Table Of Contents

Introduction

Welcome! You just took a great leap forward by deciding to improve your skills.

My name is **Jesus Castello** and I truly hope you find this book useful. If something is not clear please send me an email (jesus@rubyguides.com) and I'll be happy to help you!

In this book, I'll take you on a journey on some of the why's and how's of Ruby, as well as showing you some 'nice to know' tips. My hope is that you'll become a lot more confident and skilled and that will help you achieve your dreams.

Improving your programming skills is not only about theory but also **a lot of practice**. I recommend that you practice every day doing some coding exercises.

Consistency is important.

I would also recommend that you keep a folder with all your small projects and code examples around.

They may be useful for reference later, or you might learn something new that you can do to improve them.

> **Note**: If you don't know what to work on, you could try finding an open source projects on GitHub or using a site like codewars.com or exercism.io

When working through the lessons in this book, you should play around with all the examples provided and try to really understand the presented concepts.

Practice.

One trick you can use to see if you really understand some code is to make a prediction of what will happen if you make a change, then make the change and see if what you predicted was true.

The key is to **play with the code** instead of just looking at it so you can gain a deeper understanding.

Programming is not easy, but never let that discourage you. If you commit to continually improve your skills, it will pay off.

You can do this!

Let me give you another tip that will make you look like a genius. I used this many times for **great results**.

Here it goes:

Learn to use & love the documentation for whatever you're working with.

One time I helped someone with an ActionScript issue, and I never saw that programming language before. All I did to help this person was to search the documentation and pay attention to the details.

> **Note**: Ruby documentation can be found at http://www.rubydoc.info/stdlib/core/index, http://ruby-doc.org/ or https://devdocs.io/ruby/. Bookmark these sites!

If the documentation is not enough, remember that the source code is the real deal. Look in there to see what's going on, search for strings, classes and methods, and eventually, you will find what you are looking for.

Now, let's get started!

Using Pry

Contents

Getting Started

Did you know that you can learn a lot about Ruby without leaving the command-line? All you need is `pry`.

Pry is irb with super powers.

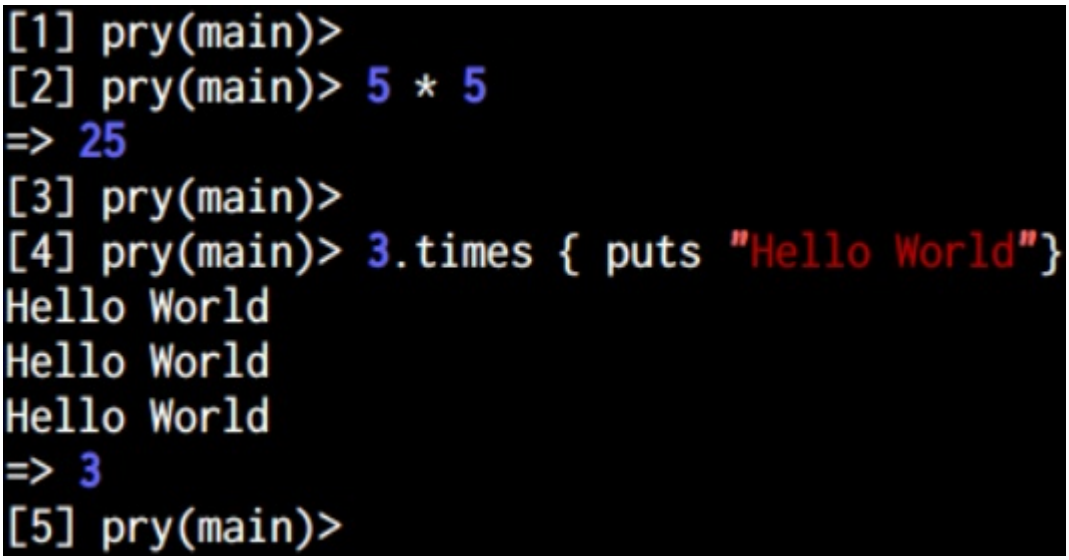
It has a number of features that let you investigate your objects & classes and learn more about them without having to take a look at the documentation. In addition, pry has syntax highlighting and auto-completion for your code.

Let's get started by installing `pry` on your system:

```
gem install pry
gem install pry-doc
gem install pry-byebug
```

Now you can start the program by just typing `pry`. You will get a prompt similar to `irb`.

You can start typing some simple Ruby snippets just to get used to it. You can also type `help` to get an overview of the available commands.



Tip: you can use the _ variable to get access to the last result. Try typing 5 * 5 and then _ * 3 to see this in action.

The Power of Pry

The most powerful pry command is `ls`.

This command will list all the constants, methods and instance variables for an object.

> **Remember**: A constant is a type of Ruby variable that starts with a capital letter, like Foo = 5, or DAYS_IN_A_WEEK = 7.

Let's see what happens when you run the `ls` command with the `String` class:

```
[3] pry(main)> ls String
Object.methods: yaml_tag
String.methods: try_convert
String#methods:
  %             bytes       chr         each_char
  *             bytesize    clear       each_codepoint
  +             byteslice   codepoints  each_line
  <<            capitalize  concat      empty?
  <=>           capitalize! count       encode
```

You can see a list of methods titled `String#methods`.

The hash (`#`) symbol means that these are **instance methods**. These methods are only available to instances of the string class.

Remember: An "instance of a class" is just a fancy name for "object". String objects are the strings themselves, like "bacon".

You'll probably find a few methods that you haven't seen before, how about taking them for a spin?

Exercise:

Pick 3 String methods you haven't seen before from the list given to you by pry. Then take a quick look at the documentation by using show-doc String#your-method-here.

For example: show-doc String#count.

Don't worry if the description for a certain method looks confusing, focus on the examples and try to understand the output for each example. You can also copy and paste the examples directly into pry to try them out.

Then ask yourself the following question:

"Can I use this method to improve some code I have written before?"

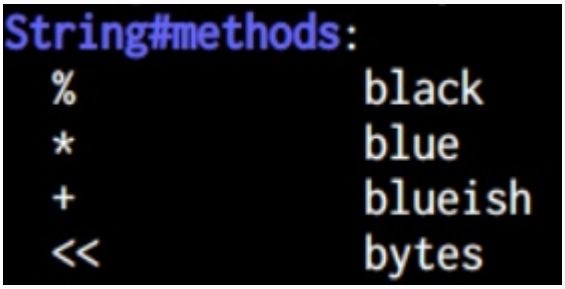
If the answer is yes then go ahead and find that code to refactor it. If the answer is no, don't worry, think about some potential uses for this new method you just discovered!

If you know a lot of built-in methods you'll be able to write better Ruby code.

Where Did This Method Come From?

You can do more things using pry.

For example, in my computer I can see that the `String` class has some extra methods that aren't normally part of this class.

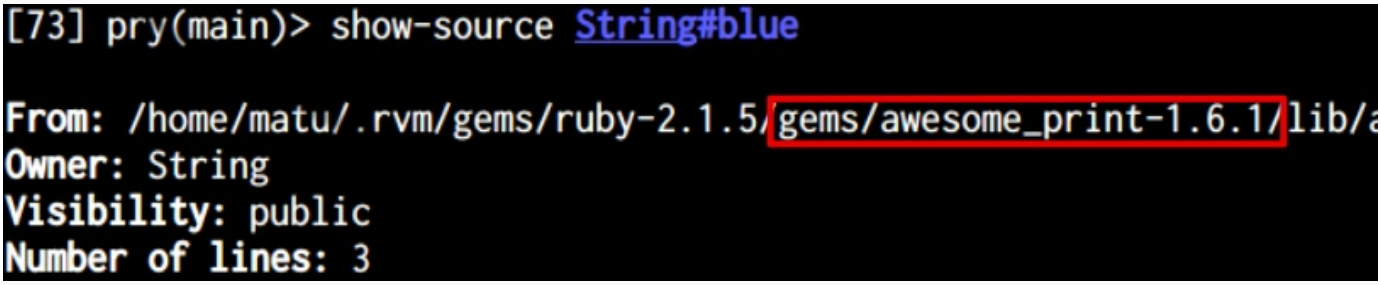


Do you see these color methods in the image? blue, black, blueish...

I wonder how they got there!

You can use the `show-source` method to uncover the truth. In my case, I found out that these are coming from the `awesome-print` gem.

Take a look at the following image:



Mystery solved!

Pry Tricks & Tips

Here are a few useful things you should know while using pry.

For example, the prompt - the bit of text at the start of every line - contains some useful information.

```
[1] pry(main)>
```

The number `1` indicates the line number inside `pry`.

It can be useful to reference later if you want to edit & reuse code you typed on that line.

To do that you use pry's edit command:

```
[1] pry(main)> def testing
[1] pry(main)*   puts 123
[1] pry(main)*  end
[2] pry(main)>
[2] pry(main)> edit -i 1
```

This will open the `testing` method inside your default editor (defined by `$EDITOR` environment variable) & allow you to edit it.

Then pry will run the code you edited inside the pry session.

You may have noticed another change.

The `>` symbol at the end of the prompt changed to `*`.

This is to indicate that you have an unfinished expression, like a method definition, class definition, etc.

If you want to go back to the regular input mode `>` without finishing the expression you can press `CTRL+C` once.

More useful pry commands:

- history
- find-method
- watch
- backtrace
- cat
- ri
- @

Use `help` plus the command name (like `help history`) inside pry to find out more details about these :)

Debugging Using Pry

Pry is useful for learning, playing around & even writing actual code...

...but it's is also useful for debugging.

To use pry as a debugger you need to have the `pry-byebug` gem installed.

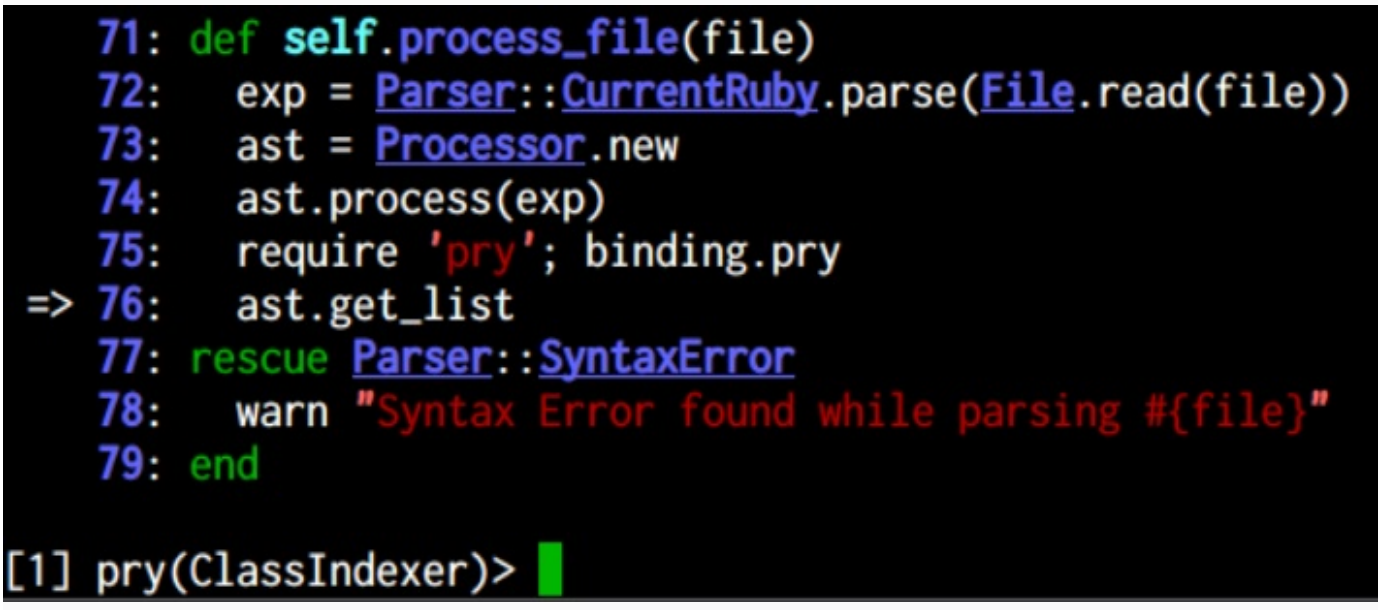
You also need to require pry into your project. You can do that via a direct `require 'pry'` or in your Gemfile.

Now find your problematic piece of code and drop this line in there:

```
binding.pry
```

Run your code.

Once you hit the code branch that has your break-point (`binding.pry`), you should get a pry session. You will see some lines of code around your break-point for context, with an arrow that indicates the current line.



At this point your code is 'frozen' and you can poke around to see what's going on.

For example, you may want to **query your local variables to see their values**. You can also run methods that your code has access to.

To get the value for a variable you just type its name & press enter.

Make sure that every variable has a correct value.

No assumptions. Check everything!

And don't forget:

Use `ls <object>` if you don't know or remember **what methods are available for a particular object**. The `ls` command can also be used without an object to see the current instance & local variables names.

If you want to advance the execution of the program you can type the `next` command. If you want to exit the pry session completely you can type `exit!`.

When you are done remove your `binding.pry` from the code so the program can work without interruption.

Understanding Exceptions

Contents

What are Exceptions and Why They Happen

Exceptions are a mechanism to handle unexpected failures inside a program.

They typically show themselves in the form of a program crash with an error message. Exceptions are used to **signal an error condition** when the program is not able to finish its work in the current state.

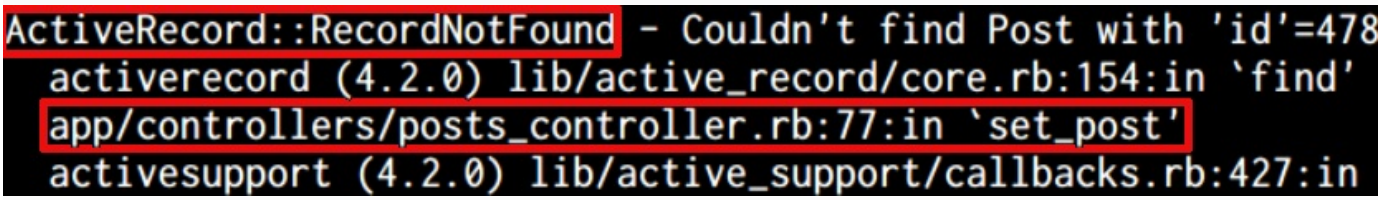
When a program raises an exception it's saying this:

"Hey, I just got this error condition (file not found, invalid data, etc.) & I don't know what to do with it. I'm letting you know so you can handle it."

In Ruby, exceptions are implemented as classes, all of them inherit from the `Exception` class.

The documentation has a list of all the built-in exceptions that you should check out now.

You have probably seen exceptions a few times before, they look like this:



This error message is composed of :

- The exception class (ActiveRecord::RecordNotFound)
- The exception message ("Couldn't find Post with id=478")
- The stack trace

All of those are vital pieces of information to help you understand and fix the problem.

Understanding Stack Traces

Error messages & exceptions are there to help you, just like your body tells you that there is something wrong when you feel pain.

The information provided by **error messages can help you find where the problem is**, so pay attention to them.

If you run the following code (from a file, instead of pry/irb):

```
def buy_milk
  "got milk"
end

def buy_bacon
  raise "No bacon left!"
end

def go_shopping
  buy_milk
  buy_bacon
end

go_shopping
```

You will get an exception thrown at you.

Let's analyze it together so that we can **understand what's happening**.

```
shopping.rb:6:in 'buy_bacon': No bacon left! (RuntimeError)
    from shopping.rb:11:in 'go_shopping'
    from shopping.rb:14:in '<main>'
```

When you get an error like this the first thing you want to do is to look at the first line. It tells you where the error happened, in this case `shopping.rb` on line `6`.

> **Ruby 2.5 update**: The stack trace is in reverse order since Ruby 2.5. This means that the most important line will be the last one instead of the first one.

The first line also tells you the name of the method where things went wrong, in this case `buy_bacon`.

On the next line you will see the actual error message and the exception class.

In this example we have a `RuntimeError` exception.

The following lines tell you the path of execution that the program took to get there. It can be useful when the actual error condition originated somewhere along the path.

In this example you can see how `buy_bacon` was called by `go_shopping` on line `11`, and `go_shopping` was called from `main` on line `14`.

> **Note**: main is not an actual method, it's used to represent the top-level context of your application

When working with some framework like Rails the stack traces tend to be a lot longer...

...in these cases you want to focus on finding lines with file names that you recognize (see the image at the start of this lesson for an example).

