27. Intro to Ruby

Intro to Ruby

Github repo

Introduction to Ruby

- Ruby Introduction
- Variables
- Conditionals
- Loops
- Methods
- Hashes
- Blocks
- Classes

Use the Docs!

Don't forget to generate the `ri` terminal docs on your system!

```
rvm docs generate # Generates docs for your Ruby version.
```

What you've Heard About Ruby

- Everything is an object!
- Similarities to JavaScript
- Use with Rails (2005) MVC Framework
- Packages are called Gems
- Similarities to Python
- More verbal / easy-to-read
- Red and shiny, so pretty!
- Community has a good reputation
- Not as popular as it used to be (why learn it!?)

What is Ruby?

Ruby, much like JavaScript, is an interpreted scripting language. It is, however, originally intended as a general-purpose programming language (JavaScript was *originally* only intended for use in web pages, though Node.js and other modern environments have changed that.)

It was originally developed in 1995 by Yukihiro Matsumoto (often referred to more simply as "Matz" in the Ruby community.) His original vision was to create a programming language that would ultimately give those writing code using it a great and fun user experience:

> I hope to see Ruby help every programmer in the world to be productive, and to enjoy programming, and to be happy. That is the primary purpose of Ruby language. — Yukihiro Matsumoto
> Again, like your experience with JavaScript, you'll find Ruby capable and suited for both object-oriented and procedural programming. You'll find it commonly used for writing command-line software, back-ends of web applications, and more.
> Note that **Ruby** is a programming language, and the popular **Ruby on Rails** is an MVC web application development framework *written in Ruby*. We'll also be looking into that framework to make use of Ruby in a more web-oriented way soon in the Lighthouse program; today will be an introduction of Ruby though, as you'll need an understanding of the language before you dive into its packages (called Gems.)
> Ruby, and its incredibly popular and influential Ruby on Rails framework for web applications, were both developed with you—the developer—in mind. Providing an extremely fast way to prototype and develop powerful applications. With any system, however, we'll always find there are pros and cons. Because of Ruby's focus on developer-friendly tools, productivity, and simplicity in writing code it finds itself as a less efficient environment than many of its competitors: you'll find executing a JavaScript file or PHP file to, more often than not, take less time. That being said, in most cases, nano- or miliseconds lost running basic scripts is not a concern with the power of modern hardware. It is only the most demanding scripts, or heaviest of server loads, that one might need to look into tricks to better scale an application or consider a different language. For the absolute best performance, after all, every interpreted scripting language should be overlooked in favour of compiled ones (C++ and C# come to mind), but these are commonly harder to debug and will take longer to write web applications in properly—pros and cons!

Why not to pick Ruby?

- It executes slower out of the box than many competitors.
- It is synchronous in nature.

Why pick Ruby?

- It is built from the ground up to be easy and fun to read and write (developer-friendly.)

- Everything is an object.
- Passionate community.

Ruby Essentials

- Ruby files end in `.rb`.
- Files are run via the `ruby` command in your terminal.
- To run code via an interactive shell, use `irb` (Interactive Ruby.)
- Convention for variable names is `snake_case` (underscore separated words.)
- There are fewer characters used in most lines of Ruby (no need for semi-colons, and many characters are optional.)

Ruby and the Command-Line

Common commands you'll likely find useful in Ruby include:

- `ruby --version` prints the active version of Ruby.
- `ruby -e 'RUBY CODE HERE'` to execute one line of Ruby in the terminal.
- `irb` to run the Interactive Ruby shell; you're able to run as many Ruby expressions as you'd like. Use `quit` to leave.
- `ruby FILENAME.rb` to execute the target script.
- `ruby FILENAME.rb -w` executes the target script with additional warnings enabled (recommended when starting out!)
- `ri CLASS_OR_METHOD_NAME` outputs any available documentation for the class or method name.
- `rvm install 1.2.3` installs a specific version of Ruby.
- `rvm use 1.2.3` changes the active version of Ruby to the one specified.
- `gem install GEMNAME` installs a global Ruby package (gem.)
- `gem list` outputs a list of installed Ruby packages (gems.)

