# LEARN RUBY IN 15 MINUTES

# TABLE OF CONTENTS

Step 1 - Installing Ruby........................................................ 1
Method Calls.......................................................................... 2
Variables................................................................................ 3
Truth and Falsehood............................................................ 4
Strings, Objects, Methods.................................................... 5
Method Chaining ................................................................. 7
Conditionals ......................................................................... 8
Defining a method................................................................ 9
Arrays.................................................................................... 12
Hashes................................................................................... 14
Iteration................................................................................. 16
Challenges............................................................................ 19
Epilogue................................................................................ 20

# STEP 1 - INSTALLING RUBY

You'll be happy to know that Ruby comes pre-installed on your Mac. Pretty neat eh?

Open the Terminal on your machine and type the following:

```
ruby -v
```

It will return something like 'ruby 1.9.3p392' or something similar. This means that you've got Ruby installed and you're ready to go. If you have a PC… Throw it in the bin and buy a Mac. You'll thank us later. (Seriously though, if you really want to keep your PC, you'll need to install Ruby yourself.)

Now go back to the Terminal and type

```
irb
```

to launch an interactive Ruby shell (so you can code along with this tutorial) and let's get cracking.

## METHOD CALLS

In Ruby, what most languages call a ‘function’ is actually referred to as a ‘method’. There are a couple of ways to call a method in Ruby.

```
puts(“Hello, world!”)
```

Here you’re calling the method puts, passing the String “Hello, world!” as the argument, inside parentheses.

Parentheses are often optional/implicit in Ruby, so you could just as well type:

```
puts “Hello, world!”
```

It’s also possible to pass multiple arguments to some methods. In these cases, each argument is separated by a comma:

```
puts(“Hello, world!”, “I am coding!”)
```

Although we often won’t need parentheses, in some cases they’re necessary. Rather than worrying about when to use them, let’s stick to using parentheses so we won’t be tripped up by these special cases.

## VARIABLES

An ordinary, regular variable is known as a local variable, and they’re created through assignment, using the ‘= ‘sign:

```
number = 17
```

Here the variable number is set to the value of 17. In Ruby, you should begin your local variables with a lower case letter. After that, they can include any alphanumeric character. Unlike Javascript variables, which are usually written like:

```
myLuckyNumber = 17
```

in Ruby, people tend to define their variables using what we call Snake Case, which uses lower case letters, and underscores in place of spaces:

```
my_lucky_number = 17
```

Makes it more readable right?

# TRUTH AND FALSEHOOD

In Ruby, everything is considered to be 'true' except false and nil.

That might seem a little strange, and at times it can be a little confusing, but it's important for you to remember

For example... 0 is true, and the String "Zero" is true as well. An empty String is also true, as is an empty array.

All numbers are true. All Strings are true. In fact, pretty much everything is true - well, everything except false and nil.

Everything in Ruby is true, except **false** and **nil.**

This will all make sense as we go through this tutorial, but for the time being, just remember that:

# STRINGS, OBJECTS, METHODS

Strings are simply a sequence of text, usually written inside quotation marks. It's possible to use either single or double quotation marks, so the following are both acceptable:

```
 my_first_name = 'Jordan'
my_middle_name = "Alexander"
```

It's possible to include any character in a String. So long as they're in between quotation marks, they'll be considered by Ruby to be a String.

```
my_age = "I am 28 years old… :) "
```

You may have heard it said that 'everything in Ruby is an object". If so, you won't be surprised by the fact that Strings are objects in Ruby. This means that Strings come with their own built-in methods which you can call, for example:

```
"jordan".upcase
```

Try it and see what it does…

Here, the object before the full stop is the String "jordan". The method we're calling on it is upcase. This is a method that will capitalise the object on which it is called.