Using Exceptions

An exception doesn't always mean that your program has to end.

You can 'trap' an exception and just log an error message or do something to correct the problem.

Never trap an exception just to ignore it!

Example:

```
File.open("random_file_001.txt")
```

Assuming that `random_file_001.txt` doesn't exist, you will get the following error.

> **Note**: This is all one line. I broke it down for readability.

```
open-file.rb:1:in `initialize':
No such file or directory @ rb_sysopen - random_file_001.txt
(Errno::ENOENT)
```

The exception in this case is `Errno::ENOENT`.

The `Errno` family of exceptions are all related to the operating system.

In the context of a larger program you may not want this code to crash just because this specific file didn't exist (or some other condition).

What you can do is to `rescue` the exception.

You can do that by wrapping the code that raises the exception within a `begin / rescue` block.

Here is an example:

```
begin
  File.open("random_file_001.txt")
rescue Errno::ENOENT
  puts "File not found :("
end
```

This code will print the `File not found :(` message and end normally.

The key here is in the `rescue` method.

This method tells Ruby to trap this exception (`Errno::ENOENT`).

If you use `rescue` without an exception name then `StandardError` is the default trapped exception. Note that this will also trap all the exceptions that inherit from `StandardError`.

Having a specific exception prevents you from trapping other exceptions that you didn't plan for. **This is important because it could hide a problem with your code** if you just try to `rescue` everything.

> **Remember**: To avoid hiding errors with your code, you need to use rescue with an argument. The argument for rescue is an exception class, the class you need to use will depend on what you are doing inside your begin block.

When capturing an exception you can also get the exception object, which contains the error message and the backtrace (also known as stack trace). This can be useful for logging purposes.

To get access to the exception object you can modify the last example like this:

```
begin
  File.open("random_file_001.txt")
rescue Errno::ENOENT => e
  puts "File not found :("
  puts e.message
  puts e.backtrace
end
```

You don't always need a `begin`, in fact Avdi Grimm says that they are often a code smell.

> **Definition (code smell)**: A code smell is like a red flag, it tells you that this code is probably not the best it can be.

Example: rescue without begin

```ruby
def open_file
  File.open("random_file_001.txt")
rescue Errno::ENOENT => e
  puts "File not found :("
  puts e.message
  puts e.backtrace
end
```

Another keyword related to exceptions that you should know is `ensure`.

The code inside `ensure` will always be run, even if an exception happens.

This can be used to clean up resources or log some kind of message.

Example:

```ruby
begin
  puts "test"
ensure
  puts "Always print this!"
end
```

In this example you can see how using `rescue` is not necessary for using `ensure`.

Here is another (simplified) example of `ensure` being used in a real project:

Rack::Lock.

```
def call
  @mutex.lock

  begin
    puts "Do something..."
  ensure
    @mutex.unlock
  end
end
```

There is one more keyword: `retry`.

When you use retry inside a `rescue` block, it will re-run the `begin` block. I would recommend that you avoid using `retry` because you can easily get into infinite loops and other issues.

Exceptions and Return Values

What happens with return values when a `rescue` is involved?

As you may know, the last thing evaluated in a method is what gets returned from that method.

This is called 'implicit return'.

The `rescue` part of your code doesn't get evaluated unless there is an exception, so you can continue to use implicit returns normally.

Here is an example:

```
def testing(exception = false)
  raise if exception
  1
rescue
  "abcd"
end

p testing
# Output: 1

p testing(true)
# Output: "abcd"
```

At Exit

When a Ruby program ends for any reason the `at_exit` block is called.

This could be useful if you want to write some sort of crash logger.

The last untrapped exception is available using the global variable `$!`.

```
at_exit do
  puts "Program finished at #{Time.now}"
end
```

If you need to end the program immediately without going through any kind of callbacks you can use the `exit!` method.

Custom Exceptions

Ruby has a number of built-in exception classes, but you can also create your own custom ones.

Example:

```
class InvalidAnswer < StandardError
end

raise InvalidAnswer, "my error message"
```

These custom exceptions can make your error reports a bit easier to manage, and if you are making some sort of gem or library it will be possible to `rescue` this specific exception instead of a more generic one.

Common Errors

There are a few exceptions that are very common, you should be aware of them and their solutions.

The no method error

The 'no method error' is probably the most common of them all.

It usually happens because you made a typo on the method name or your method is out of scope.

There is also another variation of this error where the receiver of the method call is nil, in which case the problem has to do with how that nil got in there.

Let's see an example of each:

```
class SandwichMaker
  def make_me_a_sandwich
  end
end

make = SandwichMaker.new
make.make_me_a_sandich

# NoMethodError: undefined method `make_me_a_sandich' for #<Sandwic
```

The solution for the first case is to double check that you didn't make a typo & that the method you want to use actually exists (documentation can help here).

```
num = nil
num.say_hello

# NoMethodError: undefined method `say_hello' for nil:NilClass
```

The second case (with the nil) can be a bit more complicated. You will have to trace where that nil came from (most of the time it won't be as obvious as in the example).

You can use the stack trace and a debugger to help you.

The argument error

This error happens when you call a method with the wrong number of arguments. For example, in the following code we define a method that takes two arguments, but when we call it we only pass in one of them.

```
def log_message(msg, destination)
  destination.write(msg + "\n")
end

log_message("hello there")
```

This gives us the following error:

```
ArgumentError: wrong number of arguments (1 for 2)
```

The part that says `(1 for 2)` tells us exactly how many arguments we passed in (1) and how many the method expects (2).

The solution is to change our method call to pass in the missing arguments. Alternatively, we can set a default for this argument in the method definition, like this:

```
def log_message(msg, destination = $stdout)
  destination.write(msg + "\n")
end
```

The type error

This error will appear when we try to call a method on incompatible types, for example string + integer.

Here is an example:

```
"This is a string" + 1
# TypeError: can't convert Fixnum into String
```

Ruby doesn't want to do any guessing on what we really wanted to do with this code, so it just throws its hands up until we explain how to proceed.

One solution is to call the `to_s` method. If you are using string interpolation this happens automatically.

```
number = 1
"This is a string #{number}"
```

The uninitialized constant

The 'uninitialized constant' error is a bit less common than the first three, but still very useful to understand. This error will show up whenever we try to use a constant that doesn't exist.

As a reminder, constants in Ruby always start with a capital letter, that's what makes them a constant.

Constants are used as class and module names, but they can also be used to give a name to a common value (for example `PI`).

Let's see an example:

```
animal = Cat.new

# NameError: uninitialized constant Cat
```

The solution is to make sure we don't have a typo and define that constant if we need to.

For example by defining a `Cat` class we can fix this error:

```
class Cat
end

animal = Cat.new
```

Summary

Any program can `raise` an exception when it runs into an unexpected situation, this lets users of this program know that there is a problem & the request couldn't be fulfilled.

If you are using some code (like a gem) that raises an exception you can `rescue` that exception to respond to it. If an exception is not handled then your program will crash with an error message & a stack trace.

Both the error message & the stack trace are very important to help you solve the problem (pay attention to the file name, error type & file number).

Strings

Contents

How to Use Strings in Ruby

In this chapter you may **discover new ways to work with strings** that you haven't seen before.

Or maybe it's just a review for you.

That's ok.

Both are good reasons to read & study this chapter :)

Putting Strings Together

It's pretty common to want to combine multiple strings into a larger one.

You can do this in a few different ways.

Like adding them together using the addition method:

```
first_name  = "Peter"
second_name = "Cooper"

first_name + " " + second_name
```

Or you can embed values inside another string.

We call this "string interpolation".

It results in cleaner code since you don't need all those plus symbols.

```
animal = "cat"

puts "Hello, nice #{animal} you have there!"
```

Using string interpolation has an additional benefit, it will call the `to_s` method on your variables.

This means that you can use any objects without running into a `TypeError` exception.

Example:

```
age = 18

puts "Sorry, you need to be #{age} or older to enter."
```

Remember:

To use string interpolation you need to use double quotes. A string in single quotes ignores all the special characters.

String Concatenation

You learned how to add together two or more strings using the `String#+` method.

But there is another way to do this:

The `String#<<` method.

The difference is that `String#+` creates **a new string**, while `String#<<` does not.

Example:

```
report = ""
report << "concatenating "
report << "strings "
report << "all day"
```

Creating new strings is slower than changing an already existing string.

> **Tip**: if you want the same string repeated x times you can do it like this: "A" * 20

String Justification

The `ljust` & `rjust` methods allow you to align values in a column.

Example: print in table format

```
names_with_ages =
[
  ["john", 20],
  ["peter", 30],
  ["david", 40]
]

names_with_ages.each do |name, age|
  puts name.ljust(10) + age.to_s
end
```

Prints the following table:

```
john      20
david     30
peter     40
```

The `rjust` method can also be used to **pad a string with any character you want**.

Example:

```
binary = "110"

binary.rjust(8, "0")
# "00000110"
```

Manipulating Strings

There are many ways to manipulate a string.

For example, you may want to split it into different parts, or you may want to replace certain words.

The `gsub` method (it means "Global Substitution") is very useful to **replace parts of the string or even remove them**.

Example: replace all 'dogs' with 'cats'

```
"My friend really likes dogs".gsub("dogs", "cats")
```

The gsub method is **very powerful**.

It can be used with an array of all the strings you want to replace & the Regexp.union method to replace multiple words at the same time.

Example:

```
animals_to_replace = ["tigers", "elephants", "snakes"]

"In this Zoo we have tigers, elephants & snakes"
  .gsub(*Regexp.union(animals_to_replace), "cats")
```

You can also use it with a block:

```
"a1 b2 c3".gsub(/\d+/) { |number| number.next }
```

Spliting Strings

You can also break up a string into pieces using the `split` method.

This method looks for a separator character to be able to tell where to make the cut. By default, `split` will use whitespace as the separator, but you can use anything you want.

The `split` method returns an array which you can put together again by using `join`.

Example: using split

```
str = "ABCD-123"

puts str.split("-")
# => ["ABCD", "123"]
```

Removing Newlines

If you are reading user input or lines from a file you will often want to remove the newline character.

The `chomp` method can take care of this for you.

> **Tip**: The newline special character is represented by `\n`. You can see this special character while inspecting a string (with the `p` method) instead of printing it.

Example:

```
puts "Please type your name: "
name = gets.chomp
```

> **Tip**: chomp removes the newline character by default, but it can also take a parameter to let you decide what to remove. It will only remove this character if it's the last character in the string.

How to Get a Substring

Another thing you can do with strings is to use array-like indexing to get one character or some of them.

Example: slicing a string

```
"hello there"[0..4]
# "hello"

"hello there"[1..-1]
# "ello there"
```

The `String` class has many other useful methods like `upcase / downcase`, `capitalize`, `reverse`, `include?`, `start_with?`, `end_with?`...

Use `pry` and the Ruby documentation to discover even more methods!

Working with Characters

Sometimes you need to work with individual characters, there are two main methods to do that in Ruby: `each_char` and `chars`.

For example:

```
"Ruby is cool".each_char { |ch| puts ch }
```

Since Ruby 2.0, `chars` is equivalent to `each_char.to_a`:

```
"Many words".chars
# => ["M", "a", "n", "y", " ", "w", "o", "r", "d", "s"]
```

One possible application is to combine `chars` with `map` to convert all the letters to their ASCII values:

```
"ascii".chars.map(&:ord)
# => [97, 115, 99, 105, 105]
```

Immutability and Frozen Strings

Strings in Ruby are mutable, which means you can change them. This has some implications for the performance and memory usage of your Ruby programs. Mutable strings are one of the reasons (if not the only one) that symbols exist.

In this example you can see string mutation in action:

```
str    = "My cat is black"
str[0] = ""

puts str
# Output: "y cat is black"
```

You can make a string immutable by using the `freeze` method. Trying to modify a frozen string will raise an exception.

Example:

```
str.freeze
str[0] = ""

# RuntimeError: can't modify frozen String
```

Since Ruby 2.1 frozen strings act like symbols.

> **Remember**: A symbol (like :bacon or :apple) is a kind of Ruby object that is used as an identifier, for hash keys, method arguments, etc. Don't confuse them with variables, symbols ARE values, variables POINT TO values.

Using the `object_id` method you can get the internal id of an object.

With this id you can verify that two objects are the same.

```
str_1 = "cat".freeze
str_2 = "cat".freeze

str_1.object_id == str_2.object_id # true
```

Two non-frozen strings will have different object ids, even if they have the same content.

```
"cat".object_id == "cat".object_id

# false
```

This changes if you enable the `frozen-string-literals` Ruby option.

Example:

```
$ RUBYOPT="--enable frozen-string-literal" irb

irb(main):001:0> "cat".object_id == "cat".object_id
=> true
```

More About Frozen Objects

You can use the `frozen?` method to check if a string is frozen.

Freezing is not limited to strings, any object can be frozen.

Once an object has been frozen it can't be unfrozen.

Be careful when freezing objects that can contain other objects (like an array), the contained objects won't be frozen. Freezing is only one level deep.

Example:

```
animals = ["dog", "cat"].freeze
animals[0].replace("cat")

p animals
# Output: ["cat", "cat"]
```

You don't have to freeze all your strings, but if you have the same string being used in many different places then it might be worth freezing that specific string.

Rails does this in a few places to save you memory.

Arrays

Contents

Introduction

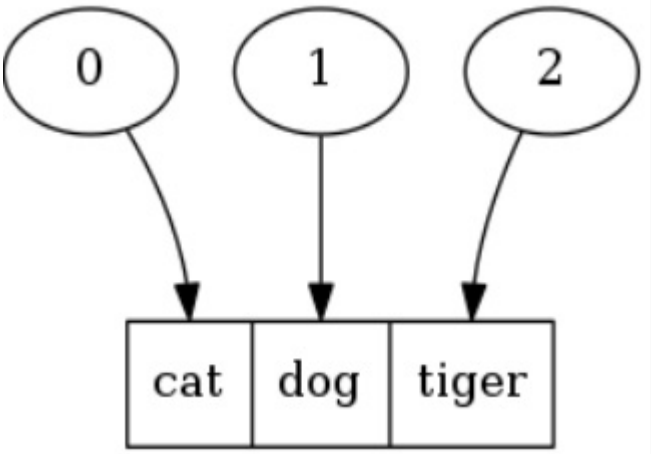
If you're already familiar with arrays I think it would be a good idea for you to read this chapter looking for some new Array methods that might be new to you.

Let's talk about arrays...

Arrays are a fundamental data structure that can store many things in one place. In Ruby you can store anything in an array, from strings to integers & even other arrays.

The elements in an array are **accessed using their index**, which starts at 0.

This is what an array containing the words "cat", "dog" and "tiger" would look like:



Basic Operations

To work with an array you will have to create one first, there are multiple ways to do it.

Initialize an empty array:

```
# Both of these do the same thing, but the second form is preferred
users = Array.new
users = []
```

Initialize an array with data:

```
users = ["john", "david", "peter"]
```

Another way to initialize your array with strings:

```
# Same effect as the last example, but faster to type
users = %w(john david peter)
```

Now let's see a few different ways to read, add and delete items from arrays.

Accessing array elements by index:

```
users[0] # First element of the array
users[1] # Second element of the array
users[2] # Third element of the array
```

Using the `first` and `last` methods:

```
users.first # First element of the array
users.last  # Last element of the array
```

These methods can also take an optional argument to get more than one element from the array:

```
users.first(3) # First three elements of the array
users.last(3)  # Last three elements of the array
```

Adding items to the array:

```
# Both of these have the same effect, << is prefered.
users.push "andrew"
users << "andrew"
```

Deleting items from the array:

```
# Delete the last element from the array
# and return it
last_user = users.pop

# Delete the element at index 0
users.delete_at(0)
```

There is also the `shift` / `unshift` methods, which are similar to pop/push but take or add elements in front of the array.

```
# Adds an element in front of the array
users.unshift "robert"

# Removes the first element of the array & returns it
users.shift

# Removes 3 elements from the array & returns them as a new array
users.shift(3)
```

What if you wanted to add something in the middle of an array? It's not a very common operation, but you can do it using the `insert` method.

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

letters.insert(2, 'test')
```

To get the size of an array (the number of elements contained in it), you can use either the `size` method or the `length` method, they are both the same:

```
users.size # 3
```

You should also know the `count` method. If a block is given to `count`, it counts the number of elements for which the block returns a true value.

For example, you can use this to count how many even numbers you have in an array.