Comments in Ruby

The only official comment character available in Ruby is the octothorpe / hashtag / pound sign: `#`. It is not uncommon for developers to quickly comment out multiple lines via the embedded document syntax as well, though it is often instead used for including snippets of other languages in your documents to be found and parsed by other interpretors.

```
# This is a single-line comment.

=begin
This
is
an embedded document...
but can be used for easy multi-line comments.
=end
```

Printing to the Terminal

One way is to use the `print` function, like so:

```
print "Hello, World!"
```

You'll find this does not add the output to a new line though. Typically, unless you want to have to manage new-line characters, you're looking for the similar `puts` function.

```
puts "Hello, World!"
```

Another, even shorter function is `p`, this wraps strings output to the terminal in quotes, so that you can quickly tell if the value is a string or not.

```
# Notice the difference in outputting this:
p "Hello, World!"

# ...and this:
p 3
```

The reason for the more descriptive output when using `p` is due to its automatic (and invisible) use of the `.inspect` method available in most any object. Note how these are identical in output:

```ruby
puts "Hello, World".inspect

p "Hello, World!"
```

Variables in Ruby

Ruby is dynamically typed (a variable's type can change) and allows for implicit typing during assignment (it can usually figure out the data-type for your variable based on the structure of the value—if there are quotes, if there is a decimal point, etc.)

An assignment of a value to a variable would look like:

```ruby
# Assignment.
name = "Sam"

# Output to terminal.
puts name
```

We can also accept user input with `gets` like so:

```ruby
print "Please enter your name: "
user_name = gets.chomp # .chomp removes the new-line character at the end of accepted input.

puts "Hello, "
print user_name
print "!"
```

Undefined Variables (`nil`)

If you'd like to set a variable to an undefined value, its type and value should be set to `nil` .

Attempting to `puts` or otherwise access an uninitialized variable will result in a `Name Error` ; there is no formal `undefined` like you may be used to in JavaScript.

Constants in Ruby

Ruby does not strictly enforce constants, but *will* give you a warning if you try to overwrite such a value. The convention for constants is to name a variable using `PascalCase` instead of `snake_case` , the interpretor watches this to help you avoid changing this.

```
# This is a constant, instead of a regular variable.
Name = "Alex"

# You'll receive warnings if you attempt to overwrite a constant's value.
Name = "Alexandra"

puts Name
```

Despite the basic convention being at least having the first letter being capitalized, *most* of the Ruby community prefers a more easily distinguishable approach of using entirely capital letters:

```
# This is a better constant; easier to notice in your code!
NAME = "Jean"

puts NAME
```

Common Literals

Common literals in Ruby include:

```
p 5 # Integer.
p 6.4 # Floating-point number.
p 'My string.' # String literal.
p "Another string." # Alternative string literal.
p /A regular expression/ # Regular expression literal (used in pattern-
matching.)
```

Learning About Ruby Objects

Everything in Ruby is an object... all values have a `.class` method you can use to retrieve the name of the class the object is an instance of:

```ruby
puts "5 is of class:"
puts 5.class

puts

puts "6.4 is of class:"
puts 6.4.class

puts

puts "'My string.' is of class:"
puts 'My string.'.class

puts

puts "\"Another string.\" is of class:"
puts "Another string.".class

puts

puts "/A regular expression/ is of class:"
puts /A regular expression/.class
```

Between use of `.inspect` and `.class`, you can learn a lot about the objects in your code. In JavaScript it is common to run `console.log()` to output values and learn more about them; in Ruby we use these methods.

Ruby also happens to include documentation in its source code, and this is available to us and makes learning about your objects and their methods even easier! If using `rvm`, you can see about generating the docs for use via this command:

```
rvm docs generate # Generates docs for your Ruby version.
```

These docs are not generated or unpacked by default to save disk space, but it is recommended you generate and use these documentations. They take up a few additional mega-bytes, but can save you a lot of searching.

You may find yourself "Google"-ing for answers a _lot_less frequently if you use `ri`, consider the following examples:

```
ri puts
```

```
ri p
ri String
```

```
ri String.empty?
```

```
ri String.size
```

```
ri Array
```

```
ri Array.sort
```

So, in Ruby, if you want to know what you're dealing with try `.class` to find out what it is, run `.inspect` to see what is inside, and if you want further instructions head over to your terminal and check the docs via `ri`!

