Jane’s code cheat sheet

Author



Table of Contents

[Bash 6](#_Toc451756223)

[Working with Ruby in the Terminal 7](#_Toc451756224)

[Reserved Words 8](#_Toc451756225)

[end 8](#_Toc451756226)

[false 8](#_Toc451756227)

[yield 8](#_Toc451756228)

[Printing to the console – puts, prints, p 9](#_Toc451756229)

[Data types 9](#_Toc451756230)

[Symbols :name can convert – to string .*to\_s* OR to symbol *.to\_sym* 9](#_Toc451756231)

[Numbers = 3 types – how do you use these??? 10](#_Toc451756232)

[Methods or Functions 10](#_Toc451756233)

[Comparison Operators 11](#_Toc451756234)

[Variables ruby = “is nice” 12](#_Toc451756235)

[Variable Interpolation in Strings ie putting variables into strings *#{ name }* 12](#_Toc451756236)

[Comments in Ruby = - # 12](#_Toc451756237)

[Getting User Input puts + variable = gets.chomp 12](#_Toc451756238)

[Controlling the flow 13](#_Toc451756239)

[If statements – if…end, if elsif else end… 13](#_Toc451756240)

[Unless statements 14](#_Toc451756241)

[Case statements 15](#_Toc451756242)

[While loops –Don’t Use these 15](#_Toc451756243)

[Until loops until ….end 15](#_Toc451756244)

[Iterators 16](#_Toc451756245)

[Blocks – content in iterators – ie between do and end or {} 16](#_Toc451756246)

[Do and end delimit a code block 16](#_Toc451756247)

[Common way – 5.timse do …end or 5.downto(0) 17](#_Toc451756248)

[*For loops (No one ever uses these)* 18](#_Toc451756249)

[Generating Random Numbers 18](#_Toc451756250)

[Arrays *[]* 18](#_Toc451756251)

[*Creation of an Array* - = [ , ] OR array.new OR array.new ( 3) or array.new (3, true) or array.new(4) {array.new(2)} 18](#_Toc451756252)

[Accessing Elements – arr[2] OR arr [-3] OR arr.at (0) OR arr.first OR arr.last OR arr.take(3) OR arr.drop (3) or arr.fetch(100, error) 20](#_Toc451756253)

[Adding Items to an Array [ , ] – .push or << or .unshift or .insert ( 3, “item” ) 21](#_Toc451756254)

[*Removing Items from an Array[ ]* – arr.pop OR arr.shift OR arr.delete\_at( 2 ) arr.compact??? OR arr.uniq (remove duplicates) 21](#_Toc451756255)

[Changing elements in an array – arr[0] = 1 23](#_Toc451756256)

[*Iterating Over Arrays - - each do end OR* arr.each { |el| puts el } OR arr.reverse\_each OR arr.map 23](#_Toc451756257)

[*Selecting Items from an Array – arr.select OR arr.reject OR arr.delete\_if or arr.keep\_if* 25](#_Toc451756258)

[Array comparisons (Venn diagram)– array1 | array2 OR array1 & array2 OR array1 - array2 26](#_Toc451756259)

[Destructive Methods vs Non-destructive Methods – destructive usually end with ! 26](#_Toc451756260)

[Predicate Method – usually return booloean – usually end in ? 26](#_Toc451756261)

[Object IDs 27](#_Toc451756262)

[False Interlude 27](#_Toc451756263)

[Hashes 27](#_Toc451756264)

[Creation of a Hash –*hash = { } OR hash = { :name => “serge, etc} or Hash.new* 27](#_Toc451756265)

[Accessing Elements – *hash [:key] OR hash [“name”]* 29](#_Toc451756266)

[Adding to a hash serge[:counterpart] = "Jane (temporarily)" 30](#_Toc451756267)

[Removing from a hash hash.delelte(:key) 30](#_Toc451756268)

[Iterating over hashes *hash.each do || ….end – different variation depending on what want to return – all keys and values, some keys and values, specific keys and values* 30](#_Toc451756269)

[Embedded Ruby 32](#_Toc451756270)

[Classes 32](#_Toc451756271)

[classes - blueprints 32](#_Toc451756272)

[CREATING CLASSES AND adding methods 32](#_Toc451756273)

[Create 32](#_Toc451756274)

[Add methods 33](#_Toc451756275)

[Create instance of class to be able to call the methods 33](#_Toc451756276)

[Getters and setters – to get and set values on a class 34](#_Toc451756277)

[Shortcuts for getters and setters 34](#_Toc451756278)

[Seeing what we have defined 34](#_Toc451756279)

[Sinatra 34](#_Toc451756280)

[Routing 35](#_Toc451756281)

[Views / Templates – ERB – embedded ruby 36](#_Toc451756282)

[Understanding the web 37](#_Toc451756283)

[Routing 38](#_Toc451756284)

[SQL – using Sqlite3 38](#_Toc451756285)

[CRUD – create, read, update, delete 38](#_Toc451756286)

[Parts of a DB 38](#_Toc451756287)

[Steps to create a DB 39](#_Toc451756288)

[Working with the DB from the Console 40](#_Toc451756289)

[Importing the database 40](#_Toc451756290)

[Importing the sql file 40](#_Toc451756291)

[ADD RECORDS – “Create.SQL” file + SQL command + import into DB 41](#_Toc451756292)