```
[1,2,3].count { |n| n.even? }
```

Here is a small cheat-sheet for you:

Action	Method
Initialize	Array.new, [], %w
Read	[0], first, last
Append	push, <<, unshift
Remove	pop, delete_at, shift

Iterating Over Arrays

Now that you have got an array wouldn't it be nice if you could enumerate its contents and print them?

Example: Print an array using `each`

```
users.each { |item| puts item }
```

Sometimes you need the index, in those cases you can use the `each_with_index` method.

> **Note**: Most of these iteration operations are available thanks to the Enumerable module, which is mixed into the `Array` class by default. Array does provide a good amount of methods too. Investigate using pry!

Example: Capitalize every word in the Array using `map`.

```
users = users.map { |user| user.capitalize }
```

The `map` method doesn't modify the array in-place, it just returns a new array with the modified elements, so we need to assign the results back to a variable.

There is also a `map!` (notice the exclamation point) method which will modify the array directly, but in general the normal version is preferred.

The `flatten` method moves any nested array elements into the main array, making a 'flat' array.

Another way to think about this is that you are removing the extra layers & just putting everything on the same level.

Example: Flat arrays

```
[[1, 2, 3], [4, 5, 6]].flatten

# Output: [1, 2, 3, 4, 5, 6]
```

Another useful method is `flat_map`. This method combines the effect of calling `map` and then calls `flatten` on the results.

Example: Flat + Map

```
[[1, 2, 3], [4, 5, 6]].flat_map { |a| a.inject(:+) }

# Output: [6, 15]
```

Another thing you may want to do is to find all the items in your array that fit certain criteria.

Example: Find all the numbers greater than 10:

```
numbers = [3, 7, 12, 2, 49]
numbers.select { |n| n > 10 }
# => 12, 49
```

More Array Operations

There are a lot of things you can do using arrays, like sorting, reversing or picking a random element.

You can use the sort method to sort an array, this will work fine if all you have is strings or numbers in your array. For more advanced sorting check out sort_by.

```
numbers = numbers.sort
```

You can also remove the duplicate elements from an array. If you find yourself doing this often you may want to consider using a Set instead.

```
numbers = [1, 3, 3, 5, 5]
numbers = numbers.uniq
# => [1, 3, 5]
```

If you want to pick one random element from your array you can use the `sample` method:

```
numbers.sample
```

You may also want to "slice" your array, taking a portion of it instead of the whole thing.

Example: Take the first 3 elements from the array, without changing it:

```
numbers.take(3)
numbers[0,3]
```

The Splat Operator

In Ruby you can do this:

```
numbers = *[1, 2], 3
```

This will return an array: `[1, 2, 3]`.

You might be wondering what in the world is this black magic, but don't worry, the explanation is easier than it looks.

The key here is the `*` symbol, which is also known as the 'splat operator' in Ruby. And what it does is expand an array into its individual elements.

Ruby converts the result (`1, 2, 3`) into another array so it can assign that array to the `numbers` variable.

And that's how `*[1, 2], 3` becomes `[1, 2, 3]`.

More Splatting

The splat operator can also be used on the left-hand side of a variable assignment. When you do that, what happens is that the variable name with the splat will try to grab as many values as possible and wrap them in an array.

Here is an example:

```
*a, b = 1,2,3,4

# a = [1, 2, 3]
# b = 4
```

Another example:

```
a, *b, c = 1,2,3,4

# a = 1
# b = [2, 3]
# c = 4
```

In other words:

A variable with a splat operator works like a "sponge" trying to absorb as many elements as possible, while still leaving one element for every other variable.

Operations Involving Multiple Arrays

If you have two arrays and want to join them you can use the `concat` method.

```
# Faster, it changes the original
users.concat(new_users)

# Slower, this creates a new array
users += new_users
```

You can extract one array from another using the minus `-` operator.

```
a = [1,2,3,4]
b = [4]

a - b
# Output: [1,2,3]
```

To get the the elements that appear in both arrays at the same time, you can do this:

```
a = [1,2,3]
b = [2,4,5]

a & b
# Output: [2]
```

The `|` operator will give you the union of two arrays with duplicated elements removed:

```
a = [1,2,3]
b = [2,4,5]

a | b
# Output: [1,2,3,4,5]
```

Hash Tables

Content

How to Use Hash Tables

Hash tables let you store `key -> value` pairs.

You can think of them as a dictionary, where **you can look-up a word in the dictionary & that word has some associated definitions**.

In Ruby, the `Hash` class lets you create new hash objects.

The keys can be anything, including strings, symbols & numbers.

Some common uses for hashes include:

- Counting things (like words from a text)
- As a dictionary (Examples: word to definition, product name to value, country code to country name...)
- As a pseudo-class to store attributes (like a book title, author, genre...)

> For this last use a better tool is the Struct class.

Basic Usage

You can create an empty hash like this:

```
prices = {}
```

Then you can add values like this:

```
prices[:bacon] = 100
```

To get the value you use the same syntax, but without the equals sign:

```
prices[:bacon]

# 100
```

Remember that **keys are unique**.

This means that you can only have one `:bacon` key in a given hash.

Iterating Over Hash Values

It can be useful to iterate over hash values and keys.

The `each` method makes this easy for us:

```
# Create a hash with pre-defined values
h = { water: 300, oil: 100 }

# Print each key / value pair
h.each do |key, value|
  puts "The key is #{key} with value #{value}"
end
```

There is also `each_value` and `each_key` if you only need to iterate over one side of the hash. Alternatively, you can use the `keys` method and the `values` method to get an array of either.

Example:

```
h = { name: 'John', age: 30 }

h.keys
# => [:name, :age]

h.values
# => ["John", 30]
```

If you want to check if a key exists use the `key?` method:

```
h = { name: 'David', age: 28 }

h.key?(:name)
# => true

h.key?(:country)
# => false
```

Default Values

One possible use for hash objects is to serve as counters or accumulators. One way to do that could be like this:

```
str   = "aabbbaa"
count = {}

str.each_char do |c|
  if count.key?(c)
    count[c] += 1
  else
    count[c] = 1
  end
end

puts count
# Output: {"a"=>4, "b"=>3}
```

In this code we are counting the number of `a` and `b` in the string & we store the result inside a hash.

Notice that we need to **check for the key & initialize it** to `1` if it doesn't exist.

If we don't do this we'll get an error!

In Ruby there are many ways to do the same thing.

For example, you can take advantage of the fact that the `Hash` class can take **a default value**.

Using a default value we can get rid of the `if / else` & end up with this much cleaner code:

```
str   = "aabbbaa"
count = Hash.new(0)

str.each_char { |c| count[c] += 1 }

puts count
```

This default value (`0`) is perfect for this example, but there is a catch.

Take a look at the following example:

Example: An unexpected result

```
h = Hash.new([])

h[:foo] << 5
h[:bar] << 10

h[:foo]
# [5,10]
```

> Notice that we aren't calling `<<` on a hash. We're calling `<<` on the array which is returned as a result of h[:foo].

Why does `h[:foo]` have a value of `[5,10]` instead of just `[5]`?

Because we are changing the default value!

It's the same array for both keys.

We can avoid this problem using a block for the default value, instead of an argument.

Example: Set default value with a block

```
h = Hash.new { |hash, key| hash[key] = [] }

h[:foo]
# Output: []

h[:bar] << 5
# Output: [5]

h.keys
# Output: [:foo, :bar]
```

In this example, I'm calling `h[:foo]` to get the current value of that key.

Because the key doesn't exist the block will be evaluated.

The result?

The missing key will be created with the default value.

But this time it will be **a copy of that default value** & not the same object for every key, which avoids the problem we had before.

Using Fetch and Merge

If you try to query a hash for a key that doesn't exists and it isn't using a default value, then you will get a `nil`.

This will probably cause an exception down the road, and it might be hard to see where it came from.

Example:

```
h = {}

h[:invalid_key]
# nil
```

Use the `fetch` method instead & you'll get a more useful error:

```
h = {}

h.fetch(:invalid_key)
# KeyError: key not found: :invalid_key
```

You can also use `fetch` to give you a default value.

This is different from setting a default value directly when creating a hash.

Using `fetch` with a default value won't set a permanent default or create a new key in the hash.

Example:

```ruby
h = {}

h.fetch(:invalid, 'default')
# 'default'
```

You can also use the block form:

```ruby
h = {}

h.fetch(:invalid) { 'default' }
# 'default'
```

The difference is that in the first version, **the default parameter will always be evaluated**. In other words, if you have a method as your default, this method will be called, even if the key exists.

With the block version that won't happen :)

How to Merge Two Hashes

Another interesting method is `merge`, this method lets you merge two hash objects.

If a value already exists, it will be overwritten by the newest one.

This method (`merge`) returns a new hash, if you want to modify the hash in-place you can use `merge!`.

```
h = { name: 'Jose', age: 29 }

h.merge(age: 30, city: 'Spain')
# => { name: 'Jose', age: 30, city: 'Spain' }
```

A typical application of `merge` is to update a set of defaults with some options passed into the method.

Example:

```ruby
class UrlFetcher
  attr_reader :config

  def initialize(url, options = {})
    @config = {
      url:      url,
      port:     80,
      verb:     "get",
      protocol: "http"
    }

    @config.merge!(options)
  end
end

fetcher = UrlFetcher.new("example.com", port: 8080)
p fetcher.config
```

The Dig Method (new in Ruby 2.3)

If you want to traverse a hash which has more hashes nested in it (this in common with JSON) then you could do something like this:

```
the_hash = {
  user: {
    name: "Peter",
    country: "UK",
    age: 30
  }
}

the_hash.fetch(:user).fetch(:name)
# "Peter"
```

That's fine...

...but imagine that for some reason the `:user` key doesn't exist.

What would happen is that **you would get an exception**, even if you use the `[]` syntax to access the keys.

Ruby 2.3 introduced a new method (`dig`) which solves this problem.

If any of the keys don't exist you will get a `nil` back.

Example:

```ruby
the_hash.dig(:user, :country)
# "UK"

the_hash.dig(:something, :country)
# nil
```

The Transform Values Method (New in Ruby 2.4)

If you want to change all the values in a hash then the new `transform_values` method introduced in Ruby 2.4 is what you're looking for.

Here's an example:

```
fruit = {
  banana: 1,
  apple: 2,
  orange: 3
}

fruit.transform_values { |v| v * 10 }

# {:banana=>10, :apple=>20, :orange=>30}
```

You can find a video here with this method in action.

Ruby Sets

A set is a Ruby class that helps you create a list of unique items.

Here's an example of how this is useful :

Let's say you are going over a big list of products.

But there are **duplicated entries** in this list & you only want unique products.

You could put them in a set, and the set will make sure your product list is always unique **without any extra work**.

Here's how to do that:

```
require 'set'

products = Set.new

products << 1
products << 1
products << 2

products
# Set: {1, 2}
```

Another benefit is that searching this list is going to be very fast:

```
products.include?(1)
# true
```

If all you're doing is adding items to an array & searching to see if an item is included or not, **a Set can improve your performance 10x**.

How are sets related to hashes?

A Set is a hash wrapped around with a different interface. A different group of methods that help you use a hash in a different way.

Ranges

Ranges let you define intervals of characters, numbers & time.

This can be useful if you need something like all the letters in the alphabet or a list of numbers in a certain interval.

It's good to know that `Range` includes `Enumerable` so you can use ranges to iterate without having to convert them to an `Array` first.

If you still need an array you can call `to_a` on the range.

There isn't much else to say about ranges, so I'm just going to give you a few examples that you can use as a reference.

Range of characters:

```
('a'..'z').to_a
# => ["a", "b", "c", "d", "e", "f", "g", "h", ...
```

Inclusive range:

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

Exclusive range:

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

Using the `step` method:

```ruby
(2..15).step(2).to_a
# => [2, 4, 6, 8, 10, 12, 14]
```

Iterating over a range:

```ruby
(1..15).each { |n| puts n }
```

Get a random number in a specific range:

```ruby
rand(1..10)
```

Check for inclusion:

```ruby
(Time.now..Time.now + 60).include?(Time.now + 30)
# true
```

Using `cover?`:

```ruby
('a'..'z').cover?("c")
```

The difference between `include?` & `cover?` is important.

Include will start with the first element in the range, then ask for the next element until it finds the end of the range.

This is going to create new objects.

`cover?` will compare your target against the ending & starting ranges **without creating new objects**.

Keep in mind that `cover?` can give you different results (specially with strings).

Here's an example:

```
('a'..'z').include? "cc" # false
('a'..'z').cover? "cc"   # true
```

This happens because string comparison is made character by character, `c` is after `a` but before `z`.

Data Serialization

Data serialization is the process of converting in-memory data structures (hash, array, etc.) into a format that can be sent over the network or saved into files.

Serialization formats can be classified in two groups:

- Plain-text
- Binary

Binary formats are usually more efficient, but they have the disadvantage of not being human-readable. In this lesson I'm going to cover some popular plain-text formats.

> **Note**: Wikipedia has a list of serialization formats that you may want to take a look at.

The format we're going to cover:

- CSV (Comma-Separated Values)
- JSON
- YAML

CSV

CSV stands for comma-separated values.

It's the simplest form of serialization you'll find!

A CSV document is composed of rows, and every row is composed of values separated by a comma.

In Ruby, we have access to a built-in `CSV` class that will allow us to easily work with this format.

Example:

```
require 'csv'

puts CSV.read("file.csv")
```

A csv file can have headers.

We can tell the `CSV` class to read these headers so we can use them.

Let's say we have the following file :

```
id,name,age
1,Peter,30
2,James,23
3,Fred,42
```

Then we can use the following code :

```ruby
file = CSV.read("users.csv", headers: true)

puts file.headers
# => ["id", "name", "age"]

file.each { |row| puts row["name"] }
# Peter
# James
# Fred
```

Notice that you get a `CSV::Table` object instead of an array, so you need to use `each` to work with the rows. You can reference a specific field in a row using hash-like syntax & the field name.

Uses:

- Easy import into spreadsheet software (like Excel or Google Docs).
- Very common format for exporting lists & tables.

YAML

YAML is a serialization format that is based on indentation levels.

For example...

A nested array serialized into YAML will look like this:

```
-
 - foo
 - bar
 - baz
```

To work with YAML you can use the built-in YAML class and the `load` and `dump` methods.

Example: Loading a YAML file

```
require 'yaml'

nested_array = YAML.load_file("array.yml")
# => [["foo", "bar", "baz"]]
```

Example: Serializing objects

```
require 'yaml'

Book     = Struct.new(:title, :author)
eloquent = Book.new('Eloquent Ruby', 'Russ Olsen')

serialized_book = YAML.dump(eloquent)
```

This won't save the book object directly to a file...

You will need to do this:

```
File.write("books.yml", serialized_book)
```

Instead of `dump` you can also use the `to_yaml` method:

```
File.write("books.yml", eloquent.to_yaml)
```

Reference: http://yaml.org/YAML*for*ruby.html

Uses: - In the Ruby world, and more specifically in Rails, the YAML format is used for configuration files. - YAML is also used for [test fixtures]

JSON

JSON stands for 'Javascript Object Notation'.

The JSON format might remind you of Ruby hashes, with the difference that keys are always strings.

Like the other formats, Ruby also has a built-in class to deal with JSON. Let's see how we can read a JSON string and convert it into a Ruby hash for easy manipulation.

```
require 'json'

json = '{"water": 300, "oil": 200}'
hash = JSON.parse(json)
```

We can also do this the other way around and convert a Ruby hash into a JSON string.