You can get a list of all the methods of an object by typing object.methods into irb. Try typing this into your irb session:

```
"my name".methods
```

You'll see a log list of methods you can use... Have a play with some of them if you're so inclined. The instructions on how to use these methods, and what they will return, can be found in the Ruby Docs at this link:

http://ruby-doc.org/core-2.0/String.html

Just like with functions in other languages, methods can have arguments:

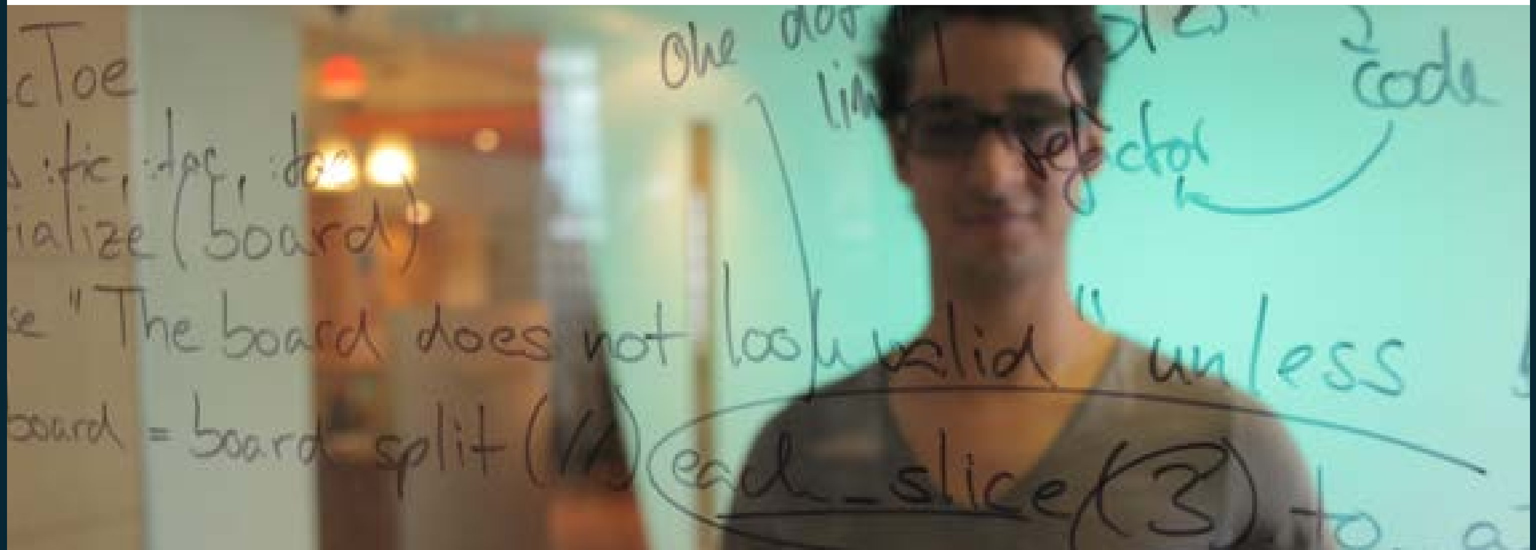
```
"jordan".include?('jor')
```

Will return true if the String object "jordan" includes the String 'jor' which we have passed as an argument to the include? method.

Or put another way, the include? method is true if its argument ('jor') is a subString of the obejct on which it is called (the String "jordan".)

**CHALLENGE**

Is the include? method case senstive? Try it in irb and find out.


cToe
ialize(board)
The board does not look valid unless
board = board.split(//).each_slice(3)
code
refactor

# METHOD CHAINING

Method chaining is a handy technique that allows you to sequentially invoke multiple methods in one expression. It helps to improve the readability of the code while reducing the amount of code you actually need to write. We chain methods using the 'dot notation'. In method chaining, the interpreter will pass along the chain of commands from left to right, passing the output from one method to the input of the next, eliminating the need for intermediate variables.