[Read records – READ.SQL file + Sql commands 41](#_Toc451756293)

[UPDATE STEP – create update.sql file + SQL command + add to DB via the console 42](#_Toc451756294)

[DELETE STEP – create delete.sql file + SQL command + add to DB via the console 42](#_Toc451756295)

[Overall structure of CRUD application 42](#_Toc451756296)

[Files 43](#_Toc451756297)

[COMMANDS 44](#_Toc451756298)

[Object Oriented Programming (OOP) 46](#_Toc451756299)

[Active Record 46](#_Toc451756300)

[Preface - Object Relational Mapping 46](#_Toc451756301)

[What does Active Record do? 46](#_Toc451756302)

[Convention over configuration – names to sort out DB associations 47](#_Toc451756303)

[How do you work with Active Record 47](#_Toc451756304)

[CRUD with Active Record 47](#_Toc451756305)

[Create 47](#_Toc451756306)

[Read 48](#_Toc451756307)

[Update 49](#_Toc451756308)

[Delete 49](#_Toc451756309)

# Bash

|  |  |
| --- | --- |
|  | *Command* |
| *Getting into bash* | *subl ~/.bash\_profile* |
|  |  |
|  |  |

RUBY

# Working with Ruby in the Terminal

ruby -v - Will return the current version of Ruby

which ruby - What is the path to the version of Ruby you are using

ruby hello\_world.rb - Runs the hello\_world.rb file

+

irb - Runs a ruby console

pry - Runs a better console

<CTRL> + D - Ends a file in irb or ruby

To quit IRB, type “quit”, “exit” or just hit Control-D

Coding with Ruby

# **Reserved Words**

The following words are reserved in Ruby: - see http://ruby-doc.org/docs/keywords/1.9/

\_\_FILE\_\_ and def end in or self unless

\_\_LINE\_\_ begin defined? ensure module redo super until

BEGIN break do false next rescue then when

END case else for nil retry true while

alias class elsif if not return undef yield

## [end](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.src/M000018.html)

Marks the [end](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000018) of a [while](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000041), [until](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000039), [begin](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000009), [if](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000022), [def](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000013), [class](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000012), [or](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000028) other keyword-based, block-based construct.

## false

[false](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000020) denotes a special object, the sole instance of FalseClass. [false](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000020) [and](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000008) [nil](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000026) are the only objects that evaluate to Boolean falsehood [in](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000023) Ruby (informally, that cause an [if](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000022) condition to fail.)

## yield

Called from inside a method body, yields control to the code block ([if](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000022) any) supplied as part of the method call. If no code block has been supplied, calling [yield](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000042)raises an exception.

[yield](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000042) can take an argument; any values thus yielded are bound to the block‘s parameters. The value of a call to [yield](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000042) is the value of the executed code block.

# Printing to the console – puts, prints, p

puts "this is like console.log print "this is also like console.log p "this is a bit more complex

Brackets are mostly optional, occasionally necessary (only in method or function chaining)

puts("this is like console.log)

p

# Data types

* Strings
* Numbers
* Arrays
* Hashes (like objects)
* Methods (like functions)
* Symbols

## Symbols :name can convert – to string .***to\_s*** OR to symbol ***.to\_sym***

Symbols don’t take up as much memory. They are assigned a static place in memory (meaning that they don't need to be redefined). They behave in the exact same way as strings are easily translated back. Always use symbols for keys on objects.

:wolf.object\_id

1147228`

# Conversions!

:wolf.to\_s

# => "wolf"

"wolf".to\_sym

# => :wolf

# Numbers = 3 types – how do you use these???

THREE TYPES!

# Floats

1.0 2.1512

# Fixnums (Integers)

1241

125125129

# Bignums

1294810294801284012840812908

2512159412125699832859328

## Methods or Functions

def hello

# A plain method

end

hello # called like this

def hello( name )

# A plain method that takes a parameter

# When calling this, you MUST pass in a parameter or it will throw an error

end

hello "Wolf" # Called this way

hello("Wolf") # Or this

def hello( name = "World" )

# A function with a default parameter

# This won't throw an error in the case that you don't pass a parameter in

end

hello # Works this way

hello("Wolf") # Or this

Methods in Ruby have an implicit return, meaning that you don't need to actually use the return keyword, it does it automatically.

Parentheses are optional!

## Comparison Operators

All the same ones. Notes this new one

<=> (returns -1 if less than, 0 if equal, and 1 if greater than)

etc.

Different to JS

true && true ie true and true

true || false ie true or false

!true

## Variables ruby = “is nice”

ruby = "is nice"

Much harder to make global variables, it isn't the default behaviour in Ruby.

## Variable Interpolation in Strings ie putting variables into strings *#{ name }*

Interpolation just means you can put code inside

name = "gilberto"

drink = "scotch"

"My name is #{ name } and I drink #{ drink }!"

Interpolation only works with double quotes!! Single quotes means leave this string alone, this is mine

## Comments in Ruby = - #

# This is is a single line comment

## Getting User Input puts + variable = gets.chomp

In JS, we have prompt etc.

# Initial greeting

puts "What is your first name?"