```
require 'json'

hash = {
  water: 500,
  oil: 100
}

p hash.to_json
```

> **Tip**: If you have a complex hash you can use the `jj` method to pretty-print it. Alternatively, you can use the awesome_print gem.

Uses:

- Very often used as the format of choice when interacting with web-based APIs.
- Many times you can use an API client that will give your Ruby objects instead of raw JSON. An example of that is the Octokit gem for the Github API.

The Ruby Object Model

Contents

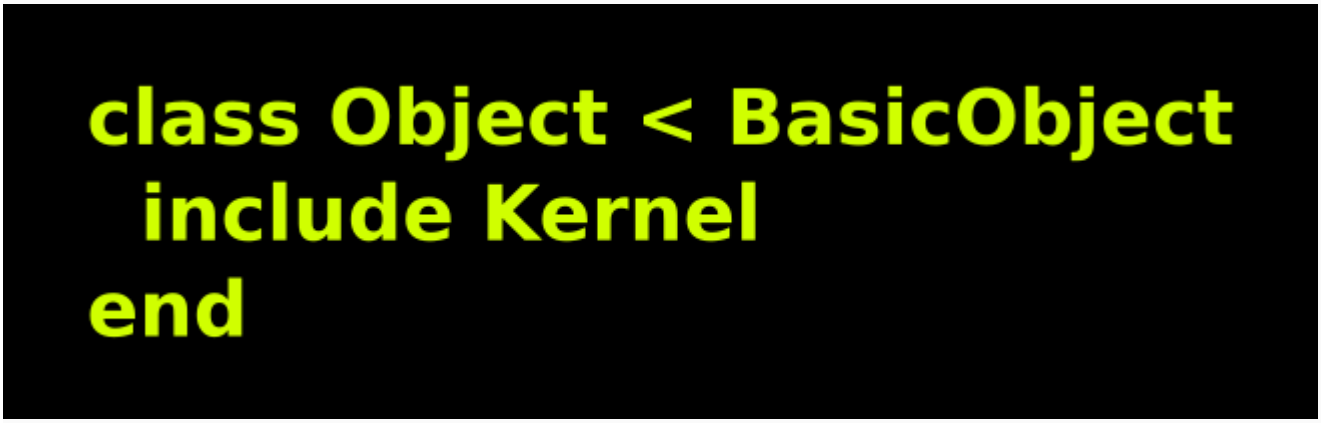
The Ruby Object Model

As you may know, Ruby has been designed as an Object-oriented language from the ground up (unlike many other languages where OOP is just an extra feature).

Everything in Ruby is an object.

All classes in Ruby inherit from the `Object` class by default. `Object` inherits from the `BasicObject` class (which was introduced in Ruby 1.9).

`Object` includes the `Kernel` module, which is where many useful methods like `puts` & `inspect` are defined.



> **Experiment**: Investigate the Object class and the Kernel module using pry's ls command, note down interesting things you notice.

You can ask for the class name of an object using the `class` method:

```
ages = [18, 22, 25, 30]
ages.class
# => Array

20.class
# => Integer
```

> **Tip**: It's useful to know what class you are working with so you know what methods are available. Get into the habit of checking the object's class whenever you are unsure how to make progress.

Classes are objects too, in fact they are instances of the class `Class`.

Try this:

```
Array.class
String.class
Class.class
```

This shows how you can make a new class by creating a `Class` object:

```
Person = Class.new
john   = Person.new

# <Person:0x429b010>
```

Note: The 0x429b010 part in <Person:0x429b010> is a hexadecimal number (base 16) that represents the object's id * 2.

Super-classes

You can find the parent classes & modules for any class using the `ancestors` method.

For example:

```
Array.ancestors
# [Array, Enumerable, Object, Kernel, BasicObject]
```

You will notice that these are in a specific order.

That's because when you call a method on `Array` that's the order in which Ruby will look for that method.

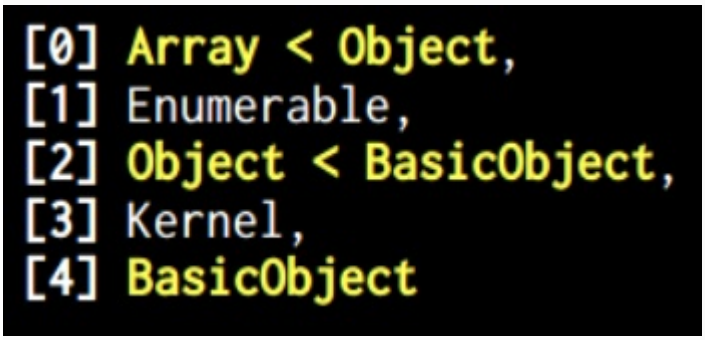
> **Note**: pry will add this `PP::ObjectMixin` module to all your objects, you can ignore it.

You can use the `included_modules` method to find out which of those are actually modules:

```
Array.included_modules
# [Enumerable, Kernel]

Array.ancestors - Array.included_modules
# [Array, Object, BasicObject]
```

Here is another way to visualize this.

Class names are in yellow & modules are in white.



If you are curious, I used the `awesome_print` gem to get this output.

> **Experiment**: Class is a subclass of Module, which might be a bit surprising to you if you haven't seen that before. What other interesting class relationships can you find using the ancestors method? For example, you could start with Integer.

The Singleton Class

There is a special kind of class in Ruby known as the 'singleton class' (also know as 'eigenclass' or 'metaclass').

This in an important part of how Ruby works that is not completely obvious.

> **Note**: Don't confuse this with the singleton design pattern.

Every object in Ruby can have one singleton class associated with it.

You can think of this singleton class as an 'auxiliary' class that only exists to hold methods. I will expand on why singleton classes are important in the next section, but for now let's see how you can interact with them.

You can get access to the singleton class of any object with the `singleton_class` method.

Example:

```
Array.singleton_class

# => #<Class:Array>
```

Are you familiar with class methods?

Well, class methods don't live on objects directly, they live in singleton classes.

You can see this in action in the following example.

Experiment: use the `singleton_methods` method on a class which has at least one class-level method.

```
class Animal
  def self.test
  end
end

p Animal.singleton_methods
# What's the output?
```

A singleton class also allows you to define methods that only exist in a single object, without altering the original class.

Example:

```
str = "test"

def str.show
  puts self
end

p str.singleton_methods
# => [:show]
```

There is another way to define methods on the `singleton_class`.

Take a look at the following example:

```ruby
str = "alternative syntax"

class << str
  def display
    puts self
  end
end

str.display
```

This can also be used inside a class definition by using `self` as the object.

For example:

```ruby
class Animal
  class << self
    def run
    end

    def eat
    end
  end
end

p Animal.singleton_methods
```

> **Note**: the class << self syntax is not very popular these days, but it's still good to know that it exists :)

The practical implications of this is that if you want to define class-level methods you define them on the singleton class.

For example, to define a class-level `attr_reader`:

```
class Cat
  singleton_class.send(:attr_reader, :age)
end

Cat.age
# nil
```

Why do we need this singleton class?

To answer that we need to understand the difference between an instance of the `Object` class and an instance of the `Class` class.

I think the best way to understand this is to dive into MRI (Matz Ruby Interpreter) source code.

> **Note**: Don't worry if you don't know any C programming. This code is very simple and I will explain what's going on.

This is the internal representation for an object instance:

```
struct RObject {
  struct RBasic basic;
  struct st_table *iv_tbl;
};
```

An `struct` in C is a way to hold multiple values in the same place, it's like a class but without methods.

The fields of this `struct` are also structs themselves, `iv_tbl` points to a list of instance variables for this object, and `basic` contains the class name for this object plus some flags (frozen, tainted...).

The main point to drive home from this code snippet is that objects don't have methods.

So where do all methods live?

Let's take a look at the code for a class instance.

```
struct RClass {
  struct RBasic basic;
  struct st_table *iv_tbl;
  struct st_table *m_tbl;
  VALUE super;
};
```

We can see that the first two lines are exactly the same, but the next two are very interesting.

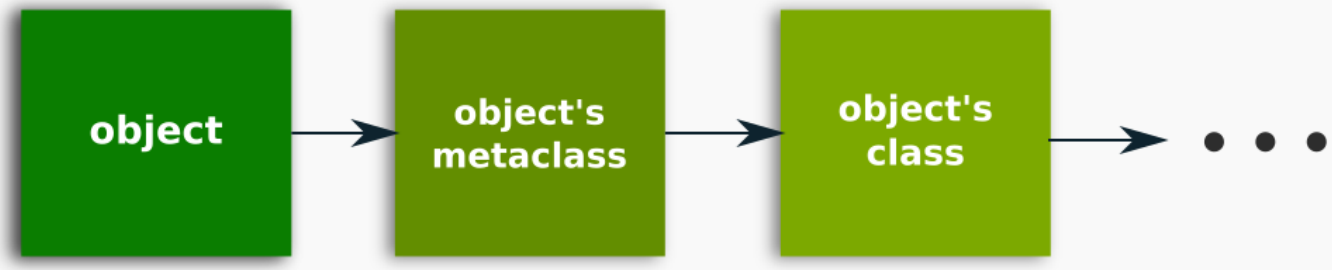
The struct `m_tbl` means "method table" and it points to the list of methods for this class. It looks like we found our first answer: **objects don't have methods, classes do**.

So what does this have to do with singleton classes?

Well if you want to add methods to an object, and you don't want to change the class, the only way to accomplish this is to add an extra class to host these methods.

The singleton class lives between the object itself and the object's class.

It looks something like this:



Remember: 'singleton class', 'meta class' and 'eigen class' are all names for the same concept.

Constants in Ruby

Constants are an important part of Ruby, mostly because they are used as class names, in addition to giving names to non-changing values like the number Pi.

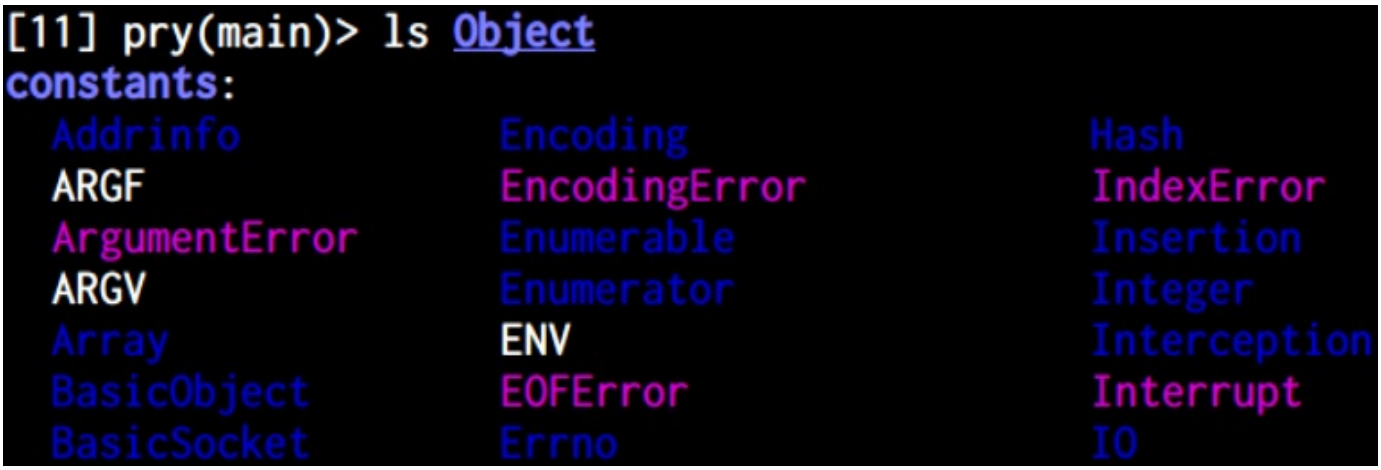
> **What's a constant?** In Ruby anything that starts with a capital letter is considered a constant. Unlike in some other languages, constants can be changed (but you will get a warning).
>
> Special case: Method names that start with uppercase letters are not constants, for example Array() is a method on Kernel. To call a method defined this way you need to use parenthesis.

Let's talk about how constants work in Ruby.

The first thing you need to know is that when you define a constant it becomes a part of the current class.

That might sound obvious, but since class names are constants that's why you can access core classes from anywhere in your code (remember that all objects inherit from `Object` by default).

If you took a look at the output of `ls Object` on `pry` you might have noticed all the constant names (and btw this is how I discovered this myself!). You can also use `Object.constants` to get the same information.



In other words:

When you define a constant at the top-level scope (outside of any module or any class) it becomes part of `Object`.

This allows you to do something like this, which is fun but useless in real code:

```
# The worst way to create a string!
Fixnum::Class::Object::String.new("testing")

# warning: toplevel constant Class referenced by Fixnum::Class
# warning: toplevel constant Object referenced by Class::Object
```

The fact that we get some warnings is a sign that we are doing something wrong :)

Constant Lookup

When inside a nested class or module, finding a constant is a bit more complicated. Ruby uses `Module.nesting` to find what the outer modules are and ask them for constants.

Here is a formula that always produces the correct list of classes that will be looked up.

```
modules =
  Module.nesting +
  (Module.nesting.first || Object).ancestors

modules +=
  Object.ancestors if Module.nesting.first.class == Module
```

So if we have something like this:

```
module Foo
  X = 10

  class Bar
    puts X
  end
end
```

The constant lookup will be done in this order:

```
[Foo::Bar, Foo, Foo::Bar, Object, Kernel, BasicObject]
```

Yeah, `Foo::Bar` is in there twice, presumably the second lookup is to find its ancestors.

Ruby Methods

Another thing you need to understand about Ruby is that there is a lot of "syntax sugar" going on.

> **Definition (syntax sugar)**: Nice-looking syntax for something that would otherwise look ugly or more complicated than it needs to be.

Let me give you an example:

```
2 + 2
```

Nothing special about this, but what is really happening here is:

```
2.+(2)
```

> **Experiment**: Try this in irb!

Yep, `+` is just a method on the `Fixnum` class.

> **Note**: Since Ruby 2.4 Fixnum is deprecated, now it's just Integer.

This is also the case with other operators, even `==` is a method. You can also override these methods if you want.

Example:

```
class Fixnum
  def +(other)
    "No addition for you!"
  end
end

puts 1 + 1
```

> **Note**: This example is fun but never do this on real code.

Here is another example:

```
"abcde"[1]
```

Is in reality:

```
"abcde".[](1)
```

You can also implement the `[]` and the `[]=` methods in your own classes if you want.

Example:

```
class UserRepository
  def initialize
    @users = ['Peter', 'James', 'Luke']
  end

  def [](idx)
    @users[idx]
  end
end

users = UserRepository.new
puts users[0]
# Output: Peter
```

In this case we are just delegating the `[]` method without any special logic.

In Ruby you can also use the Forwardable module to get the same result, but with a performance penalty.

Procs & Lambdas

Content

- Understanding Ruby Blocks
- The Yield Keyword
- What is a Lambda?
- Lambdas vs Procs
- Closures
- Using to_proc

Ruby blocks, Procs & Lambdas

What are they?

How do they work?

How are they different from each other?

You will learn that & a lot more by reading this chapter!

Understanding Ruby Blocks

Ruby blocks are little anonymous functions that can be passed into methods.

Blocks are enclosed in a `do / end` statement or between brackets `{}`, and they can have multiple arguments.

The argument names are defined between two pipe `|` characters.

If you have used `each` before, then you have used blocks!

Here is an example:

```
# Form 1: recommended for single line blocks
[1, 2, 3].each { |num| puts num }

# |num|      => block arguments
# puts num => block body
```

```
# Form 2: recommended for multi-line blocks
[1, 2, 3].each do |num|
  puts num
end
```

Ruby blocks are useful because they allow you to save a bit of logic (code) & use it later.

This could be something like:

- Writing data to a file

- Comparing if one element is equal to another
- Printing an error message

Or anything else that you can normally do with Ruby.

Ruby Yield Keyword

What does `yield` mean in Ruby?

Yield is a **Ruby keyword** that calls a block when you use it.

It's how methods USE blocks!

When you use the `yield` keyword, **the code inside the block will run** & do its work.

Just like when you call a regular Ruby method.

Here's an example:

```
def print_once
  yield
end

print_once { puts "Block is being run" }
```

This runs any block passed to `print_once`, as a result, `"Block is being run"` will be printed on the screen.

Running Blocks Multiple Times

Did you know...

That `yield` can be used multiple times?

Every time you call `yield`, the block will run, so this allows you to use the same block multiple times within one method.

Example:

```
def print_twice
  yield
  yield
end

print_twice { puts "Hello" }

# "Hello"
# "Hello"
```

And just like methods...

You can pass any number of arguments to `yield` .

Example:

```ruby
def one_two_three
  yield 1
  yield 2
  yield 3
end

one_two_three { |number| puts number * 10 }
# 10, 20, 30
```

These arguments then become the block's arguments.

In this example `number`.

Open "pry" or "irb" now & give this a try to see how it works.

Implicit vs Explicit Blocks

Blocks can be "explicit" or "implicit".

Explicit means that you give it a name in your parameter list.

You can pass an explicit block to another method or save it into a variable to use later.

Here is an example:

```
def explicit_block(&block)
  block.call # same as yield
end

explicit_block { puts "Explicit block called" }
```

Notice the `&block` parameter...

That's how you define the block's name!

How To Check If A Block Was Given

If you try to `yield` without a block...

You will get a `no block given (yield)` error!

You can check if a block has been passed into your method with the `block_given?` method.

Example:

```
def do_something_with_block
  return "No block given" unless block_given?
  yield
end
```

This prevents the error if someone (including yourself) calls your method without a block.

What is a Lambda?

A lambda is a way to define a block & its parameters with some special syntax.

You can save this lambda into a variable for later use.

The **syntax for defining a Ruby lambda** looks like this:

```
say_something = -> { puts "This is a lambda" }
```

> You can also use the alternative syntax: lambda instead of ->.

Defining a lambda won't run the code inside it, just like defining a method won't run the method, you need to use the `call` method for that.

Example:

```
say_something = -> { puts "This is a lambda" }
say_something.call

# "This is a lambda"
```

There are other ways to `call` a `lambda`, it's good to know they exist, however, I recommend sticking with `call` for clarity.

Here's the list:

```ruby
my_lambda = -> { puts "Lambda called" }

my_lambda.call
my_lambda.()
my_lambda[]
my_lambda.===
```

Lambdas can also take arguments, here is an example:

```ruby
times_two = ->(x) { x * 2 }
times_two.call(10)
# 20
```

If you pass the wrong number of arguments to a `lambda`, it will raise an exception, just like a regular method.

Lambdas vs Procs

Procs are a very similar concept...

One of the differences is how you create them.

Example:

```
my_proc = Proc.new { |x| puts x }
```

There is no dedicated `Lambda` class. A `lambda` is just a special `Proc` object. If you take a look at the instance methods from `Proc`, you will notice there is a `lambda?` method.

Now:

A proc behaves differently than a lambda, specially when it comes to arguments:

```
t = Proc.new { |x,y| puts "I don't care about arguments!" }
t.call
# "I don't care about arguments!"
```

Another difference between `procs` & `lambdas` is how they react to a `return` statement.

A `lambda` will `return` normally, like a regular method.

But a `proc` will try to `return` from the current context.

Here's what I mean:

If you run the following code, you will notice how the `proc` raises a `LocalJumpError` exception.

The reason is that you can't `return` from the top-level context.

Try this:

```
# Should work
my_lambda = -> { return 1 }
puts "Lambda result: #{my_lambda.call}"

# Should raise exception
my_proc = Proc.new { return 1 }
puts "Proc result: #{my_proc.call}"
```

If the `proc` was inside a method, then calling `return` would be equivalent to returning from that method.

This is demonstrated in the following example.