Not sure what you can do with an object? Every object has a `.methods` method as well that will simply list out all methods it is associated with.

```
puts "String Methods:"
puts "***************"
puts "Hello, World!".methods

puts

puts "Integer Methods:"
puts "****************"
puts 3.class.methods

puts

puts "Array Methods:"
puts "**************"
puts Array.methods

puts

my_bool = true
puts "TrueClass Methods:"
puts "******************"
puts my_bool.methods
```

Strongly-Typed; The Comparison Operator

JavaScript was a weakly-typed language. We found we could get away with all sorts of strange comparisons of values, sometimes to strange effect. Consider the following JavaScript; guess what each will evaluate to:

```
true + true === 2 // => true
true === 1 // => false

0 == false // => true
0 === false // => false
0 === +false // => true

const myArray = [];
if (myArray) console.log('myArray TRUTHY');
else console.log('myArray FALSEY');
// => "myArray TRUTHY"
```

It can be hard to know what is going to happen when the language you're writing code in allows you to compare anything. In some cases a value's type gets coerced to another to assist in comparison (especially in cases wherein we compare via `==` in JavaScript.)

Ruby, unlike JavaScript, is strongly typed. Situations like the above cannot, and will not happen. It is expected we are more careful and cautious with our comparisons to avoid what we've seen in JavaScript—confusing code. Because of this expectation, our code becomes more legible and more predictable. Instead of a coercive and a strict type comparison operator, in Ruby we need only a single comparison operator that is already strict in typing: `==`.

```
puts 3 == "3" # => false
```

If we want to compare the above values and treat both as numbers, we must put thought into it and formally convert their types (or attempt to) to the same. As developers, we'd have to decide if the values above should both be strings or both be integers.

We can attempt to convert the type of a value via various methods like `.to_i` for converting to an integer, or `.to_s` to convert to a string.

```
num = 7

p num
p num.to_s # Note that this doesn't mutate the value stored in the
variable; this line outputs a string.
p num # The variable still holds a number!
```

Note how this can be used to ensure far more intentional and predictable comparisons:

```
print '3 == "3" is '
puts 3 == "3"

puts

print '3 == "3".to_i is '
puts 3 == "3".to_i

puts

print '3.to_s == "3" is '
puts 3.to_s == "3"
```

Everything in Ruby is an object… with Ruby you'll get used to just about everything you're working with having a whole heap of properties and methods. It can be helpful, especially when getting started, to always have an `irb` tab open in your terminal to run tests and inspect values.

Stay in the habit of cracking objects open; we did this all the time in JavaScript with `console.log()`!

String Concatenation; String Interpolation

Basic string concatenation is a breeze! Ensure your values are strings and you can glue them together with the `+` symbol:

```
welcome_text = "Hello,"
welcome_name = "Robin"
welcome_string = welcome_text + " " + welcome_name + "!"

puts welcome_string
```

String interpolation will feel familiar to JavaScript's string template literals, check this out:

```
first_name = "Jesse"
last_name = "Asher"

full_name = '#{first_name} #{last_name}' # Doesn't work in single quotes.
print "Single Quotes: "
puts full_name

puts

full_name = "#{first_name} #{last_name}" # String interpolation requires
double quotes!
print "Double Quotes: "
puts full_name
```

String interpolation attempts to run an object's `.to_s` method automatically; concatenation does not.

The multiplication operator, `*`, is actually compatible with strings (technically a method of strings.)

```
ri String.*
```

It ends up repeating the string the number of times specified!

```
puts "Hello from Ruby! " * 3
```

Conditionals

If statements aren't much of a surprise, read the following example; try changing the number to get each result:

```
print "Enter an integer: "
num_to_check = gets.chomp.to_i

if num_to_check > 3
  puts "Number is greater than 3."
elsif num_to_check < 3
  puts "Number is less than 3."
else
  puts "Number is 3."
end
```

Note that instead of curly braces `{}`, our statements require an `end` to let Ruby know we're done. Typing `elsif` instead of `else if` may also take some getting used to! Notice how it is a trend in Ruby that we'll often require fewer keystrokes to reach our goal: this is in that recurring theme of it being a developer-friendly language.