# first\_name = gets

# This will wait for user input, and include the new line in the variable

first\_name = gets.chomp

# This will wait for user input, and strip the new line from the variable

# For more documentation on chomp - http://ruby-doc.org/core-2.2.0/String.html#method-i-chomp

puts "Your first name is #{ first\_name }."

# Controlling the flow

## If statements – if…end, if elsif else end…

if 13 > 10

p "Yep, it is a bigger number"

end

grade = "A"

if conditional

# To do

elsif conditional

# To do

else

# To do

end

p "Yep, it is a bigger number" if 13 > 10 # This only works in single line statements

# It's called a modifier (if modifier)

## Unless statements

x = 1

unless x > 2

puts "x is less than 2"

else

puts "x is greater than 2"

end

code\_to\_perform unless conditional

## Case statements

Think of these as shorter if statements, but don't overuse them (particularly in JS)

grade = 'B'

case grade

when 'A'

p 'Great Job'

when 'B'

p 'Good Job'

when 'C'

p 'Adequate Job'

else

p 'Talk to the Hand'

end

# Very similar to the switch statement in Javascript!

Now that we know this stuff, give [these exercises a go.](https://gist.github.com/ga-wolf/0d5d10bfa2b2871ae19727d2077711f9)

## While loops –Don’t Use these

## Until loops until ….end

until conditional

statement

end

i = 0

until i == 5

puts "I: #{ i }"

i += 1

end

# Iterators

## Blocks – content in iterators – ie between do and end or {}

The content in iterators are called blocks, they are quite similar to anonymous functions in javascript.

arr = [1, 2, 3]

# The content between the do and the end is the block

arr.each do |el|

puts el

end

# The content between the curly brackets is the block

arr.each { |el| puts el }

## Do and end delimit a code block

Paired with [end](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000018), can delimit a code block:

array.each do |element|

puts element \* 10

end

In this context, [do](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000015)/[end](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000018) is equivalent to curly braces, except that curly braces have higher precedence. In this example:

puts [1,2,3].map {|x| x \* 10 }

the code block binds to map; thus the output is:

10

20

30

In this version, however:

puts [1,2,3].map do |x| x \* 10 end

the code is interpreted as puts([1,2,3].map) [do](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000015) |x| x \* 10 [end](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000018). Since puts doesn‘t take a block, the block is ignored [and](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000008) the statement prints the value of the blockless [1,2,3].map (which returns an Enumerator).

[do](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000015) can also (optionally) appear at the [end](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000018) of a [for](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000021)/[in](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000023) statement. (See [for](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000021) [for](http://ruby-doc.org/docs/keywords/1.9/files/keywords_rb.html#M000021) an example.)

## Common way – 5.timse do …end or 5.downto(0)

So, so common in Ruby.

5.times do

puts "OMG"

end

5.times do |i|

puts "I: #{ i }"

end

# The thing that times wants to pass into me is stored as the parameter between the pipe characters

5.downto(0) do |i|

puts "I: #{ i }"

end

## ***For loops (No one ever uses these)***

## Generating Random Numbers

Random.rand # Generates a number between 0 and 1

Random.rand(10) # Generates a random number up to 10 (including zero and 10)

Random.rand(5..10) # Generates a number between 5 and 10 (also includes them)

Random.rand(5...10) # Does not include 5 and 10

Now that you know this stuff, have a crack at these [exercises.](https://gist.github.com/ga-wolf/6d598a3f18727fa692fbd54b8a76d55f)

# Arrays *[]*

## *Creation of an Array* - = [ , ] OR array.new OR array.new ( 3) or array.new (3, true) or array.new(4) {array.new(2)}

# LITERAL CONSTRUCTOR

bros = []

bros = [ 'groucho', 'harpo', 'chico' ]

# These work for reassignment as well!

arr[0] = 0

arr[0] = 1

# CLASS KEYWORD

bros = Array.new # => []

bros = Array.new( 3 ) # => [ nil, nil, nil ]

bros = Array.new( 3, true ) # => [ true, true, true ]

bros = Array.new(4) { Hash.new } # => [{}, {}, {}, {}]

bros = Array.new(2) { Array.new(2) } # => [ [nil, nil], [nil, nil] ]

# CHARACTER MODIFIERS - http://en.wikibooks.org/wiki/Ruby\_Programming/Syntax/Literals

|  |  |
| --- | --- |
| %w[ ] | Non-interpolated Array of words, separated by whitespace |
| %W[ ] | Interpolated Array of words, separated by whitespace |

%w{ Hello World }

%w[ Hello World ]

%w/ Hello World /

## Accessing Elements – arr[2] OR arr [-3] OR arr.at (0) OR arr.first OR arr.last OR arr.take(3) OR arr.drop (3) or arr.fetch(100, error)

arr = [1, 2, 3, 4, 5, 6]

arr[2] # => 3

arr[100] # => nil

arr[-1] # => 6

arr[-3] # => 4

arr[2, 3] # => [3, 4, 5]

arr[1..4] # => [2, 3, 4, 5]

arr[-1..-2] # => [5, 6]

arr.at(0) # => 1

arr.first # => 1

arr.last # => 6

arr.take(3) # => [1, 2, 3] - Grabs the first three elements

arr.drop(3) # => [4, 5, 6] - Grabs the last three elements

arr.fetch(100) # => IndexError: index 100 outside of array bounds: -6...6

arr.fetch(100, "ERROR") # => "ERROR"