```
def call_proc
  puts "Before proc"
  my_proc = Proc.new { return 2 }
  my_proc.call
  puts "After proc"
end

p call_proc
# Prints "Before proc" but not "After proc"
```

Here is a summary of how `procs` and `lambdas` are different:

- Lambdas are defined with `-> {}` and procs with `Proc.new {}`.
- Procs return from the current method, while lambdas return from the lambda itself.
- Procs don't care about the correct number of arguments, while lambdas will raise an exception.

Taking a look at this list, we can see that `lambdas` are a lot closer to a regular method than `procs` are.

Closures

Ruby procs & lambdas also have another special attribute. When you create a Ruby proc, it captures the current execution scope with it.

This concept, which is sometimes called closure, means that a `proc` will carry with it values like local variables and methods from the context where it was defined.

They don't carry the actual values, but a reference to them, so if the variables change after the proc is created, the proc will always have the latest version.

Let's see an example:

```
def call_proc(my_proc)
  count = 500
  my_proc.call
end

count   = 1
my_proc = Proc.new { puts count }

p call_proc(my_proc) # What does this print?
```

In this example we have a local `count` variable, which is set to `1`.

We also have a proc named `my`*`proc`*`, and a call`proc` method which runs (via the `call` method) any proc or lambda that is passed in as an argument.

What do you think this program will print?

It would seem like `500` is the most logical conclusion, but because of the 'closure' effect this will print `1`.

This happens because the proc is using the value of `count` from the place where the proc was defined, and that's outside of the method definition.

The Binding Class

Where do Ruby procs & lambdas store this scope information?

Let me tell you about the `Binding` class...

When you create a `Binding` object via the `binding` method, you are creating an 'anchor' to this point in the code.

Every variable, method & class defined at this point will be available later via this object, even if you are in a completely different scope.

Example:

```
def return_binding
  foo = 100
  binding
end

# Foo is available thanks to the binding,
# even though we are outside of the method
# where it was defined.
puts return_binding.class
puts return_binding.eval('foo')

# If you try to print foo directly you will get an error.
# The reason is that foo was never defined outside of the method.
puts foo
```

In other words, executing something under the context of a `binding` object is the same as if that code was in the same place where that `binding` was defined (remember the 'anchor' metaphor).

You don't need to use `binding` objects directly, but it's still good to know this is a thing :)

Using to_proc

Ruby syntax can be very elegant, take a look at this example:

```
["cat", "dog"].map(&:upcase)
```

How does this work?

The key is in the `&` symbol in `&:upcase`. Using the ampersand symbol like this calls the `to_proc` method on whatever object is on the right. In this case we have a symbol, so it will call `Symbol#to_proc`.

If you take a look at the documentation for the Symbol class, you will find this

> **to_proc**:
>
> Return a Proc object which responds to the given method name.

So when you do `%w(foo bar).map(&:upcase)`. What `&:upcase` is really doing is this:

```
Proc.new { |word| word.upcase }
```

Here is what a simple implementation of the `to_proc` method looks like:

```ruby
def to_proc(sym)
  Proc.new { |x| x.send(sym) }
end

to_proc(:size).call("test")
```

What is this `send` method?

You will learn about it in the metaprogramming lesson!

Video Tutorial

https://www.youtube.com/watch?v=92yuNm6Ts0c

Wrapping Up

In this chapter you learned how blocks work, the differences between Ruby procs & lambdas and you also learned about the "closure" effect that happens whenever you create a block.

Now it's time to practice :)

Require

Contents

Loading Code

If you had to have every line of code in the same file it would be really hard to manage projects that consist of more than a few methods & classes.

It would also be really hard to share code.

That's why methods like `require` exist in Ruby, to allow you to **load code from other files**.

But how does `require` work?

The key is in the `$LOAD_PATH` global variable. This variable holds a list of paths that will be searched for files to load.

> **Exercise**: Explore Ruby global variables. You can get a list of all the global variables using the global_variables method or with ls -g in pry. You can find descriptions here: https://github.com/ruby/ruby/blob/trunk/doc/globals.rdoc

When you use `require`, one of two things can happen:

- If the required file **is found**, it's loaded & added to the $LOADED_FEATURES array.
- If the file is **not found** in one of these paths (in $LOAD_PATH) then an exception is raised.

Example:

```
require 'json'
```

Notice that the return value for `require` is `true` if the file was loaded for the first time & `false` if it was already loaded.

Btw this is only for loading CODE.

If you want to read a text file use the `File` class & associated methods.

Loading Gems

Rubygems overrides the `Kernel#require` method to make loading of gems possible. Here is a simplified version of the code:

```
module Kernel
  def require(file)
    original_require(file)
  rescue LoadError
    loaded = load_gem(file)
    raise unless loaded
  end
end
```

Print Working Directory

Sometimes it may be useful to know exactly where your program is running.

Especially if you are running inside an editor like Atom, which changes the working directory.

You can find out the current working directory for your program using the `pwd` method from the `Dir` class.

Like this:

```
Dir.pwd
# /tmp
```

If you are getting errors when trying to load a file make sure you are on the correct path :)

Relative Loading

Another method which is often useful is `require_relative`.

This method lets you require files relative to the current directory without having to change the `$LOAD_PATH`.

Note that `require_relative` starts from the directory where the file is located and not from *your* current directory, this is a key difference in `require` vs `require_relative`.

The code for `require_relative` looks something like this:

```
def require_relative(path)
  # Get full path of current file
  current_path = File.realdirpath(__dir__)

  # Combine path with our file name
  full_path = File.expand_path(path, current_path)

  # Call normal require
  require(full_path)
end
```

If you are curious, here is a link to the actual code in Rubinius: https://github.com/rubinius/rubinius/blob/dbd33a6dadd407041fcafa6651986ae

When you are building a gem you should be able to just use `require`, as long as all your code is under the `/lib` directory. This works because gems always add `<gem_path>/lib` to the `$LOAD_PATH`.

Here is the code responsible for that:

```
lib = File.expand_path('../lib', __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
```

You can find that code on the top of every `.gemspec` file, which is loaded before the rest of the gem files.

The gemspec is not loaded while testing, but if you use a `spec_helper` file, `rspec` will add `lib/` and `spec/` to the `$LOAD_PATH` automatically.

Load vs Require

The `load` method is only really useful on irb/pry to reload some file you are working on.

The difference with `require` is that `load` will not check `$LOADED_FEATURES` to see if a file has already been loaded.

What this means is that if you call `load` 10 times for the same file, that file will be loaded 10 times. With `require` it will only be loaded once.

Another difference:

With `load` you need to provide the file extension, but `require` will append `.rb` as the extension, it will also try `.so` in Unix systems and `.dll` in Windows systems.

Regular Expressions

Contents

Using Regular Expressions

Regular expressions enable you to search for specific patterns inside strings.

They are used to find and validate data.

For example, think about an email address. With a regular expression **you can define what a valid email address looks like** & compare any string against that expression to find out if it's valid or not.

Regular expressions are defined between two forward slashes, to differentiate them from other language syntax. The most simple expressions match a word or even a single letter.

For example:

```
# Find the word 'like'
"Do you like cats?" =~ /like/
```

If there is a match, this code will return the index of the first letter in the word. If there is no match, then this code will return `nil`.

> Note: Since this case is pretty simple, we could have used the include? method without a regex.

Character Classes

A character class lets you define either a range or a list of characters to match. For example, `[aeiou]` matches any vowel.

Example: Does the string *contain* a vowel?

```
def contains_vowel(str)
  str =~ /[aeiou]/
end

contains_vowel("test") # returns 1
contains_vowel("sky")  # returns nil
```

This will not take into account the *amount* of characters, we will see how to do that later, in the "Modifiers" section.

Character Ranges

We can use ranges to match multiple letters or numbers without having to type them all out.

A range like `[2-5]` is equivalent to `[2345]`.

Some useful ranges:

- **[0-9]** matches any number from 0 to 9
- **[a-z]** matches any letter from a to z (no caps)
- **[^a-z]** matches the opposite of this range, in this case anything that's not a letter in the a-z range

---

Example: Does this string contain any numbers?

```
def contains_number(str)
  str =~ /[0-9]/
end

contains_number("The year is 2015")  # returns 12
contains_number("The cat is black")  # returns nil
```

> **Remember**: the return value when using `=~` is either the string index or nil

There is a nice shorthand syntax for specifying character ranges:

- **w** is equivalent to **[0-9a-zA-Z-]**
- **d** is the same as **[0-9]**
- **s** matches **spaces**

There is also the negative form of these:

- **W** anything that's not in **[0-9a-zA-Z-]**
- **D** anything that's **not a number**
- **S** anything that's **not a space**

Using a Wildcard

The dot character `.` matches anything but new lines.

If you need to use a literal `.` then you will have to escape it. Escaping a special character removes its special meaning.

Example:

```
# If you don't escape then the letter will match
"5a5".match(/\d.\d/)

# In this case only a literal dot matches
"5a5".match(/\d\.\d/) # nil
"5.5".match(/\d\.\d/) # match
```

Modifiers

Up until now we have only been able to match a single character at a time. To match multiple characters we can use pattern modifiers.

Modifier	Description
+	1 or more
*	0 or more
?	0 or 1
{3,5}	between 3 and 5

We can combine everything we learned so far to create more complex regular expressions.

Example: Does this look like an IP address?

```
# Note that this will also match some invalid IP address
# like 999.999.999.999, but in this case we just care about the format.

def ip_address?(str)
  # We use !! to convert the return value to a boolean
  !!(str =~ /^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/)
end

ip_address?("192.168.1.1")  # returns true
ip_address?("0000.0000")    # returns false
```

Exact Matching

If you need exact matches you will need another type of modifier. Let's see an example so you can see what I'm talking about.

Example: Find if a string is exactly five letters long.

This code will match because it has at more than five, but it's not what we want.

```
"Regexp are cool".match /\w{5}/
```

Instead, we will use the 'beginning of line' and 'end of line' modifiers:

```
"Regexp are cool".match /^\w{5}$/
```

This time it won't match because of the extra modifiers.

I know this is a rather contrived example, because we could just have used the `size` method to find the length, but I think it gets the idea across.

Here's another example where we want to know if a string is composed exclusively by numbers.

```
"123abc".match /^\d+$/
"123".match /^\d+$/
```

If you take out the modifiers (`^` and `$`) then both strings would match, but we don't want the first string to match because it contains both letters AND numbers.

Start of String vs New Line

If you want to match strictly at the start of a string, and not just at the start of a new line (after a `\n`), you need to use `\A` and `\Z` instead of `^` and `$`.

In this example you can see the difference:

```
"foo\nbar".match(/^bar/)  # <MatchData "bar">
"foo\nbar".match(/\Abar/) # nil
```

Capture Groups

With capture groups, you can capture part of a match and reuse it later. To capture a match, you enclose in parenthesis the part you want to capture.

Example: Parsing a log file

```
Line       = Struct.new(:time, :type, :msg)
LOG_FORMAT = /(\d{2}:\d{2}) (\w+) (.*)/

def parse_line(line)
  line.match(LOG_FORMAT) { |m| Line.new(*m.captures) }
end

parse_line("12:41 INFO User has logged in.")
# This produces objects like this:
# <struct Line time="12:41", type="INFO", msg="User has logged in.">
```

In this example, I'm using `match` instead of `=~`.

The `match` method returns a `MatchData` object if there is a match, `nil` otherwise. MatchData has many useful methods, check out the documentation!

You can access the captured data using the `captures` method or treating the `MatchData` object like an array, the zero index will have the full match, and the following indexes will have the matched groups.

You can also have non-capturing groups.

They will let you group expressions together without a performance penalty. You may also find named groups useful for making complex expressions easier to read.

Syntax	Description
(?:...)	Non-capturing group
(?<foo>...)	Named group

Example: Using Named Groups

```
m = "David 30".match /(?<name>\w+) (?<age>\d+)/
m[:age]
# => "30"
m[:name]
# => "David"
```

Exercise: Create a regular expression that can extract the name, the age and the country from the following string.

The string is : "name: John, age: 20, country: USA" .

Back-Referencing

When you use capture groups there is something very interesting you can do: substitution.

This is useful if you text editor supports regular expressions (vim, sublime and atom can do that, for example) or when using the `gsub` method in Ruby.

Let's say we want to wrap every word in a string with a dash before and after.

To do this using a regex, we can capture every word using `(\w+)` and then back-reference the captured word.

```
"regex are great".gsub(/(\w+)/, '-\1-')
```

> **Note**: The group number needs to be in single quotes for this to work.

This code will give us `-regex- -are- -great-` as the output. The `\1` refers to the first capture group. If we had three capture groups then we could use `\2` and `\3` to reference those.

Now here comes a slightly confusing part: in some places you use `\1` and others `$1` for back-referencing. In Atom's "Find and Replace" feature you want to use `$1`, but in vim you need `\1`.

Note: In Ruby $1 is a global variable. All the global variables starting with a number are special. They can't be directly assigned a value and they are "method-local".

Look ahead / Look behind

This is a more advanced technique that might not be available in all regex implementations.

It is available in Ruby, so let's take advantage of it.

Look ahead lets you peek and see if there is a specific match before or after.

Here's a table:

Name	Description
(?=pat)	Positive lookahead
(?<=pat)	Positive lookbehind
(?!pat)	Negative lookahead
(?<!pat)	Negative lookbehind

Example: is there a number preceded by at least one letter?