You can use parenthesis for conditionals if you'd like, but they are entirely optional and often excluded:

```
num_to_check = 5

if (num_to_check > 3) # Optional parenthesis.
  puts "Number is greater than 3."
elsif (num_to_check < 3) # Optional parenthesis.
  puts "Number is less than 3."
else
  puts "Number is 3."
end
```

Consider the following conditional statement:

```
string_to_check = "Racecar"

if string_to_check != "Minivan"
  puts "Might be a Racecar."
end
```

Matz sees it as awkward, and potentially feel negative to use "not equals (`!=`)." While it still works, we are offered an alternative to `if` for such cases:

```
string_to_check = "Racecar"

unless string_to_check == "Minivan" # The inverse is checked; we seldom
need "!=".
  puts "Might be a Racecar."
end
```

`unless` statements can still include an `else` if/as necessary.

In JavaScript we could get away with one-line conditional statements like...

```
if (3 > 2) console.log('3 is greater than 2.');
```

Can we write single-line conditional statements in Ruby? We can, they do look a little different though!

```
puts "3 is greater than 2." if 3 > 2
```

Note that it reads more like plain English... *"output the string if the condition is true."* Of course, this can take some getting used to after spending so much time with JavaScript!

It is important to know, when writing conditions, that the only `false` values are considered to be the `false` boolean value and `nil`. All other values, if used as your condition, would be treated as `true`. This is much simpler than JavaScript and its weakly typed / coercive approach.

```
snowing = false
puts "Put away the shovel." unless snowing
```

Ruby also supports the ternary syntax you are used to:

```
example_int = 5
puts example_int < 10 && example_int > -10 ? "Single digit." : "Multiple
digits."

example_int_two = 36
puts example_int_two < 10 && example_int_two > -10 ? "Single digit." :
"Multiple digits."
```

Recall that JavaScript, like many languages, has a handy alternative to `if` statements: `switch` statements!

```
const username = 'Regan';
const messageType = 'hello'; // Change this to test switch cases.

switch (messageType) {
  case 'hello':
    console.log(`Hi there, ${username}; welcome to the program!`);
    break;
  case 'goodbye':
    console.log(`Time to go? Thanks for stopping by, ${username}!
Cheers.`);
    break;
  default:
    console.log(`How are you doing today, ${username}?`);
    break;
}
```

Ruby has `case` statements, which serve the same purpose but look (and named) a bit different:

```
username = "Regan"
message_type = "hello"

case message_type
  when "hello"
    puts "Hi there, #{username}; welcome to the program!"
  when "goodbye"
    puts "Time to go? Thanks for stopping by, #{username}! Cheers."
  else
    puts "How are you doing today, #{username}?" # Default response.
end
```

Loops

Like most languages, Ruby offers us a variety of options when it comes to loops! You'll have to decide which is most suitable for the task you're working on, on a case-by-case basis.

Break Loop

A break loop begins with a `loop do`, followed by set of instructions you'd like to repeat, proceeded by an `end` to let Ruby know which line ends the statement. Including only these components will result in an endless loop, so be careful to include a `break` somewhere in between `loop do` and `end`. Usually, you'll want to use this to terminate the loop. A basic example might look as follows:

```
iterator_int = 0

loop do
  iterator_int += 1 # Note that Ruby does not support: iterator_int++
  puts "The break loop has run #{iterator_int} times."

  break if iterator_int > 9 # break terminates the loop.
end
```

While Loop

With the `while` loop, we get to define the cosudndition for termination in the initializing statement. Unless the cause for breaking the loop is rather complex, this is often preferred as it will save you a line:

```
loop_num = 17

while loop_num > 3
  loop_num -= 2
  puts "loop_num=#{loop_num}"
end
```

Much like how `if` has an inverse: `unless`, `while` has an inverse: `until`.

```
new_loop_num = 3

until new_loop_num > 729
  new_loop_num *= 3
  puts "new_loop_num=#{new_loop_num}"
end
```

For...In Loop

When it comes to arrays, in JavaScript we had the nicety of `for...of` loops. Just like that, Ruby offers us the similiar `For...In` loop! Observe:

```
names = ["Jordan", "Cameron", "Kirby", "Laurel", "Keegan", "Hollis",
"Randi", "Leigh"]

puts "Names in names array:"
for name in names do
  puts name
end
```