For example, these two expressions will have identical outputs:

Using intermediate variables:

```
name = "jordan"
upper_case_jordan = name.upcase
jordan_backward_in_caps = upper_case_jordan.reverse
```

Using method chaining:

```
jordan_backward_in_caps = "jordan".upcase.reverse
```

In both cases, the String "jordan" is first upcased using the upcase method and then the letters are reversed using the reverse method. The upcased & reversed String is then set to the variable jordan_backwards_in_caps.

In the first instance we set the String "jordan" to a variable, then set the upcased version of the String to a new variable upper_case_jordan, and then set the reversed & upcased version to a new variable jordan_backwards_in_caps

In the second version, we chain the upcase and reverse methods to the string. In one line we set the variable jordan_backwards_in_caps to the string "Jordan", first upcased and then reversed.
Here is a slightly more complicated version... Try running it in irb and see what the output will be:

# CONDITIONALS

Conditionals are super simple in Ruby. They're structured as follows:

```
if some_condition
     puts "this code will be executed if some_condition is true"
else
     puts "this code will be executed if some_condition is false"
end
```

You'll need to put the if, else and end keywords on separate lines as shown above. Technically you don't have to indent the code, but you definitely should. It's the expected convention (and Ruby loves convention!). Not to mention the fact that it makes the code much easier to read later.

But what about if you want more than one condition? You can use the keyword elsif, and you can use it as many times as you want:

```
if some_condition
     puts "this code will be executed if some_condition is true"
elsif another_condition
     puts "this code will be executed if another_condition is true"
elsif a_third_condition
     puts "this code will be executed if a_third_condition is true"
else
     puts "this code will be executed if none of the above are true"
end
```

As always, have a play in irb and get a feel for how it works. Forgotten how to load irb? It's simple. Go to your terminal and enter the command:

```
irb
```

# DEFINING A METHOD

The anatomy of a method definition is as follows:

```
def method_name(argument1, argument2)
     #your method's code goes here
end
```

As you can see, you start a method definition with the keyword def and close it with the keyword end. After the def goes the name of the method. This can be anything you like, except one of the Ruby reserved words.

You should try to give your methods clear, descriptive names. Ideally the method should read like a regular English sentence and, if pushed, someone should be able to pretty much guess what the method should do just by the name of the method and the name(s) of the argument(s) that can be passed to it.

If your method takes any arguments, they go in parentheses after the method_name, and are separated with commas.
Some methods take no arguments:

```
def say_happy_birthday
     puts "Happy Birthday!!"
end
```

While others take one argument:

```
def say_happy_birthday_to(name)
     puts "Happy Birthday #{name}"
end
```

While others can take two or more arguments:

```
    def say_happy_birthday_to(name, repetitions)
        repetitions.times { |i| puts “Happy Birthday #{name}”}
    end
```

How do you think you would call this method to say happy birthday to someone 3 times?

```
    say_happy_birthday_to(“Jordan”, 3)
```

In Ruby, by default, methods will return the last executed statement in the definition. So:

```
    def yearly_salary(monthly_salary)
        monthly_salary * 12
    end
```

Will both return the value of monthly_salary multiplied by 12.

You can assign the output of a method to a variable too, so:

```
    jordans_wishful_yearly_salary = yearly_salary(100,000)
```

Will set the variable jordans_wishful_yearly_salary to the integer 1,200,00, since we defined the method yearly_salary(100,000) to return 100,000 * 12, which is 1,200,000.

Before you look at the solution on the next page, why don't you try to write a method that will check whether the number given to it as argument is positive or negative, and display a message with the answer? Try to write it in irb now.

**SOLUTION**

```
    def positive_or_negative(number)
        if number > 0
            puts “Number is positive”
        else
            puts “Number is negative”
        end
    end
```

You can test if your method works once you've defined it by entering:

```
    positive_or_negative(-7)
```

Which should now display Number is negative.

**CHALLENGE**

Could you try to extend the make_positive method to accommodate the case of 0, which is neither positive nor negative? Tweet @MakersAcademy if you want help, or send us your solution when you've worked it out!