```
def number_after_word?(str)
  str =~ /(?<=\w) (\d+)/
end

number_after_word?("Grade 99")
```

Putting It All Together

Regular expressions can be used with many Ruby methods.

- .split
- .scan
- .gsub
- and many more...

Example: Get all words from a string using `scan`

```
"this is some string".scan(/\w+/)
# => ["this", "is", "some", "string"]
```

Example: Capitalize all words in a string

```
str.gsub(/\w+/, &:capitalize)
```

Example: Split a camelCasing string

```
"camelCasingTestFooBar".split(/(?=[A-Z])/)
```

Match vs Scan

The `match` method will only look for the first occurrence of the pattern you want to find, and it will always return either a `MatchData` object or `nil`.

On the other hand, if you want to 'scan' a string for multiple occurrences of a pattern, then you want to use the `scan` method.

This method returns an array with the results, if there are no matches it will return an empty array.

Conclusion

Regular expressions are amazing, but sometimes they can be a bit tricky. Using a tool like rubular.com can help you build your regexp in a more interactive way. Now go ahead and write your own!

> If you want to learn how Regular Expressions work under the hood, you should watch this video.

Modules

How to Use Modules in Ruby

The main purpose of modules is to group together methods and constants.

One difference with classes is that **you can't create instances of a module**.

Modules can also be used for name-spacing.

Here is an example module:

```
module Numbers
  PI = 3.141592
end
```

There are a few different ways you can use modules:

- You can use them as a way to collect related utility methods & constants in one place.
- You can include the instance methods & constants from a module into a class to share functionality.
- You can include all the methods in a module as class-level methods.

Examples

Example 1: Using a constant

```
puts Numbers::PI
# => 3.141592
```

Notice the `::` syntax which is used to reach into nested modules and classes.

Using `::` can also be useful when you need to access the global namespace. To do that you need to prefix it, like this: `::Array`. This can be useful when you don't have access to `Object`'s constants, for example inside a `BasicObject` sub-class.

Example 2: Using a method

```
module Numbers
  def self.double(number)
    number * 2
  end
end

puts Numbers.double(30)
# => 60
```

In this case you don't need the `::` syntax because we are working with a class-level method (modules have a `singleton_class` too).

Example 3: Using `include`

```
class Calculator
  include Numbers # Include module in inheritance chain
end

# Methods will be looked up in this order
Calculator.ancestors
```

Including a module will give the class access to all the constants and instance methods as if they where part of the class itself.

Included methods have higher priority than superclass methods.

We can see that in the following example:

```
module Life
end

class Animal
end

class Tiger < Animal
  include Life
end

# The module (Life) is earlier in the list than
# the super-class (Animal)

Tiger.ancestors
=> [Tiger, Life, Animal, Object, Kernel, BasicObject]
```

The alternative version of `include` is `prepend` which has a similar effect.

The difference is that `prepend` will put the module earlier on the `ancestors` list...

...this means that methods from a 'prepended' module will take precedence over any other methods.

Here is what the `ancestors` look like if we `prepend` the `Life` module:

```
Tiger.ancestors
=> [Life, Tiger, Animal, Object, Kernel, BasicObject]
```

Notice how `Life` is the first thing in the list.

Example 4: Using `extend`

```
module ClassMethods
  def add_product
  end
end

class Store
  # Add methods from the module to the singleton class of this object
  extend ClassMethods
end

Store.singleton_methods
# => [:add_product]
```

The `extend` keyword allows you to add the methods defined inside a module as if they were class-level methods of another class.

In fact, what `extend` does is equivalent to this:

```
Store.singleton_class.instance_eval { include(ClassMethods) }
```

Example 5: Using `extend self`

```ruby
module Formatter
  extend self

  def wrap_with(symbol, str)
    "#{symbol * 4} #{str} #{symbol * 4}"
  end
end

puts Formatter.wrap_with("*", "hello")
```

This defines all the methods in the module as class-level methods, which lets you call the methods directly from the module, while retaining the ability to use these methods for inclusion.

In other words: the methods are defined both as instance methods and as singleton methods.

Enumerable

Contents

What is Enumerable

Enumerable is a module that is included in `Array`, `Hash`, `Range` and a few other classes.

This module is composed of a number of very powerful iteration methods.

Like the `each` method.

In this chapter you will learn about some of the most interesting `Enumerable` methods, then you will learn how you can include `Enumerable` in your own classes to make them more powerful!

You will also reimplement some `Enumerable` methods on your own to understand how they work & then we will study more advanced features like lazy enumerators.

Let's do this!

Enumerable Examples

Use `inject` when you want to apply some operation on every element of the array and need some kind of accumulator to hold the result.

For example, if you want to add up all the numbers in an array, you need a variable to hold the on-going total.

This variable will be the accumulator.

The starting value for the accumulator can be:

- The first element yielded by each
- The argument passed to inject (if any)

Example:

```
[10,20,30].inject { |total, num| total + num }
# 60
```

You can also use the short form:

```
[10,20,30].inject(&:+)
# 60
```

Here is another `inject` example which prints the values at every iteration.

```ruby
[10,20,30,40,50].inject do |total, num|
  puts "Total: #{total}\t" + "Number: #{num}"
  total + num
end

# Total: 10   Number: 20
# Total: 30   Number: 30
# Total: 60   Number: 40
# Total: 100  Number: 50
```

> **Note**: I used the \t special character here for better formatting, but it has nothing to do with inject itself :)

You can also use `inject` with a `Hash` object (because `Hash` includes `Enumerable`).

But notice that your 2nd parameter will be an array with two elements:

- The hash key on index 0
- The hash value on index 1

Like this one:

```ruby
[:bacon, 10]
```

In the following example this will be represented by the `values` variable.

The accumulator variable (`total`) will still be the 1st parameter.

Example:

```
prices = {
  bacon: 10,
  coconut: 20
}

total_price =
  prices.inject(0) { |total, values| total + values[1] }

puts total_price
# 30
```

Bonus Trick:

```
prices.inject(0) { |total, (key, value)| total + value }
```

This is called "deconstruction".

It will separate the key & value into their own variables.

What's the benefit?

This makes it more clear that you're working with two separate variables & allows you to give them a name, instead of having to guess what values are on what array indexes.

Using The All? Method

Use `all?` when you want to check if all the items match a condition, you can also use `none?` for the opposite effect (zero items match the condition). Both methods return either `true` or `false`.

> **Definition (predicate method)**: A predicate method is a method that always returns a boolean value (true or false). In Ruby, it's common for predicate methods to end with a question mark (?).

Example: Is the length of all the words equal to 3?

```
%w(cat dog cow).all? { |word| word.length == 3 }

# true
```

You can also use the `any?` method.

This methods returns `true` if at least one of the items matches the condition.

How to Use The Reject Method

Use `reject` (which is the opposite of `select`) when you want all the items that don't match a certain condition.

```
numbers = [1,2,3,4,5]
numbers.reject(&:odd?)

# => [2,4]
```

How to Use the Partition Method

Use `partition` when you want to split your items in two groups:

- one that matches the condition
- another that doesn't

This example returns two arrays, the first one has the even numbers and the second one has the odd numbers.

```
numbers = [1,2,3,4,5]
numbers.partition(&:even?)

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

How to Use the Sort By Method

Use `sort_by` when you need to sort a collection in a specific way (different than the default sort).

For example, if you want to sort an array of numbers by their last digit you can do something like this:

```
[48,75,19,12,21].sort_by { |n| n.to_s[-1] }

# => [21, 12, 75, 48, 19]
```

The `n.to_s[-1]` part can be a bit confusing, what this is saying is: "convert a number to a string and take the last digit".

How to Use the Each Cons Method

Use `each_cons` when you need sequential sub-arrays.

This can be used for finding n-grams or to check if an array contains a sequential list of numbers when combined with `all?`.

Example:

```
numbers = [3,6,5,4,7]

numbers.sort.each_cons(2).all? { |x, y| x == y - 1 }
```

Including Enumerable

Since `Enumerable` is a module you can include it in your own classes so they can have methods like `select`, `map` or `inject` available to them.

To make this work you will need to do one thing first:

Implement the `each` method in your class.

This is important because all the `Enumerable` methods are based on `each`.

Here is an example:

```ruby
class UserRepository
  extend Enumerable

  def self.all
    [
      User.new('test', 18, 'Spain'),
      User.new('roberto', 24, 'Cuba'),
      User.new('john', 30, 'France')
    ]
  end

  def self.each
    all.each { |user| yield user }
  end
end

User = Struct.new(:name, :age, :country)

UserRepository.select { |user| user.age > 20 }
UserRepository.map(&:name)
```

In this example I used `extend` to have the `Enumerable` methods available at the class-level.

Yu can see `select` & `map` in action on the last two lines.

How Does Each Work?

If your data is not based on another object that already implements `each`... or if you just want to learn what `each` is really doing, then you are going to like this example.

Take a look:

```
class FileCabinet
  def initialize
    @documents = [1,2,3]
  end

  def size
    @documents.size
  end

  def each
    count = 0

    while count < size
      yield @documents[count]
      count += 1
    end
  end
end

FileCabinet.new.each { |doc| puts doc }
```

This is the secret behind `each`:

- a while loop
- yield
- a counter variable

> **Note**: Review the 'Procs & Lambdas' lesson if you are not sure how yield works.

Implementing Map

Let's implement some of the methods provided by `Enumerable` so you can learn more about how they work.

We are going to start with this class:

```
class Library
  attr_accessor :books

  def initialize
    @books = [
      Book.new('Eloquent Ruby',  'Russ Olsen', 448),
      Book.new('Confident Ruby', 'Avdi Grimm', 296)
    ]
  end
end

Book = Struct.new(:title, :author, :pages)
```

Let's implement the `map` method.

These are the steps you need:

1. Create a new array
2. Iterate over the data using `each`
3. Call yield to process every element (which gives you the modified element)
4. Save the output from yield into the results array
5. Return the results

Example:

```
def map
  results = []

  @books.each { |book| results << yield(book) }

  results
end
```

Now we should be able to use this method like a normal `map`.

```
library = Library.new
library.map(&:title)
```

> **Exercise**: Using the Library class as a starting point, implement the following Enumerable methods: select, all?, count.

Creating a Custom Enumerable Method

You may also want to implement your own `Enumerable` methods.

For example, you may want to have an `each_name` method.

This is how you can extend the `UserRepository` example to implement this idea:

```
def self.each_name
  return to_enum(:each_name) unless block_given?
  all.each { |user| yield user.name }
end
```

You should pay attention to the `to_enum(:each_name)` method, this allows you to use the `each_name` method without a block.

Enumerator Objects

When you call `to_enum` you get an `Enumerator` object.

This object holds a reference to the original object it was called from and also the method name that was used.

This means that you can call `Enumerable` methods on this new object as if it was your real object. But if you call `each` it will use the method that was passed as an argument to `to_enum`.

I know, this can be a bit confusing, but please bear with me.

Let's study an example:

```
enum = [1,2,3,4].select
enum.class
# Enumerator
```

In this example we generate an `Enumerator` object by calling `select` without a block.

Let's see what happens when we try to interact with this new object.

```
enum.each { |u| u > 2 }
# [3,4]
```

You can see that using `each` acts as if we called `select` directly.

This happens because the `Enumerator` object remembers what method it was created from, and it calls that method for us.

This all makes sense if you think about it.

Remember that all the `Enumerable` methods are implemented by using `each` in one way or another. In fact, this is the reason why you are able to chain `Enumerable` methods!

The `Enumerator` class also has something to offer of its own, the `next` method.

```
enum.next
# 1
enum.next
# 2
enum.next
# 3
```

Using this method we can ask for new elements when we need them.

After we request the last one an `StopIteration` exception will be raised.

```
enum.next
# StopIteration: iteration reached an end
```

There is also a `rewind` method that lets you start over.

If you want to go over all the possible values using `next`, you can use the `loop` method, which creates an infinite loop.

Example:

```
loop { puts enum.next }
```

The `loop` will stop once it finds an `StopIteration` exception, without actually raising the exception.

Lazy Enumerators

When you use an `Enumerable` method like `map` you will get a new array with the result.

Sometimes you want to do something else with this array...

Like figuring out if all the elements meet a condition using the `all?` method.

But what if your original array is really big?

Using `map` + `all?` will process the **entire array** at least once for `map`, and most likely twice with `all?`.

> **Note**: The all? method returns false as soon as it finds an element that doesn't match the condition.

But there is a way around that by using Ruby's **Lazy Enumerators**.

A feature that was introduced in Ruby 2.0.

Here is an example:

```
big_array = Array(1..10_000)
big_array.lazy.map { |n| n ** 2 }.all? { |n| n < 100 }
```

This part would normally return an array:

```
big_array.map { |n| n ** 2 }
```

But if you call `lazy` before calling `map` you get an `Enumerator::Lazy` object.

This allows you to do a partial `map` and only get the amount of results you need.

In this example, the `all?` method will ask `map` for one item at a time and stop as soon one of the elements doesn't match the condition, which will save your program a lot of work!

One way to actually see this in action is by adding a `p` to the example I showed you before.

> **Note**: In case you are not familiar with `p`, it's a method provided by the Kernel module which is equivalent to doing `puts object.inspect`. It's very useful for debugging!

```
big_array = Array(1..10_000)
big_array.lazy.map { |n| p n; n ** 2 }.all? { |n| n < 100 }
```

If you try this code in `irb` you will notice that only 1 through 10 are printed, but if you take away the `lazy` then every number is printed.

The reason this happens is that a lazy `Enumerator` will process only as many elements as necessary & not more.

That's why we call them lazy :)

Here is another example:

```
require 'prime'

# Never ends
(1..Float::INFINITY).select(&:prime?).first(10)

# Very fast
(1..Float::INFINITY).lazy.select(&:prime?).first(10)
```

In this example I want to get the first 10 prime numbers.

I'm using an infinite range so I can ask for as many prime numbers as I want.

Notice how the non-lazy version will never finish because **it's trying to find all the prime numbers from 0 to infinity** before giving us the 10 we asked for. But the lazy version will do the minimal amount of work necessary to get what we want.

Generators

One more thing you can do with `Enumerator` is to create a 'generator' object.

Let's say that you have an infinite sequence of numbers like fibonacci or prime numbers.

Then you can do something like this:

```ruby
my_generator =
Enumerator.new do |yielder|
  # Set starting values
  first  = 0
  second = 1

  # Yield the first number in the sequence
  yielder.yield(1)

  loop do
    # Calculate next fibonacci number
    fib = first + second

    # Yield the result
    yielder.yield(fib)

    # Advance the sequence
    first, second = second, fib
  end
end

p my_generator.take(10)
p my_generator.take(5)

5.times { p my_generator.next }
```

Notice how `next` and `take` don't interfere. Regular `Enumerable` methods will just start over, while `next` will keep track of where you are.

By the way, the `yielder.yield` method here has nothing to do with your regular `yield` keyword, so don't confuse them!

Most of the time what this is really calling is the Fiber.yield method.

Summary

Enumerable is the most powerful & important Ruby module because it makes your life easier when working with a collection of objects (Arrays, Hashes, Ranges...).

If you take just one thing out of this chapter, remember the top 3 `Enumerable` methods:

- map (TRANSFORM)
- select (FILTER)
- inject (COMBINE)

I gave you a keyword to remember for each of these methods. That keyword represents the main use for each method.

Using Sinatra

Contents

Introduction to Sinatra

Sinatra is a very simple and powerful web framework. In this chapter you will learn how it works and how to build simple apps using Sinatra.

If you haven't started using Rails yet, this is a great way to get started writing & understanding web applications.

If you already use Rails, Sinatra is great to use when you just need a small web application without all the extra features that come with Rails.

Onward!

Getting Started with Sinatra

Let's get started with the simplest possible Sinatra app. The first thing you need to do is to map a path to an action.

For example:

```
require 'sinatra'

get '/' do
  "Hello World!"
end
```

We have a few components here:

1. We require the Sinatra gem
2. We call the `get` method, which creates a new route (URL end point)
3. We tell Sinatra that when someone loads `/` (the base URL, like http://example.com/) that it should return the words "Hello World!" to the browser

Now, to see this in action, you need to start this script normally with a `ruby simple.rb` and visit `http://localhost:4567` with your browser.

This will output the string "Hello World!" into your browser. That's cool but not that useful, we need to be able to take some form of input from the user to make it more interesting.

For example, this will save anything after `/` under the `params` hash with the `:name` key:

```
get '/:name' do
end
```

Note: When you make changes to your Sinatra code you will have to restart you app. You can use the shotgun gem for auto-reloading. With shotgun installed, start your app with `shotgun app.rb` instead of `ruby app.rb`.

Base64 API

Let's build a base64 encoding API as an example:

```
require 'sinatra'
require 'base64'

get '/encode/:string' do
  Base64.strict_encode64(params[:string])
end

get '/decode/:string' do
  Base64.strict_decode64(params[:string])
end
```

Notice how you can access parameters using `params`, which is just a regular Ruby hash.

Now you can point your browser to `http://localhost:4567/encode/hello` to get the base64 encoded version of the word 'hello'. Alternatively, you can use a tool like `curl` to interact with your app from the command-line.

```
curl http://localhost:4567/encode/hello
```

Exercise: Play around with this! Try to change the code so that your app returns the length of the parameter (params[:string]).

Using the Post Method

Besides the `get` method you can also use the `post` method to create a new route.

You should use `post` when you are sending a request to the server instead of a query (for example: create a new user account).

Here is an example:

```
post '/create_user' do
  "User #{params[:name]} created!"
end
```

You could setup a form to test your `post` method, but it's faster to just use `curl`:

```
curl -d 'name=test' localhost:4567/create_user
```

Using Views

You will want to start using views when you need to render more than a simple string.