Each Loop

There is also a `forEach` equivalent via `.each`:

```
animals = ["Tardigrade", "Ostrich", "Giraffe", "Orca", "Grey Wolf", "Sow
Bug", "Lobster"]

puts "List of animals:"
animals.each do |animal|
  puts animal
end
```

You can grab the index if you'd like as well:

```
animals = ["Tardigrade", "Ostrich", "Giraffe", "Orca", "Grey Wolf", "Sow
Bug", "Lobster"]

puts "List of animals:"
animals.each_with_index do |animal, index|
  puts "#{index}. #{animal}"
end
```

Ruby supports number ranges by use of the range operators: `..` and `...`. This makes it very easy to check if a number exists between two known values, or even easily loop from one value to another:

```
# A range including the maximum number.
puts "30..36:"
puts "*******"
(30..36).each do |num|
  puts num
end

puts

# A range excluding the maximum number.
puts "30...36:"
puts "********"
(30...36).each do |num|
  puts num
end
```

Times Method Loop

For simple iterations, we can void unnecessary additional work like we see in `loop` `break` and `while` loops by using integer objects' `.times` method. See the following demonstration:

```
20.times do |iteration|
  puts "Times iterated: #{iteration}"
end
```

This last example is something that will come in very handy throughout your time with Ruby, it is much simpler syntax for looping than the approach taken by almost every other language.

Methods

In JavaScript (and many languages,) we refer to named sets of repeatable instructions as **functions**. If they're found inside of a class or an object, we instead refer to them as **methods**.

Everything is an object in Ruby, so only the term **method** is used for these sorts of structures. Ruby is so intent on everything being objects, that even mathematical operators are considered number methods. Observe, a simple line of Ruby:

```
p 1 + 3
```

Of course, the above will output **4**, but what do we mean by mathematical operators being methods? Well, it turns out that technically the above is a more developer-readable way of writing the following:

```
p 1.+(3)
```

That's right... `1` is an integer object with a method called `+` inside. That method can be passed an argument (`3` , in this case) and returns their sum. Despite this happening and us having the option of typing mathematical operations in this way, the Ruby interpretor allows us to write it in the more human readable fashion.

Writing your own Methods

The syntax in Ruby for writing methods is by use of the `def` keyword followed by the name of your method. Like most statements in Ruby that take up multiple lines, you'll end the function with the aptly named `end` keyword as you might expect. Note that the value in the last line of your method will be returned by default unless you specify otherwise.

```ruby
def hello_world
  puts "Hello, World!"
end

# Run it by calling its name:
hello_world

# Parenthesis are optional:
hello_world()
```

Parameters can be specified after the method name, much like we had seen in JavaScript. Optionally, you can wrap parameters in parenthesis.

```ruby
def say_hello_to name
  puts "Hello, #{name}!"
end

# Run your method:
say_hello_to "George"
```

You can call upon a method with, or without, parenthesis surrounding arguments. Note that when calling upon a method, it is expected that you provide the same number of arguments as there are parameters unless default values are set.

```ruby
def say_hello_to name
  puts "Hello, #{name}"
end

# Running this without passing an argument will cause an error.
say_hello_to # Comment this line out so it does not crash your program.

def say_hello_from name="World"
  puts "Hello, from #{name}!"
end

# Because there is a default value for parameter "name", that argument is
optional.
say_hello_from # No error occurs here.

say_hello_from "Bailey"
```

Hashes

These are most comparable to objects in JavaScript; such a structure formed by key-value pairs is often referred to as a map, associative array, dictionary, or hash—usually varying in name only as a formality of the language you are using. Even the syntax for hashes you may find similar to JavaScript's objects:

```
employee = {
  "first_name" => "Evan", # The "fat arrow" is usually called a "hash rocket" by Ruby devs.
  "last_name" => "Valentine",
  "title" => "Supervisor",
  "active" => true
}

p employee
```