# ARRAYS

You can think of an Array as a list - a list that's defined using the square brackets [ ], and a list which can contain any number of objects, of any type.

You can have an empty Array:

```
[]
```

Or an Array with some numbers in it:

```
[1, 2, 3, 4, 5]
```

You can even have an Array with multiple data types. This one has an integer, 2 Strings and a floating point number:

```
[ 1, ‘two’, 3.0, ‘four and five’]
```

You can even have what's called a multi-dimensional Array... Arrays within Arrays!

```
[ 1, [2, 3], [4,5] ]
```

You can access each item in an Array using it's numerical position. Type the following into irb:

```
my_Array = [1, ‘two’, 3]
my_Array[1]
```

This will return the item in the Array at position 1.

Did you try it? Were you a little shocked to find that it returned ‘two’??

This is because Arrays in Ruby start counting at 0. The first item in the Array is accessed by executing my_Array[0], the second item with my_Array[1] and so on.

You can change an element simply by reassigning it:

```
my_Array[1] = 2
```

If you call my_Array now, you'll see that its contents are [1, 2, 3]

Just like Strings, Arrays come with built in methods, such as length:

```
my_Array.length
```

If you're coding along with me, this will return 3.

You can also add new elements to the end of an Array:

```
my_Array.push(‘four’)
```

What will my_Array be now? And what will its length be?

**SOLUTION**

my_Array, after the changes executed above, should now be equal to:

[1, 2, 3, ‘four’]

# HASHES

A good way to think of a hash is as an Array where you access items not by their numerical position but by a unique key.

In an Array, the ‘key’ for accessing a certain object is the number which describes that objects position in that Array.

In a hash, you can decide which ‘key’ is used to access a given value. For example, let’s say you had an Array of capital cities:

```
capital_cities = [“London”, “Madrid”, “Tokyo”]
```

You would be able to get the capital city of England by entering:

```
capital_cities[0]
```

But this isn’t very expressive, is it? Nor is it very clear what we expect capital_cities[0] to return.

Rather than the integer 0 being the key that returns the capital city of England, wouldn’t it be great if ‘England’ was the key, and ‘London’ the return value?

That’s what a hash is for.
An example hash can be created as follows:

```
capital_cities = { “England” => “London”, “Spain” => “Madrid”,
“Japan” => “Tokyo” }
```

This means that we can access the value “London” with the key “England” as follows:

```
capital_cities[“England”]
```

See how it similar it looks to an Array, just with the integer replaced with a more expressive, unique key?

We can then add a new capital city with the expression:

```
capital_cities[“Hell”] = “Pandaemonium”
```

Simples :)

# ITERATION

When you write a script that repeats itself somehow, it is called iteration. There are many ways to do iteration. Here are a couple of examples:

```
4.times do
    puts “Happy Birthday to you”
end
```

This will print “Happy Birthday to you” to the screen 4 times.

The part between the do and end is called a block. If you want, you can replace the do and end with curly braces - they are synonymous:

```
4.times { puts “Happy Birthday to you” }
```

The convention in Ruby is that if the contents of the block fit on one line, use curly braces. If the contents of the block require more than one line, use do and end.

But this is a little repetitive, isn’t it? You can do more interesting iterations by creating a local variable inside the block:

```
[1, 2, 3].each { |x| puts “I am on iteration #{x}” }
```

(note that the local variable x is defined inside pipes, which you’ll find by pressing shift + backslash on your Mac)

Let’s break this down.

On the first iteration, Ruby will take the first item in the Array, the integer 1, and assign it to the local variable x. It will then print out ‘I am on iteration x’, which will literally display “I am on iteration 1”, since the local variable x has been assigned to the first value in the Array on this iteration, which is 1.

On the second iteration, Ruby will take the second item in the Array, the integer 2, and assign it to the local variable x. It will then print out ‘I am on iteration x’, which will literally display “I am on iteration 2”.

On the third iteration, Ruby will take the third item in the Array, the integer 3, and assign it to the local variable x. It will then print out ‘I am on iteration x’, which will literally display “I am on iteration 3”.