Sinatra has built-in support for views and you can use a templating engine like erb or haml.

Views must be located under the `/views` directory.

Example: rendering a view using haml

```
require 'sinatra'
require 'haml'

get '/' do
  # This will render `views/index.haml`
  haml :index
end
```

Here is a one line `Ã¬ndex.haml` you can use for testing:

```
%h1 Sinatra is awesome
```

You can pass data to the view via instance variables.

Sinatra + ActiveRecord

The next step is being able to interact with the database. The good news is that you can use `ActiveRecord` so you don't have to mess around with raw SQL.

First, you need to require the `active_record` gem and establish a connection:

```
require 'active_record'

ActiveRecord::Base.establish_connection(
    adapter: "sqlite3",
    database: "/tmp/sinatra.db"
)
```

You also need to define a model for every table you want to access. Remember that table names are `plural` and model names are `singular`.

```
class User < ActiveRecord::Base
end
```

Connections are not automatically closed when using `ActiveRecord` with Sinatra. We can add this snippet of code to do that for us:

```
after do
  ActiveRecord::Base.clear_active_connections!
end
```

Let's Write a Voting App

To bring everything together let's write a simple voting app using Sinatra.

Sinatra doesn't have a built-in migration system so I created a table on my database using `sqlite3 /tmp/sinatra.db`

This is what the schema looks like:

```
Column |  Type   |           Modifiers
-------+---------+-----------------------------------
id     | integer | not null default
count  | integer | default 0
name   | text    |
Indexes:
   "votes_pkey" PRIMARY KEY, btree (id)
```

> **Note**: It's important that your tables have an `id` primary key or you are going to run into issues. Also if you are not familiar with the schema output above don't worry about that, just copy & paste the sql code below in your sqlite3 prompt to create the table.

This is the `sql` query used to create that table:

```
CREATE TABLE votes (
  name text,
  count integer DEFAULT 0,
  id integer NOT NULL
);
```

And here is the model:

```
class Vote < ActiveRecord::Base
end
```

This is the index action, which gets all the votes from the database and renders the index view:

```
get '/' do
  @votes = Vote.all.order(:id)
  haml :index
end
```

And this is the `post` action that will register the votes:

```
post '/vote' do
  vote_name = params[:name]

  vote = Vote.find_or_create_by(name: vote_name)

  vote.count += 1
  vote.save!

  "Count updated!"
end
```

Now we can visit the index to see the results and use `curl` to cast our vote:

```
curl -d 'name=test' localhost:4567/vote
```

Since we are using `find_or_create_by` a new entry will be created for things with 0 votes.

Conclusion

Sinatra is a great framework that can help you put together a quick web app without the overhead of everything Rails has to offer. Now it's your turn, go and create something nice using Sinatra and share it with the world :)

You can find the code for the voting app here: https://github.com/matugm/sinatra-voting

Another version which uses Sequel instead of AR can be found here: https://github.com/matugm/sinatra-voting/commit/5424a57b60b817a88ae7998d24ac27b362601a25

Exercise 1: Take a look the Sinatra documentation -> http://www.sinatrarb.com/intro.html

Exercise 2: The sidekiq project uses Sinatra for its web interface. Explore the code to get a feel for a real-world Sinatra use case.

Exercise 3: Modify the base64 example to return json instead of plain text. You can use the sinatra/json module to help you with that.

Performance

Content

- Benchmarking
- Using ruby-prof
- Optimization Example
- More profiling

Profiling Ruby Applications

If your application feels slow you can't just start making random changes and hope for the best.

You need to **use tools to measure the performance of your code** to learn where to focus your efforts.

There are two kinds of measurements you can do:

- Benchmarking
- Profiling

Benchmarking is the comparison between two or more things that get the same job done (for example, Apache vs Nginx).

Profiling is about finding the bottlenecks, the things that are slow in a system.

The resulting data from running a performance test will become your baseline.

With this baseline you will be able to tell if your changes are making the app faster or not.

Benchmarking

The simplest way to benchmark some Ruby code is by saving the current time, executing the code, and then comparing the time it took to complete.

Here is an example:

```
def expensive_method
  50000.times { |n| n * n }
end

def measure_time
  before = Time.now
  yield
  puts Time.now - before
end

measure_time { expensive_method }

> 0.639588159
```

Another thing you may want to do is to compare similar functions, for example is `=~` faster than `.match`?

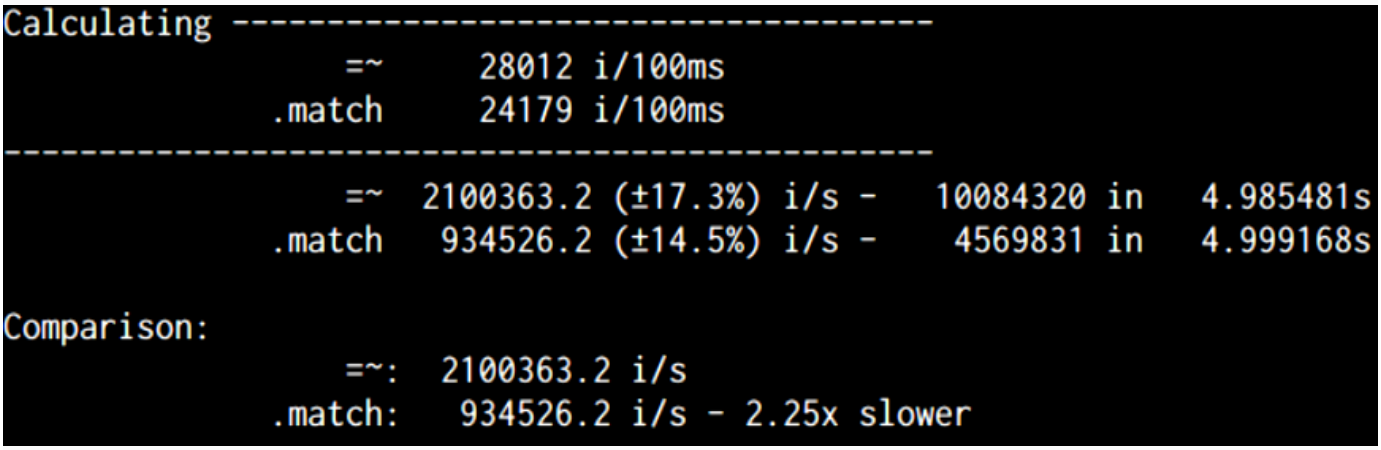
You can find out using the `benchmark/ips` (iterations per second) gem.

```
gem install benchmark-ips
```

Once you have installed the gem you can write a benchmarking script like this one:

```
require 'benchmark/ips'

str = "test"
Benchmark.ips do |x|
  x.report("=~") { str =~ /test/ }
  x.report(".match") { str.match /test/ }

  x.compare!
end
```

When you run this code you will get an output similar to the following image.



This shows that `.match` is the inferior method in terms of performance.

Not too surprising...

Knowing that `.match` has to create new objects, while `=~` just returns `nil` or an index.

Using ruby-prof

Instead of benchmarking a few methods against each other you can profile everything. Ruby's built-in profiler doesn't produce the best results so you are going to need the `ruby-prof` gem.

```
gem install ruby-prof
```

You can run it like this & it will show you every method that takes more than 5% of the total run-time.

```
ruby-prof -m 5 <script.rb>
```

Output:

```
Total: 0.004784
Sort by: self_time

 %self      total      self      wait     child     calls  name
 38.04      0.003     0.002     0.000     0.001        40  Range#each
 12.85      0.001     0.001     0.000     0.000       261  Fixnum#==
 11.03      0.001     0.001     0.000     0.000       223  Fixnum#%
  9.84      0.005     0.000     0.000     0.004         1  Integer#times
  7.83      0.000     0.000     0.000     0.000        28  IO#write
```

The meaning of the columns goes like this :

Column	Description
%self	% of the total time for this method
total	time spent by this method + its children
self	time spent by this method
wait	time spent waiting on an external resource (network, disk)
children	time spent on calling other methods
calls	number of times this method was called
name	method name

You can use the `qcachegrind` program for another way to visualize the profiler data.

With `ruby-prof` & the `call_tree` option you can get `cachegrind` files that can be used for analysis.

Here's how:

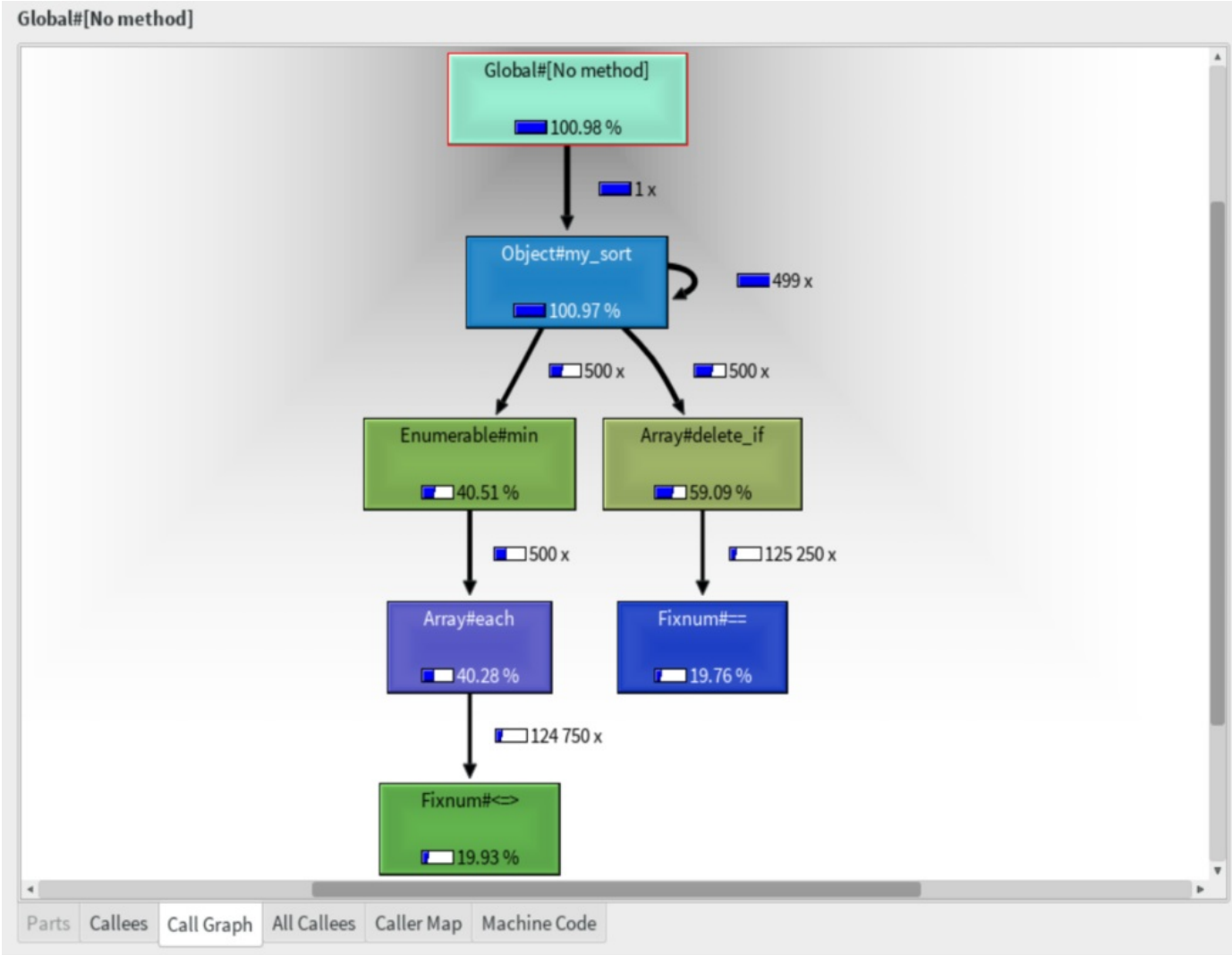
```
ruby-prof -p call_tree -f /tmp/test.grind example.rb
```

The parameters are:

- -p for output format
- -f for the output file name.

What I'm saying here is:

"Run `example.rb` with profiling enabled, save the output to `/tmp/test.grind` and use `call_tree` as the output format".



In this image you can see that the method `my_sort` is called 1 time and it calls itself 499 times, which tells you that this is a recursive method. This would have been really hard to see on the regular `ruby-prof` output.

> **Definition**: A recursive method is a method that calls itself.

You can also see how `my_sort` calls **two other methods**:

- Enumerable#min
- Array#delete_if

Optimization Example

Now let's see how you can optimize some Ruby code using profiling. We are going to work with this simple sorting function:

```
def my_sort(nums, sorted = [])
  min = nums.min

  sorted << min
  nums.delete_if { |n| n == min }

  if nums.size > 0
    my_sort(nums, sorted)
  else
    sorted
  end
end

my_sort (1..500).to_a.shuffle
```

> **Note**: I used this same code for the profiling examples in the last section.

You can load the results on `qcachegrind` and then you want to sort by the `self` column. This is the total time a specific function took to run.

You can see that `Array#delete_if` stands out.

Now:

Turn your attention to the call graph on the right, it will help you understand the situation.

The `delete_if` method calls `Fixnum#==` a lot of times, which means we are going through our whole array to find the number to delete.

That's not very efficient.

Let's see if we can improve this...

A quick look at the Ruby documentation on the `Array` class & we can find the `delete` method.

We can try it & see if it speeds things up.

> **Note**: Remember that you can find the Ruby documentation here. It's very useful to have it on your bookmarks. If you prefer offline documentation try Zeal.

But if we keep looking, after the `delete` method we will find the `delete_at` method.

It has the following description:

> "Deletes the element at the specified index, returning that element, or nil if the index is out of range."

Sounds like what we need, but how do we get the index?

We could use the `index` method, but that would defeat the purpose since it has to search the array to find the index.

What we really need is a custom `min` function that will return the index in addition to the number.

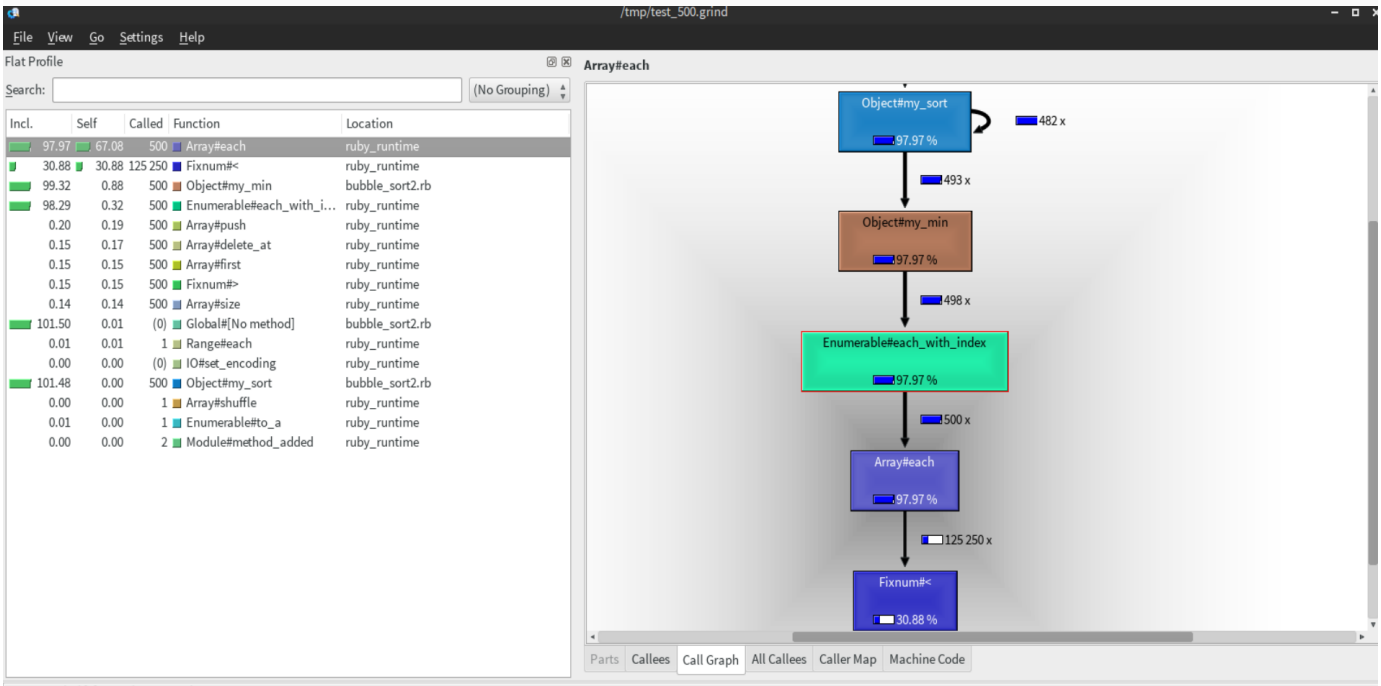
This is the custom `min` I wrote:

```
def my_min(nums)
  min = nums.first
  index = 0

   nums.each_with_index do |n, idx|
     if n < min
       min   = n
       index = idx
     end
   end

  [min, index]
end
```

Here is the sorting function which uses this new `min` function:

```ruby
def my_sort(nums, sorted = [])

  min, index = my_min(nums)

  sorted.push(min)
  nums.delete_at(index)

  if nums.size > 0
    my_sort nums, sorted
  else
    sorted
  end
end

my_sort (1..500).to_a.shuffle
```

If we profile the modified code we will get the following results:



As you can see on the call graph, we managed to cut off the whole `Array#delete_if` branch, which saves us a lot of work.