## Adding Items to an Array [ , ] – .push or << or .unshift or .insert ( 3, “item” )

arr = [1, 2, 3, 4]

arr.push(5) # => [1, 2, 3, 4, 5]

arr << 6 # => [1, 2, 3, 4, 5, 6] Uses push behind the scenes

arr.unshift(9) # => [0, 1, 2, 3, 4, 5, 6] Adds an element to the start

arr.insert( 3, 'Serge' ) # => [ 0, 1, 2, 'Serge', 3, 4, 5, 6 ]

arr.insert( 4, 'didnt marry', 'Jane') # => [0, 1, 2, 'Serge', 'didnt marry', 'Jane', 3, 4, 5, 6]

## *Removing Items from an Array[ ]* – arr.pop OR arr.shift OR arr.delete\_at( 2 ) arr.compact??? OR arr.uniq (remove duplicates)

# Pop removes the last element and returns it (it is destructive)

arr = [1, 2, 3, 4, 5, 6]

arr.pop # => 6

arr # => [1, 2, 3, 4, 5]

# To retrieve and at the same time remove the first item

arr.shift # => 1

# Delete at a particular index

arr.delete\_at( 2 )

# To delete a particular element anywhere

arr = [1, 2, 2, 3]

arr.delete(2) # => [1, 3]

# Compact will remove nil values

arr = ['foo', 0, nil, 'bar', 7, 'baz', nil]

arr.compact #=> ['foo', 0, 'bar', 7, 'baz']

# Remove duplicates

arr = [2, 5, 6, 556, 6, 6, 8, 9, 0, 123, 556]

arr.uniq # => [2, 5, 6, 556, 8, 9, 0, 123]

## Changing elements in an array – arr[0] = 1

# These work for reassignment as well!

arr[0] = 0

arr[0] = 1

## *Iterating Over Arrays - - each do end OR* arr.each { |el| puts el } OR arr.reverse\_each OR arr.map

arr = [1, 2, 3, 4, 5]

arr.each do |el|

puts el

end

arr.each { |el| puts el }

arr.reverse\_each do |el|

puts el

end

arr.reverse\_each { |el| puts el }

# The map method will create a new array based on the original one, but with the values modified by the supplied block

arr = [1, 2, 3]

arr.map { |a| 2 \* a } # => Returns [ 2, 4, 6 ] but doesn't change the original

arr.map! { |a| 2 \* a } # => Changes the original and returns it

# DON'T DO IT THESE WAYS!

arr = [1,2,3,4,5,6]

for x in 0..(arr.length-1)

puts arr[x]

end

# or, with while:

x = 0

while x < arr.length

puts arr[x]

x += 1

end

for el in arr

puts el

end

## *Selecting Items from an Array – arr.select OR arr.reject OR arr.delete\_if or arr.keep\_if*

Elements can be selected from an array according to criteria defined in a block. The selection can happen in a destructive or a non-destructive manner. While the destructive operations will modify the array they were called on, the non-destructive methods usually return a new array with the selected elements, but leave the original array unchanged.

arr = [1, 2, 3, 4, 5, 6]

arr.select { |a| a > 3 } # => [4, 5, 6]

arr.reject { |a| a < 4 } # => [4, 5, 6]

# You can use these two with the exclamation mark to make them destructive

# The next two are destructive!

arr.delete\_if { |a| a < 4 } # => [4, 5, 6]

arr.keep\_if { |a| a < 4 } # => [1, 2, 3]

## Array comparisons (Venn diagram)– array1 | array2 OR array1 & array2 OR array1 - array2

array1 = ["x", "y", "z"]

array2 = ["w", "x", "y"]

array1 | array2

# Combine Arrays & Remove Duplicates(Union)

# => ["x", "y", "z", "w"]

array1 & array2

# Get Common Elements between Two Arrays(Intersection)

# => ["x", "y"]

array1 - array2

# Remove Any Elements from Array 1 that are contained in Array 2.(Difference)

# => ["z"]

## Destructive Methods vs Non-destructive Methods – destructive usually end with !

There are destructive methods and non-destructive methods in Ruby. Destructive methods will affect the original, whereas non-destructive will leave it alone and just return an altered copy. Destructive methods normally end with an !.

## Predicate Method – usually return booloean – usually end in ?

More or less, predicate methods are those that return a boolean value. They always end with a ?.

## Object IDs

In Ruby, every single thing is an object. Absolutely everything. Doesn't matter if it is a string, boolean or anything - they are all objects and all get assigned an object\_id. Everytime a new one is created, even if it looks identical, a new object\_id (a new place in memory) will be created.

"Wolf".object\_id

# => 70131971988560

{}.object\_id

70131953807740

false.object\_id

0

Each time you *reference* a new string, hash or array, it declares a new object\_id - meaning that it takes up more memory in your Ruby program. If you imagine a database with thousands of entries, these small things add up to huge amounts of memory.

## False Interlude

The only things that are considered false in Ruby are the boolean false, and the nil value.