Periods are used to access methods, so to access values held within a hash we instead use square brackets:

```
employee_1 = {
  "first_name" => "Kit",
  "last_name" => "Schuyler",
  "title" => "Human Resources",
  "active" => true
}

# Accessing values in the hash by key:
puts "Employee Name: #{employee_1["last_name"]}, #{employee_1["first_name"]}"
```

In most languages, a string is a primitive, and causes very little performance overhead. What is that recurring theme with Ruby, though? Yes, even strings are objects that are initialized and come complete with all sorts of methods. Because of this, it becomes much more efficient to work with something much slimmer... symbols. In JavaScript we touched on symbols, but they are not often used in that language as more convenient primitives are available. Recall that symbols don't really act as a complex value... they end up just being a pointer that aims at its label in memory. They can't be changed once set, and are simply an easy way at setting a unique identifier that can't do much else. A unique identifier that can't do much else... that's the solution we see implemented in Ruby to ensure faster execution times and less slow-down in your apps. The syntax for a symbol in Ruby is a colon proceeded by a symbol label/name. Let's build a hash using symbols instead of strings for keys this time:

```
employee_2 = {
  :first_name => "Meredith",
  :last_name => "Emerson",
  :title => "Manager",
  :active => true
}

# Accessing values in the hash by symbol-based key:
puts "Employee Name: #{employee_2[:last_name]},
#{employee_2[:first_name]}"
```

This was seen as a little awkward to type and read, so a shorthand was developed that will feel even more familiar to what we encountered in JavaScript:

```
employee_3 = {
  first_name: "Kree",
  last_name: "Mackenzie",
  title: "Stock Associate",
  active: true
}

# Accessing values in the hash by symbol-based keys:
puts "Employee Name: #{employee_3[:last_name]},
#{employee_3[:first_name]}"
```

We can check a value for a key we have received dynamically by converting a string to a symbol like so:

```
key_1 = "title"
key_2 = "last_name"

employee_4 = {
  first_name: "Nevada",
  last_name: "Waverly",
  key_1.to_sym => "Stock Associate", # Setting symbol key via string
(convert to symbol.)
  active: true
}

# Retrieving a symbol key via string (convert to symbol.)
puts "#{employee_4[key_2.to_sym]}, #{employee_4[:first_name]}:
#{employee_4[:title]}"
```

Blocks

A block in JavaScript was typically seen as any number of lines that reside inside of an opening and corresponding closing curly brace. This is true as well in Ruby, however Ruby has two sets of opening / closing markers:

1. `{` and `}`
2. `do` and `end`

The above are equivalent! Consider the following example, and note how the code will behave the same either way:

```
operating_systems = ["FreeDOS", "Linux", "MacOS", "ReactOS", "Windows"]

# Blocks can be contained within do...end,
operating_systems.each do |os|
  puts  os
end

# and blocks can also be contained within {...}
operating_systems.each { |os|
  puts os
}
```

Typically `do...end` is considered easier to read, and is favoured by Ruby developers when the block is multi-line.

With this idea of blocks in Ruby, we can now explore the idea of Lambdas.

Lambdas

Lambdas are essentially a way for us to store a block of code in a variable. This can also be described as an anonymous function. There are two syntaxes for lambdas:

1. Standard: `lambda { |foo| }`
2. Literal: `->(foo) {}`

Note that when accepting a lambda as an argument, you'll need to preceed its name with an ampersand (`&` .) See an example of both syntaxes in action:

```
insects = ["Mosquito", "Dragonfly", "Grasshopper", "Mantis", "Butterfly"]

output_string = lambda { |string| puts string }
output_string_2 = ->(string) { puts string }

puts "Standard lambda experiment:"
insects.each &output_string

puts

puts "Lambda literal experiment:"
insects.each &output_string_2
```

Blocks can be passed into methods, very similarly to what we often saw in JavaScript with callbacks.

```
say_hi = lambda { puts "Hi!" }

def my_method &callback
  callback.call # Use the block's .call method to invoke/run/execute it.
end

my_method &say_hi
```

Classes

Classes are a blueprint that we can use to build predictable and consistent objects from. This should be a familiar concept from your time in JavaScript, and the syntax should, overall, feel natural as well.