The sorting function is now much faster (about 40%)!

> **Note**: Ruby's built-in sort will always be faster than any sort you can come up because it's implemented at the C level.

More Profiling Tools

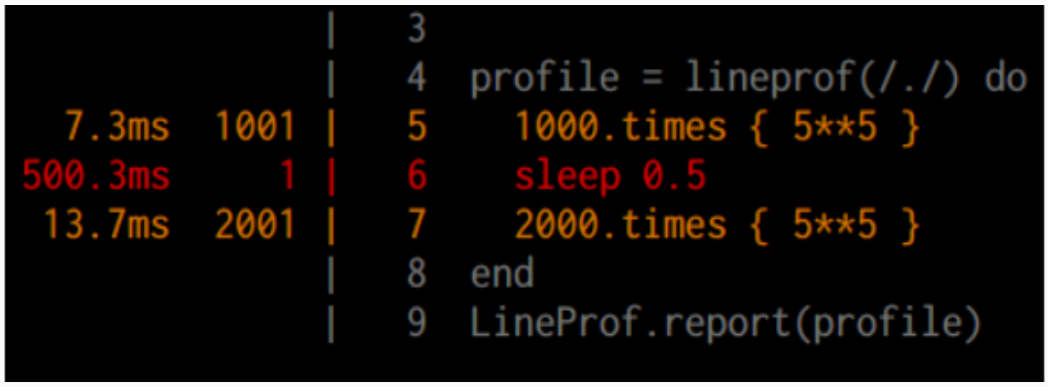
It's always good to have more tools in your toolbox.

Take a look at the following links:

- **rack-mini-profiler**https://github.com/MiniProfiler/rack-mini-profiler
- **memory_profiler**https://github.com/SamSaffron/memory_profiler
- **stackprof** https://github.com/tmm1/stackprof

You may also want to take a look at rblineprof which focuses on profiling individual lines of code instead of methods.

Here is an example of rblineprof. Using the rblineprof-report gem for nice output.



Stackprof can also do line profiling & show you exactly what lines of code are the slowest ones when you use the `--method` option.

Comparing Objects

What happens when you want to sort or compare two objects?

In the case of numbers or strings this is a well-defined operation.

Example:

```
10 > 5
# true

"abc" == "a"
# false
```

But how does this work and how can you use this in your own classes? The key is in the `Comparable` module and the `<=>` method.

This method is used to determine if one object is lesser, equal or greater than another object. These 3 different results are represented in the form of a number.

Value	Meaning
-1	Less than
0	Equals
1	Greater than

> **Note**: In the Ruby world, the `<=>` symbol is known as the 'spaceship' operator.

Example:

```ruby
class Product
  include Comparable

  # Use the price to decide if two products are equal
  def <=>(other)
    self.price <=> other.price
  end
end

p1 = Product.new(100)
p2 = Product.new(200)

p1 == p2
# false

p1 < p2
# true
```

As long as you have the `<=>` method defined in your class, the `Comparable` module gives you the following methods: `<`, `<=`, `==`, `>`, `>=` and `between?`.

This also makes your class more powerful because other methods (like `sort`) use `<=>` as a way to decide the order.

If a class doesn't implement the `==` method in one way or another, then it will be up to `BasicObject#==` to decide.

The `==` method on `BasicObject` compares the object's internal ids, which means that it will only return `true` when both objects are the same object.

That method looks something like this:

```
class BasicObject
  def ==(other)
    object_id == other.object_id
  end
end
```

Metaprogramming

Contents

Introduction

A lot of people seem confused about metaprogramming, and that's normal.

The reason is that metaprogramming is not a clearly defined concept. It's more like a set of things you can do to interact with your Ruby program in a different way that you normally would.

For example:

Metaprogramming lets you inspect & mess around with the internals of Ruby objects. Being able to inspect your objects in detail will help you improve your debugging skills.

> **Definition (meta)**: "A prefix added to the name of a subject that analyzes the original one but at a more abstract, higher level."

With metaprogramming you can also generate new code dynamically, like defining new methods during the execution of your program.

That's how methods like `attr_reader` or `attr_writer` work!

Metaprogramming is a very powerful technique that's it's easy to abuse. I think the following quote is very fitting & you should remember it when working with any metaprogramming methods.

> "With great power comes great responsibility."

Sending Messages

Using the `send` method you can call other methods without the typical dot notation.

This method takes two arguments:

- The name of the method you want to call (in string or symbol form)
- The arguments for that method

Here is an example:

```
# This is equivalent to: puts "Hello"
send(:puts, "Hello")
```

When using metaprogramming methods you'll often bypass visibility rules like `private` & `protected`. That is the case for `send`, but there is a `public_send` variation if you need it to respect private methods.

Here is an example so you can understand what I'm talking about:

```
class PrivateTesting
  private

  def internal_method
    "Only for internal use"
  end
end
```

Calling `internal_method` will raise an exception since it's declared as private.

```
pt = PrivateTesting.new
pt.internal_method
NoMethodError: private method `internal_method' called for #<PrivateTest
```

But if we use `send` it will just ignore the fact that the method is private and call it anyway.

```
pt.send(:internal_method)
# => "Only for internal use"
```

What is Self?

The special variable `self` holds a reference to the current object.

You're always working within an object in Ruby .

And `self` gives you access to that object.

Here's an example...

When you call a method like this:

```
puts 123
```

You're really doing this:

```
self.send(:puts, 123)
```

Here is how `self` changes in different contexts:

```ruby
# Top-level context, self is just an instance of Object
puts self

class FindingSelf
  # self is the class
  puts self

  def show_self
    # self is the object instance
    puts self
  end
end
```

If you try to call a private method using `self` you will get an error.

Private methods can only be called directly.

You can see this in action if you run the following in irb:

```ruby
puts "method call"
# "method call"

self.puts "method call"
# NoMethodError: private method `puts' called for main:Object
```

As a reminder, the `puts` method comes from the `Kernel` module, it's made private via the `module_function` method.

Take a look at the [Rubinius source code] for `Kernel`.

Class Introspection

The `ls` command in `pry` lets you find some information about a class, like the methods and instance variables it has.

But how does that work?

There a number of methods provided by Ruby that give you this information if you ask for it.

Let's take a look at the `instance_methods` method....

...like the name says it will list all the instance methods for this class, including those inherited from its parent classes.

Example:

```
String.instance_methods.size
# 172
```

Often we just want to see the methods that are directly defined by the class, to do that we can pass `false` to `instance_methods`.

Like this:

```
String.instance_methods(false).size
# 111
```

We can also get the private methods using `private_instance_methods`.

```
String.private_instance_methods(false).size
# 2
```

To get a list of all the methods inherited by a class, you can do this:

```
Array.instance_methods - Array.instance_methods(false)
```

Similarly you can also get the `instance_variables` for an instance of a class.

Take a look at the following example:

```
class Total
  def initialize
    @count = 2
  end
end

Total.new.instance_variables
# => [:@count]
```

It is also possible to use `respond_to?` to ask an object if it can respond to a certain method call.

Example:

```ruby
class A
  def say
  end
end

class B < A
end

p B.new.respond_to?(:say)

# => true
```

The Method Method (not a typo)

Ruby gives us a method called `method` that we can use for some interesting things. For example, we can find out where in the source code a method is defined.

Example (run this from a file instead of irb/pry):

```
class Animal
  def self.speak
  end
end

p Animal.method(:speak).source_location
```

When you call `method` you get back an instance of the `Method` class. The `source_location` method I used in this example belongs to that class.

> **Note**: There is also an `instance_method` method which returns an UnboundMethod object. It is unbound because it doesn't have a receiver (an instance of a class to operate in). As always, I encourage you to explore the differences using pry and the ls command.

There are two more interesting things you can do with this:

- You can use `call` to execute the method
- You can use `to_proc` to turn this method into a proc

The `to_proc` method lets you do something like this:

```
my_proc = method(:puts)

[5,10,20].each(&my_proc)
```

Another useful method available on `Method` objects is the `owner` method.

```
Array.instance_method(:sort_by).owner
# Enumerable
```

This can be very useful if you are trying to fix or understand some code.

Example:

```
Fixnum === 3 # true
3 === Fixnum # false
```

The key here is `===`, which is just a method, so this is what is really going on:

```
Fixnum.===(3)
3.===(Fixnum)
```

Knowing this we can use the `owner` method to discover where these methods are defined.

```
Fixnum.method(:===).owner # Module
```

And now we can look at the documentation for `Module#===` to understand that `Fixnum === 3` checks that `3` is of type `Fixnum`.

But `3 === Fixnum` checks that `Fixnum` has a value of `3`, which it doesn't :)

Dynamic Constants

It is also possible to create constants dynamically. The `const_get` and `const_set` methods exist for that purpose.

For example:

```
class Colors
  def self.define_constant_colors
    colors = {
      blue: '#1c0078',
      red:  '#d72525',
      green: '#15bf00'
    }

    colors.each { |color, code| const_set(color.upcase, code) }
  end
end

Colors.define_constant_colors
puts Colors::GREEN
```

The Eval Methods

You can execute Ruby code inside a string using the `eval` method.

This is very dangerous as it could allow a user of your application to break into your system, so be careful!

I would recommend avoiding using `eval` entirely...

But it's good to be aware of its danger in case you come across it in someone else's code.

Example:

```
eval "puts 5 + 5"
```

There is also two other variants to `eval`:

- instance_eval
- class_eval

These allow you to evaluate code in the context of a specific object instance or class.

Here is an example of using `instance_eval` to set an instance variable without an `attr_writer`.

You probably want to avoid this in real code because this breaks encapsulation.

```ruby
class Example
  def initialize
    @value = 10
  end
end

example = Example.new
example.instance_eval("@value = 20")
p example
```

Here is another example of `instance_eval`.

This example also introduces `define_singleton_method`, which can be used to define a class-level method at run-time.

```ruby
String.instance_eval do
  define_singleton_method(:hello) {
    puts "hello there"
  }
end

String.hello
```

> **Note**: If you need to define an instance method you can use define_method instead of define_singleton_method.

Method Missing

With `method_missing` you can capture method calls for methods that currently don't exist. This is how Rails 'dynamic finders' work (`find_by_name`, `find_by_age`, etc.).

Experiment: What do you think this code will print? Run it in pry and see what happens.

```
class Finder
  def method_missing(name, *args, &block)
    puts "The method #{name} was called but it's not defined in #{self.class}
  end
end

finder = Finder.new
finder.find_cool_stuff
```

Please be aware that using `method_missing` has an important performance cost! The main reason is that `method_missing` is the last thing Ruby looks for when trying to find a method in the inheritance chain.

Creating Methods

If you want to create a method in Ruby what would you do?

Probably something like this:

```
def the_method
  # method body...
end
```

But what if you want to create **new methods** while your program is running?

Well metaprogramming gives you that power.

I'm sure you are familiar with "accessor" methods: `attr_writer`, `attr_reader` & `attr_accessor`.

These are methods that create new methods for you. To save you time & effort.

For example...

```
attr_accessor :display
```

...will create two methods for you: `display` & `display=`.

The good news is that you can do the same :)

Here's an example:

```
class Foo
  define_method :display { instance_variable_get("@display") }
end
```

The key here is the `define_method` method. All you need to make it work is a method name & block.

Notice that this will create an instance method on `Foo`, if you want to create a singleton method you have to do it like this:

```
class Foo
  define_singleton_method :reporter { instance_variable_get("@reporter") }
end
```

Key takeaway:

You should still write your methods using your regular "def" syntax, but if you need to create new methods, where the method name is not known in advance, then `define_method` could be a good fit.

Understanding Hooks

Hooks are a set of methods that allow you to react to events.

These methods are:

- included
- extended
- prepended

Here's an example:

```
module Products
  def self.included(base)
    puts "> Products included into #{base}!"
  end
end

class Shop
  include Products
end

class Cart
  include Products
end
```

This will produce:

```
> Products included into Shop!
> Products included into Cart!
```

There is a Ruby pattern that uses the `includes` hook to define a module with both instance methods & class methods.

It looks like this:

```ruby
module Machine
  def instance_method
  end

  def self.included(base)
    base.extend(ClassMethods)
  end

  module ClassMethods
    def class_method
    end
  end
end

class Human
  include Machine
end

Human.singleton_methods
# [:class_method]

Human.instance_methods.grep /instance_method/
# [:instance_method]
```

Review the chapter on modules if you don't remember what `include` & `extend` do.

Writing A DSL

A DSL (Domain Specific Language) is a way to provide nice and easy syntax to use or configure an application or library. If you have used something like RSpec or Rake before then you have used a DSL.

For example, in RSpec you can define a test like this:

```
describe Animal
  it 'must have a name' do
    expect(animal.name).to eq "Garfield"
  end
end
```

You can see that it almost reads like English.

There a are a few ways to implement a DSL. I will show you one way that combines a few different things you have learned in this book.

Our DSL is going to be used for sending email and it's going to look like this:

```
Mail.deliver do
  from    "jesus_castello@blackbytes.info"
  to      "example@example.com"
  subject "Welcome"
  body    "Hi there! Welcome to my newsletter."
end
```

What we have here is a class-level method being called on `Mail` with a block. Inside the block we have a few method calls with arguments.

To make this work I decided to use a module with `extend self`. This module will implement the `deliver` method and the DSL methods (`from`, `to`, `subject`, `body`).

Here is our first attempt:

```
module Mail
  extend self

  def deliver(&block)
    @options = {}
    yield
    send_now
  end

  def from(email)
    @options[:from] = email
  end

  # rest of the methods here...
end
```

When we run this code we will get this error: `undefined method 'from' for main:Object`.

Remember that procs & lambdas carry their execution context around, which means that when we call `yield` in this example, the block will not be able to see methods on the current object.

We can get around this by using an explicit block and `instance_eval(&block)`.

The following code should work.

It will build a hash with all the options passed in via the block and print them at the end.

```ruby
module Mail
  extend self

  def deliver(&block)
    @options = {}
    instance_eval(&block)
    send_now
  end

  def from(email)
    @options[:from] = email
  end

  def to(email)
    @options[:to] = email
  end

  def subject(title)
    @options[:subject] = title
  end

  def body(msg)
    @options[:body] = msg
  end

  def send_now
    puts "Sending email..."
    p @options
  end
end
```

These methods look a bit too verbose, don't they remind you of accessor methods?

Let's sprinkle some metaprogramming magic into this code. I'm going to use `define_method` so we can say something like this:

```
dsl_methods :from, :to, :subject, :body
```

To make this work we need to add this:

```
def dsl_methods(*args)
  args.each do |name|
    define_method(name) { |data| @options[name] = data }
  end
end
```

So our final code will look like this:

```ruby
module Mail
  extend self

  def dsl_methods(*args)
    args.each do |name|
      define_method(name) { |data| @options[name] = data }
    end
  end

  dsl_methods :from, :to, :subject, :body

  def deliver(&block)
    @options = {}
    instance_eval(&block)
    send_now
  end

  def send_now
    puts "Sending email..."
    p @options
  end
end
```

Notice that the `dsl_methods` line has to be **after** you define the method, otherwise you will get an error.

Conclusion

As you've seen in this book Ruby is a very powerful language, but the more advanced features like metaprogramming can sometimes obscure programming logic, so keep that in mind.

The good news is that with this new understanding you will become a more effective Ruby developer.

Remember to practice, review & enjoy programming with Ruby.

Thanks for reading!