# Hashes

## Creation of a Hash –*hash = { } OR hash = { :name => “serge, etc} or Hash.new*

# Literal Constructor

# "=>"" is called a hash rocket

hash = {}

serge = {

:name => "Serge",

:nationality => "French"

}

serge = {

"name" => "Serge",

"nationality" => "French"

}

serge = { # Keys stored as symbols!

name: "Serge",

nationality: "French"

}

# Class Constructor

hash = Hash.new

# Normally a hash will return nil if the property is undefined

# We can pass in default values to this quite easily though

hash = Hash.new( "WOLF" )

hash["Jack"] #=> Will return "WOLF"

# If you create the hash using the literal though...

hash = {}

hash.default = "WOLF"

hash["JACK"] #=> Will return "WOLF"

## Accessing Elements – *hash [:key] OR hash [“name”]*

serge = { # Keys stored as symbols!

name: "Serge",

nationality: "French"

}

serge[:name]

serge = {

"name" => "Serge",

"nationality" => "French"

}

serge["name"]

## Adding to a hash serge[:counterpart] = "Jane (temporarily)"

# Notice no hash rocket!

serge[:counterpart] = "Jane (temporarily)"

# This is the same way as you access them!

p serge[:counterpart] # => "Jane (temporarily)"

Removing from a hash hash.delelte(:key)

serge.delete(:counterpart)

## Iterating over hashes ***hash.each do || ….end – different variation depending on what want to return – all keys and values, some keys and values, specific keys and values***

serge = { # Keys stored as symbols!

name: "Serge",

nationality: "French"

}

# Will run for keys and values

serge.each do |all|

puts all

end

# Will run for each key and value pair

serge.each do |key, value|

puts "Key: #{key} and Value: #{value}"

end

# Return the current key

serge.keys.each do |key|

puts key

end

# Return the current value

serge.values.each do |value|

puts value

end

# Thousands of other ways to do this though

# Embedded Ruby

# Classes

## classes as blueprints

Everything in Ruby is an object. Every object inherits from a class. A class has a bundle of characteristics that have been defined to go with it, and we can create our own classes.

Everything in Ruby inherits from a class (is an instance of a class) and in some capacity inherit from Object. This doesn't pop up that regularly, but to see what I mean...

{}.class

[].class

"".class

# etc.

# If you want to see everything a data type inherits from...

{}.class.ancestors

[].class.ancestors

"".class.ancestors

Treat classes as a factory or a blueprint, something that gives another thing all the details that it needs. We use them to stop duplicate code, make manageable, easy-to-debug code.

## CREATING CLASSES AND adding methods

### Create

class Person

end

### Add methods

Comes from being able to add methods! We can encapsulate functionality with classes.

class Person

def speak

end

def laugh

end

end

### Create instance of class to be able to call the methods

In Ruby, we need to create an instance of the class. Think of the class itself being a blueprint, and the instance being the house that is built from it (it is the thing with all the power and functionality).

class Person

def speak

puts "Speak"

end

def laugh

end

end

To access the methods – create an instance of the class

person = Person.new

person.speak # Will work!

## Getters and setters – to get and set values on a class

Obviously, though. We want to be able to store something on the person themselves! Maybe we want to know their name, age or gender for example.

We use what are called "getters" and "setters" to do this.

## Shortcuts for getters and setters

There’s a long way to define getters and setters - but it is a hassle to define each one of these like this.

On a creation of a new instance in Ruby, when the .new method is called, it will also call automatically an initialize method.

## Seeing what we have defined

person.class.instance\_methods

# If you pass in the value false to the instance methods method, it will only show you the methods you have defined.

person.class.instance\_methods( false )

Building an application

# Routing

2 types of routing in code

get '/' do

"This is a get request to the root path - visited at localhost:4567/"

end

get '/anything' do

"This is a get request - visited at localhost:4567/anything" # This is returned as a response

end

Routes are matched in the order they are defined. The first route that matches the request is invoked.

These are literal matches. We actually have to visit /anything, this is annoying because we don't always know what we are going to receive.

The way we solve this is by using *named parameters*. Whatever is matched by the thing prefixed with the colon is stored in the params hash (which is automatically generated for us) and stored????

get '/hello/:name' do

# matches "/hello/foo" or "/hello/bar" or anything else that starts with "/hello/""

# params['name'] might be 'foo' or 'bar'

"Hello #{ params['name'] }!"

end

## 

As you might have guessed, params is an alias for the paramaters method. params comes from ActionController::Base, which is accessed by your application via ApplicationController. Specifically, params refers to the parameters being passed to the controller via a GET or POST request. In a GET request, params get passed to the controller from the URL in the user’s browser. For example, if our app’s controller looked like and the user typed in: then the controller would pass in {:name => “avi”} to the show method, which would set the @person instance variable to the person in the database with the name “avi”. In a POST request, params will get passed to the controller usually from a form. For example, say our app’s controller looked like and our form looked like If the user submitted the form with the name “avi” and the email address “avi@example.com” the controller would pass the hash :person => { :name => “Avi”, :email => “avi@example.com”} to the create method. Here, @person = Person.new(params[:person]) will become @person = Person.new(name: “Avi”, email: “avi@example.com”)

## Views / Templates – ERB – embedded ruby

1. Views folder
   1. layout.erb file. This will be used by default everytime Sinatra sees the erb command. This is where all of the stuff that you want to be on every page should go.

