
   puts i
   i += 1
end

puts i += 1 until i == 5 #a handy inline until

n = 0
begin
      n += 1
end while n < 100
```

1.3.8 nil, .nil? And .empty?

No script this time but open IRB and try the following noting the return values IRB gives you.

NOTE the .nil? and .empty? are methods that follow the common Ruby idiom of having a ? at the end of method calls that return true/false

Exercise 7

```
a = nil
b = 5

a.nil?
b.nil?

c = {} #hash
d = {:a => 1}
e = [] #array
f  = [1]

c.empty?
d.empty?
e.empty?
f.empty?
```

NOTE also that Rails provides a third method .blank? That can be used to determine 'empty' strings, arrays and hashes e.g. “”, “ ”, nil, [], and {}. This is a 'core extension' to the Object class provided by Rails (so not available in IRB which is Ruby only). See monkey patching later on.

1.5 Ruby OO

Single inheritance, method overriding & namespaces.

No interfaces (since it's weakly typed) but mixins allow for standard sections of code to be included & reused. Multiple mixins are allowed.

Does not allow method overloading but methods can have optional arguments.

It has class (or static) variables and methods

Ruby Object class has to_s, eql?, clone, inspect to be overridden as required

Also Ruby OO has very useful meta-programming techniques that allow classes and instances of classes to be created, managed and altered at runtime. This takes some getting used to if you're from a strictly typed language but with a little practice it can be a pragmatic way of writing idiomatic code.

Exercise 8 create a file called south_park_character.rb

```
module SouthPark
  class Character

    CREATORS = ["Trey Parker", "Matt Stone"]  #this is a constant, they must start
with a capital letter but are commonly all capitals.

    @@series = nil  #class variables have to be initialized
    @@episode = 14

    #this is the constructor with an optional argument.
    def initialize(name, args={})
        @name = name
        @age = args[:age] || 10.5
        @catchphrase = args[:phrase] || "Oh my God, they killed Kenny (#{@name} -
aged #{@age})"
    end

     def name=(value)
       @name = value #'setter' initializes instance level variable if it hasn't been
already
     end

     def name
       @name #ruby will always return the last object in a method
     end

     #but rather than write out all the setters and getters this method defines them
around    @variables of the same name

     #NOTE, by convention attr_accessor goes at the top of the class
```

```
    attr_accessor :age, :catchphrase, :height

    #a class or static method can be defined by start self.method_name
    def self.episode=(value)
      @@episode = value
    end

    #a class method can also be defined by using the class nameinstead of self
    def Character.episode
      @@episode
    end

  end #end class
end #end module
```

1. Open IRB in the same folder that has the class file

2. Load the code into IRB:

   'load south_park_character.rb'

3. Create some characters:

   cartman = SouthPark::Character.new("cartman", :catchphrase => "respect mah authoritah")
   kyle = SouthPark::Character.new("kyle", :age => 8)

4. Change the episode for all your characters:

   SouthPark::Character.episode = 15

5. Type

   creators = SouthPark::Character::CREATORS

NOTE!! the two colons used to access the constant CREATORS

6. Type 'cartman' or any other character you've created and hit return. You should see output something like this:

   <SouthPark::Character:0x0000010129f480 @name="cartman", @age=12, @catchphrase="Oh my God, they killed Kenny (cartman - aged 12)">

Now type

   cartman.height = 5

then examine the character again by typing

```
cartman.inspect
```

Note that the @height instance variable has been initialized

7. Go back to cartman.rb script and create another class called Canadian and have Canadian inherit from SouthPark::Character. If a phrase hasn't been provided we'll override the default value from "Oh my God..." to "Aboot"

```
class Canadian < SouthPark::Character

  def initialize(name, args={})
    args[:phrase] = "Aboot" unless args[:phrase]
    super(name, args)
  end

end
```

'load cartman.rb' in IRB will reload the script then create an instance of Canadian. Have a play with some of its values.

1.6 Closures

Allow us to reuse code whilst adapting it's behaviour for specific uses. They're a neat way to apply logic to collections for example and inject logic into complex error handling.

1.6.1 Closures & Loops

Exercise 9

Create a script called closures and run each one by loading the file again as you build the script. Can you guess how #3 & #4 are working ?

```
myrange = (1..20)

#1
myrange.each do |num|
    puts "each num is #{num}"
end

#2
myarray = myrange.collect do |num|
   num + 100
end

puts "myarray after collect is #{myarray.inspect}"

#3
myarray = myrange.select do |num|
   num % 3 == 0
end
```

```
puts “myarray after select is #{myarray.inspect}”

#4
myarray = myrange.inject(0) do |memo, num|
    memo + num
end

puts “myarray after inject is #{myarray.inspect}”
```

Closures are simply methods that accept a Block of code as an argument. In #3 above, we're calling the select method and passing the block of code {|num| num %3 == 0} as an argument in much the same way we'd pass a string or an integer.

Exercise 10 We'll create our own closure method to see what Ruby is doing

Create a script called simple_closure.rb

```
class Closure2 < Array

  def collect2 &block # the ampersand tells us it's a block of code
     newarray = []
     self.each do |num|
       newarray << yield(num)
     end
     newarray
  end

end
```

load the script and:

```
c = Closure2.new
c << 1
c << 2
c << 3

c.collect2 do |num|
        num + 1
end
```

1.6.2 Closures & Injection

Closures aren't just handy ways of manipulating collections. Consider the following Rails code (this is not an exercise):

```
def paranoid_response &block

    begin
        object = &block.call
        render :json => object.to_json
    rescue RecordNotFoundException => e
        render #something appropriate
    rescue RecordNotValidException
        render #something appropriate
    rescue IOException => e
        render #something appropriate
    end

end
```

This hierarchy of exception handling could easily be required in many parts of a web application but this method can be effectively reused:

```
def index
    paranoid_response do
        Person.find(1)
    end
end
```