Make sense?

Try and understand in your head what would happen if you ran the following:

```
[“went to market”, “stayed home”, “had roast beef”].each do |x|
    puts “this little piggy #{x}”
end
```

And now enter it into irb and see if you were right... Were you?
What about this one?

```
default = “clap your hands”
special = “and you really want to show it”
[default, default, special, default]each do |x|
    puts “if you’re happy and you know it #{x}”
end
```

Making sense yet?

Have a play in irb and see if you can get the hang of it. Why not create an Array of numbers and see if you can iterate over them, displaying, on each occasion, the number multiplied by 2?

You should know that you can iterate over a hash as well as an Array. Let's go back to our old example:

```
    capital_cities = { “England” => “London”, “Spain” => “Madrid”,
“Japan” => “Tokyo” }
```

You could print these out in a nice format by doing:

```
    capital_cities.each { |country, city| puts “#{country}’s capital is
#{city}”}
```

Enter all this into irb... Do you see how it works?
On the first iteration, the key “England” is assigned to the local variable “country”, and the value “London” is assigned to the local variable “city”. (There's nothing clever going on here - the computer doesn't recognise them as countries and cities. If the local variables were x and y, it would assign “England” to x and “London” to y - it's simply going through the key/value pairs in order and assigning them to the local variables that you specify, whatever they may be.

Ruby will then print the statement “#{country}'s capital is #{city}”, with country replaced by “England” and city replaced by “London”.

On the second iteration, the key “Spain” is assigned to the first local variable, which is “country”, and the value “Madrid” is assigned to the second local variable which, for the purposes of readability, we have defined as “city”. Ruby will then print the statement “#{country}'s capital is #{city}”, with country replaced by “Spain” and city replaced by “Madrid”. And so on...

Have a play with this in irb. Don't be afraid to try things out. Have fun. You now know some of the basic syntax of how to write Ruby code in irb, so see how far it can take you.

# CHALLENGES

Here are a few challenges you can try to solve (use Google and the Ruby Docs (http://ruby-doc.org/core-2.1.1/) if you don't know certain things... Or you could tweet @ MakersAcademy!)

**CHALLENGES**

**- Write a method that will tell you if a number is odd or even.**

**- Write a method that takes on argument and returns the square of that number.**

**- Write a method called ‘shout’ that takes a String as an input and returns that String in capital letters.**

**- Write a ‘greeter’ method that takes a name as an input such that I could write “greeter(“Jordan”) and it would display “Hello Jordan! How are you today?”**

**- Iterate over an Array of numbers to display the square of each number in the Array**

**- Iterate over an Array of numbers and only display even numbers**

**- Create a hash containing your 5 best friends, with each persons name as the key and their age as the value. Iterate over that Array to display 5 examples that look like: “Jordan is 28 years old”**

# EPILOGUE

That’s it!

If you’ve enjoyed yourself, and you’re interested in getting serious about learning Web Development, you might want to check out an intensive bootcamp like Makers Academy. We run a 12 week, full time, super intensive web developer bootcamp based in the heart of London’s tech community, and new courses start every 6 weeks.

The course is designed for everyone - from complete novices, to computer science grads wanting practical experience, to founders sick of looking for their technical co-founder and everything in between.

You’ll learn Ruby on Rails, HTML5 & CSS3, Agile & Lean Development, Javascript and jQuery, Git & Heroku, and software design. You’ll learn through firsthand experience, community-driven classrooms, pairing, and project-based work. Makers Academy’s goal is to have a 100% placement rate among graduates seeking employment, and we have a successful track record placing students in high-earning junior developer roles, averaging over £30k / annum.

For more info, check out Makers Academy, and if you liked this mini-tutorial, please tweet us your thoughts and feel free to share this with anyone you know who’s interested in learning to code.

http://www.makersacademy.com

MAKERS ACADEMY

# NOTES

MAKERS ACADEMY