Need to include the following on the page at the end before the body

<%= yield %> this pulls in the content as routed (directed by the main.erb folder This grabs the contents of the file requested and puts it straight over itself.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Document</title>

</head>

<body>

<%= yield %>

</body>

1. Main.erb page

get '/post' do

erb :post

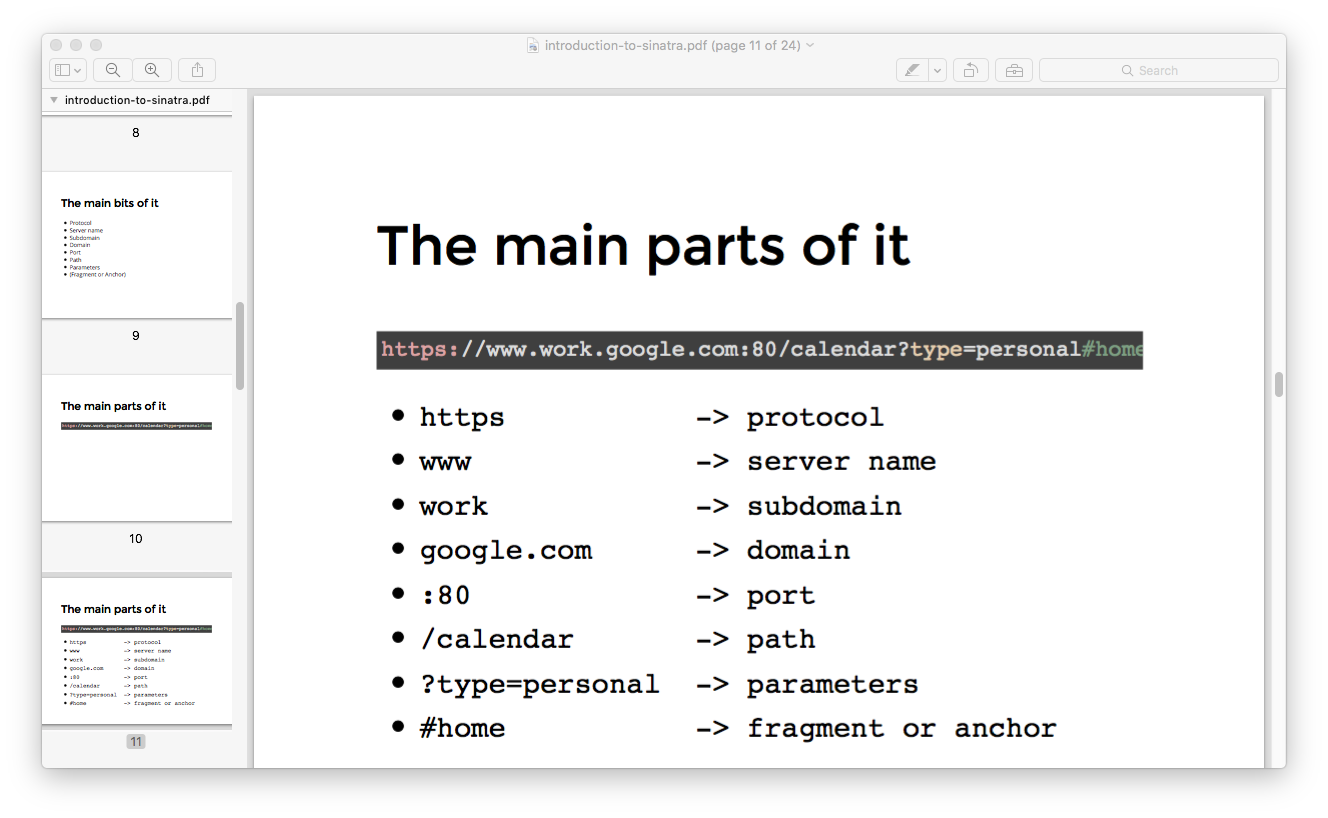
end

</html>

For more information about Sinatra, see [here.](http://www.sinatrarb.com/intro.html)

1. Public folder
   1. style.css

# Understanding the web



# Routing

We respond to requests (paths), that's how all web servers work

This is called routing. Routes are always matched in the order they are defined - first route it matches will be used

# SQL – using Sqlite3

## CRUD – **c**reate, **r**ead, **u**pdate, **d**elete

When we talk about tables and databases, there is really only 4 tasks that we need to do.

* **C**reate
* **R**ead
* **U**pdate
* **D**elete

This is called CRUD.

## Parts of a DB

Involved in every database, there are a couple of things. We have:

* The Database itself
* Individual tables
* Individual records on tables

## Steps to create a DB

1. Create a database file – NEVER look at this. If you delete it, it’s gone forever.
2. Create a .sql file – call it the name of the table you want.
3. In the .sql create the table with the columns that you need in the them:

CREATE TABLE person (

id INTEGER PRIMARY KEY, ### AUTOINCREMENT GOES SOMEWHER HERE

first\_name TEXT,

last\_name TEXT,

age INTEGER

);

1. Back in the console, need to run the database. Do this using

sqlite3 desired\_database\_name.db < add\_this\_table.sql

sqlite3 database.db < person.sql

This line will create the database.db file if necessary, and if not - it will just add whatever is defined in the .sql file specified. It imports the details from the .sql into the database.db.

1. Check it’s there. Do this in the CONSOLE
   1. type in sqlite3 database.db and hit enter in the terminal. This will open up a direct line to the database in the current folder
   2. type .schema - shows the current tables.