Let's build a class to learn what they look like in Ruby:

```ruby
class Pet # We start with the "class" keyword, followed by a PascalCased
name.
  def initialize name, type, age # The constructor is a method called
"initialize".
    @name = name # Properties are assigned and read like variables, but
are preceeded with an "@" character.
    @type = type
    @age = age
  end

  def say_hello # It is easy to add additional methods like we normally
would.
    puts "#{@name} says: Greetings!"
  end
end

# As everything behaves as an object, we use class' ".new" methods to
create an instance.
paxton = Pet.new "Paxton", "Bearded Dragon", 2

# Crack it open, see what's inside!
p paxton

# Run methods from the object as per usual.
paxton.say_hello
```

Note that in the above example we cannot access name, type, or age of the pet object. If we try to access `paxton.name`, `paxton["name"]`, `paxton.@name`, or even `paxton["@name"]` we'll be met with an error for each of these attempts. This is because by default, all property values are considered `private`—meaning that they are only accessible within the class, and not outside in the rest of your program. We can make properties publically accessible by adding an attribute reader for any properties you'd like to make public:

```ruby
class Pet
  attr_reader :name, :type, :age # We can make properties readable via
attr_reader.

  def initialize name, type, age
    @name = name
    @type = type
    @age = age
  end

  def say_hello
    puts "#{@name} says: Greetings!"
  end
end

aston = Pet.new "Aston", "Axolotl", 1

# We can now access properties via their public reader methods:
puts "#{aston.name} is an #{aston.type} that is #{aston.age} year old!"
```

Should you want the properties to be writeable (that is, to be able to re-assign their values) you can add an attribute writer as well:

```ruby
class Pet
  attr_reader :name, :type, :age
  attr_writer :age # We can make properties readable via attr_writer.

  def initialize name, type, age
    @name = name
    @type = type
    @age = age
  end

  def say_hello
    puts "#{@name} says: Greetings!"
  end
end

jo = Pet.new "Jo", "Monitor Lizard", 4

# Happy birthday, Jo!
jo.age += 1 # Age should now be 5.

# We can now access properties via their public reader methods:
puts "#{jo.name} is a #{jo.type} that is #{jo.age} years old!"
```

For properties you'd like to both read and write, you can instead make use of the attribute accessor which allows both at once:

```ruby
class Pet
  attr_reader :name, :type
  attr_accessor :age # We can make properties readable AND writable via attr_accessor.

  def initialize name, type, age
    @name = name
    @type = type
    @age = age
  end

  def say_hello
    puts "#{@name} says: Greetings!"
  end
end

gex = Pet.new "Gex", "Gecko", 26

# Happy birthday, gex!
gex.age += 1 # Age should now be 27.

# We can now access properties via their public reader methods:
puts "#{gex.name} is a #{gex.type} that is #{gex.age} year old!"
```

Exceptions in Ruby

When running code that is likely to fail or throw an error, it is a good idea to run it in a protected statement. In JavaScript this was a try...catch, in Ruby please observe rescuing a failure in code:

```ruby
begin
  # Your code that may fail.
rescue # You can grab the exception, if available.
  # Error handler.
else
  # Executes if no error present.
ensure
  # Always executes whether error-free or error-filled.
end
```

Many of these steps are optional, often you may end up with code that is quite analogous to try...catch:

```
begin
  puts "string".non_existant_method
rescue Exception => exception
  puts exception.message
  p exception.backtrace
end
```

If you'd like to include your own errors in your code, you can utilize the `raise` keyword. Here is a basic example, but know that you can even set up your own exception classes if you so choose:

```
begin
  test_val = "abc"
  raise Exception, "Integer expected!" unless test_val.class == "Integer"
rescue Exception => exception
  puts exception.message # Your "raised" message will appear here!
  p exception.backtrace
end
```

Resources

- Official Ruby Language Website
- About Ruby
- Ruby Documentations
  - Ruby API
  - String
  - Integer
  - Array
  - Hash
  - Symbol
  - Class
- Try Ruby in the Browser
- Ruby in 20 Minutes
- Learn Ruby Online (Interactive Ruby Tutorial)
- Book: Programming Ruby, The Pragmatic Programmer's Guide

- Book: O'Reilly's The Ruby Programming Language *Warren's favourite!*

View on Compass | Leave Feedback