## Working with the DB from the Console

### Importing the database

1. Make sure you’ve run the database. Do this using

sqlite3 desired\_database\_name.db < add\_this\_table.sql

sqlite3 database.db < person.sql

This line will create the database.db file if necessary, and if not - it will just add whatever is defined in the .sql file specified. It imports the details from the .sql into the database.db.

1. Check it’s there. Do this in the CONSOLE
   1. type in sqlite3 database.db and hit enter in the terminal. This will open up a direct line to the database in the current folder
   2. type .schema - shows the current tables.

### Importing the sql file

can import that SQL into the database –

sqlite3 database\_name.db < insert\_stuff.sql

## ADD RECORDS – “Create.SQL” file + SQL command + import into DB – using SQLlite3 only

1. Create a create.sql file.
2. Copy current database schema into that file and comment out.
3. User SQL to add data to it.

INSERT INTO person (id, first\_name, last\_name, age) VALUES ( 0, "Zed", "Shaw", 37 );

-- We don't need to tell the attributes though, it can look just like this...

INSERT INTO person VALUES (0, "Zed", "Shaw", 37);

## Read records – READ.SQL file + Sql commands

1. Create read.sql file
2. Use the right commands in the SQL file

-- SELECT what FROM what\_table;

-- SELECT what FROM what\_table WHERE options;

SELECT \* FROM person; -- this will select all attributes and all records from the person database

SELECT name FROM person; -- only show the name attributes

SELECT \* FROM person WHERE first\_name == "Zed"; -- show all attributes from records in the person database where the first\_name is "Zed"

1. Add it to the database via the CONSOL

sqlite3 database\_name.db < insert\_stuff.sql

## UPDATE STEP – create update.sql file + SQL command + add to DB via the console

UPDATE table SET attribute\_name = attribute\_value WHERE attribute\_name = attribute\_value;

UPDATE person SET first\_name = "WOLF" WHERE first\_name = "Zed";

## DELETE STEP – create delete.sql file + SQL command + add to DB via the console

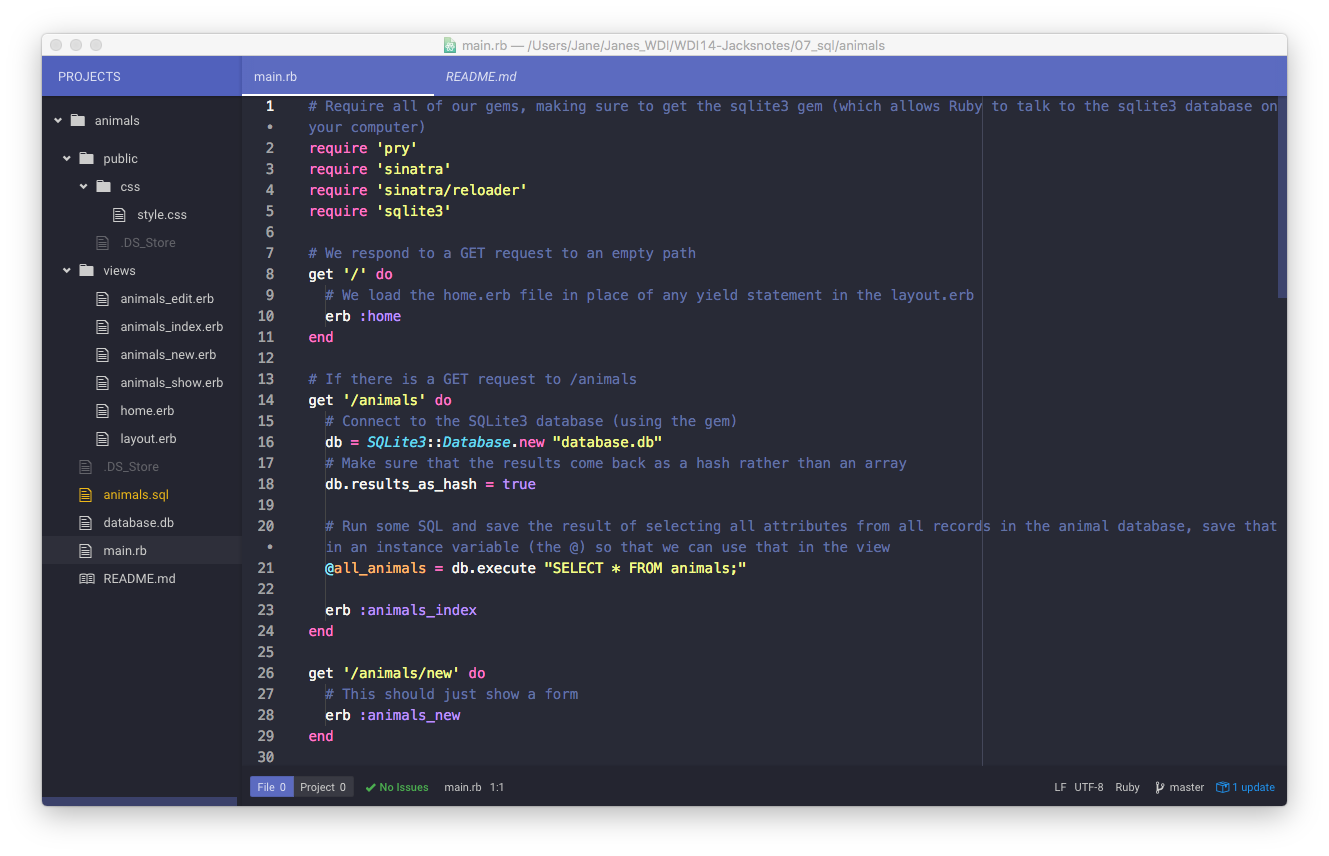
DELETE FROM what\_table WHERE what\_attributes = what\_value;

DELETE FROM person WHERE first\_name = "Zed"; -- Delete all records in the person table where the first\_name is equal to "Zed"

# Overall structure of CRUD application Using sinatra

In terms of actual structure of an application, this is the structure of a CRUD application. 7 views for all of this! The #new and the #edit are just ways to show the actual form.

### Files



## COMMANDS

| STEP |  |  |  | **File name** |  |
| --- | --- | --- | --- | --- | --- |
|  | VERBS | **URLS** | Line in main.erb | **SQL file name** | **SQL COMMAND** |
| CREATE | POST | /butterflies |  | #create | INSERT |
|  |  | /butterflies/new |  | #new |  |
| READ | GET | /butterflies |  | #index | SELECT |
|  | GET | /butterflies/:id |  | #show | SELECT |
| UPDATE | POST | /butterflies/:id |  | #update | UPDATE |
|  |  | /butterflies/:id/edit |  | #edit |  |
| DELETE | (Delete) | /butterflies/:id |  |  | DELETE |  |

CRUD is the foundation of most applications on the web, it is the thing that powers it! Important to get the principles of it.

# Object Oriented Programming (OOP)

Basically, OOP is an approach to development that tries to replicate real life. It is pretty much always done using objects or classes as namespaces and treats them as a way to make your code "modular".

# Active Record

## Preface - Object Relational Mapping

One of the most signifcant principles in Object-Oriented Programming is the idea of rich objects - things that store data, and allow it to be retrieved in logical and concise ways. This pattern is what Active Record strives for. They call it Object Relational Mapping, or ORM. It's basic principle is that rich objects in your application should be connected with database tables. Using ORM, the properties and relationships of the objects in an application can be easily stored and retrieved from a database without writing SQL statements directly and with less overall database access code. Also is far more secure due to lessened risks in regards to SQL Injections.

## What does Active Record do?

Active Record, as an ORM, gives us several mechanisms, the most important being the ability to:

* Represent models and their data.
* Represent associations between these models.
* Represent inheritance hierarchies through related models.
* Validate models before they get persisted to the database.
* Perform database operations in an object-oriented fashion.

## Convention over configuration – names to sort out DB associations

Convention over configuration is big in development anyway, but it is particularly full on when it comes to Active Record. That is because it uses the names to sort out assosciations etc. Make sure you follow these rules!!

* **Database Tables** - Plural with underscores seperating words (articles, line\_items etc.)
* **Model / Class Names** - Singular with the first letter of each word capitalized (Article, LineItem etc.)

## How do you work with Active Record

1. require it in your file

require 'active\_record' # make sure this is at the top of the file!

1. establish the connection to the database. These are annoying lines that will be done automatically in Rails! Just copy and paste them. They go in WHAT FILE?????

# Sets up our connection to the database.db we have created

ActiveRecord::Base.establish\_connection(

:adapter => 'sqlite3',

:database => 'database.db'

)

ActiveRecord::Base.logger = Logger.new(STDERR) # Logs out the Active Record generated SQL in the terminal. This isn't necessary but very helpful and cool to see what it is actually running

## CRUD with Active Record

Once we have our models defined (i.e. our classes), we can get into the CRUD stuff. There are always a lot of options of how to do this!

### Create

plant = Plant.new

plant.name = "Hibiscus"

plant.flowers = true

plant.save # YOU MUST RUN THIS LINE! (When using the new approach)

# OR

plant.create( :name => "Hibiscus", :flowers => true )

# OR

user = User.new do |u|

u.name = "David"

u.occupation = "Code Artist"

end # THIS WILL RUN THE SAVE AUTOMATICALLY

### Read

Plant.all

Plant.first

Plant.last

Plant.find( 10 ) # Find with an ID

Plant.find\_by( :name => "Hibiscus" ) # Returns the first plant that this works with

Plant.where( :name => "Hibiscus" ) # Returns all instances where this is appropriate

### Update

plant = Plant.find\_by( :name => "Hibiscus" )

plant.name = "Hibiscus 2"

plant.save

plant = Plant.find\_by( :name => 'Hibiscus 2' )

plant.update( :name => 'Hibiscus' ) # This will save automatically

### Delete

plant = Plant.find\_by( :name => 'Hibiscus' )

plant.destroy

Plant.destroy\_all

[Homework!](https://gist.github.com/ga-wolf/fa9e3278f71906cfcc9fd34d3bd59526)

#### Brief Interlude of Random Stuff

|  | **SQL** | **HTTP Verbs** | **Active Record** |
| --- | --- | --- | --- |
| Create | INSERT | (Put) or POST | .create or .new/.save |
| Read | SELECT | GET | .find or .find\_by |
| Update | UPDATE | (Patch) or POST | .update or .find/.save |
| Delete | DELETE | (Delete) or POST or GET | .destroy or .destroy\_all |