In this lecture, we will discuss…

- Brief history of Ruby
- Another programming language to learn? Why?
- Basic Ruby principles and conventions

Ruby History

- Invented by Yukihiro “Matz” Matsumoto
- Version 1.0 released in 1996 (Japan)
- Popularized by Ruby on Rails beginning in 2005

Ruby: High Level Overview

- Dynamic
- Object-oriented
  - Object-possessed, almost everything is an object
- **Elegant**, **expressive and declarative**
  - Terse at times, but extremely readable
- Influenced by Perl, Smalltalk, Eiffel and Lisp

“Designed to make programmers happy”

…Java…

```
public class Print3Times {
  public static void main(String[] args){
    for(int i = 0; i < 3; i++) {
      System.out.println("Hello World!");
    }
  }
}
```

…Ruby…

```
3.times { puts "Hello World" }
```

Carried away…

https://github.com/jpfuentes2/a-letter-to-Augusta#the-letter

The Letter

```
require "./love"

a_letter to: Augusta do
  twas(only: 16.months.ago) { The::Universe << You.to(OurFamily) }
  life.has :been => %w(i n c r e d i b l y).zip(*"wonderful!").ever_since
  We::Wish.we_could { experience these_moments: over & over }
  You.will always_be: Loved, and: Cherished
  until Infinity.ends do; Forever.(); end
end
```

Ruby Basics

- 2 space indentation for each nested level is encouraged
  - Not required (unlike Python)

- # is used for comments
  - Use comments in moderation – the code itself should tell the story

- Everything is evaluated!

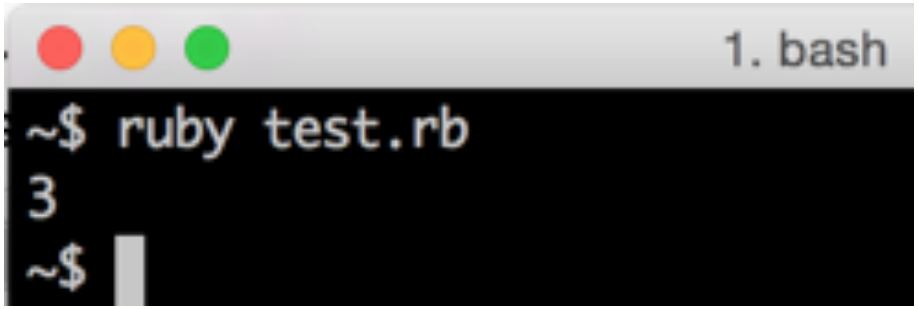
```
# this is a comment
puts 5 # so is this
3 # and this
```

Printing to Console

- *puts* - Standard Ruby method to print strings to console (as in **put s**tring)
  - Adds a new line after the printed string
  - Similar to *System.out.println()* in Java
  - Used for most of the examples
- *p* - Prints out internal representation of an object
  - Debugger-style output

```
p "Got it" # => Got it
```

Executing Ruby



Naming Conventions

- Variables
  - Lowercase or `snake_case` if multiple words
- Constants
  - Either `ALL_CAPS` or `FirstCap`
- Classes (and Modules)
  - `CamelCase`

Drop the Semicolons

- Leave semicolons off at the end of the line
- Can cram several statements in with a semicolon in between
  - Usually highly discouraged

```
a = 3 # semicolons not needed
a = 2; b = 3 # sometimes used
```

IRB – Interactive Ruby

- Console-based interactive Ruby interpreter
  - REPL (Read Evaluate Print Loop)
- Comes with a Ruby installation
- Lets you experiment (quickly!)

```
~$ irb
irb(main):001:0> "hello world"
=> "hello world"
irb(main):002:0> puts "hello world"
hello world
=> nil
```

Anything evaluates to something – no need to assign to a variable

puts returns nil

Summary

- Ruby is extremely expressive
- Everything is evaluated

What's next?

- Flow of control in Ruby

In this lecture, we will discuss…

- Flow of Control
  - if / elsif / else
  - case
  - until / unless?
  - while / for
- What is true and what is false?
- What in the world is === ?

Flow of Control

- `if, unless, elsif, else`
- No parentheses or curly braces
- Use `end` to close flow control block

```ruby
a = 5 # declare a variable

if  a == 3
  puts "a is 3"
elsif a == 5
  puts "a is 5"
else
  puts "a is not 3 or 5"
end

# => a is 5
```

```ruby
a = 5

unless a == 6
  puts "a is not 6"
end

# => a is not 6
```

Flow of Control

- `while, until`

```ruby
a = 10

while a > 9
  puts a
  a -= 1
  # same as a = a - 1
end

# => 10
```

```ruby
a = 9

until a >= 10
  puts a
  a += 1
end

# => 9
```

Flow of Control: Modifier Form

- `if, unless, while, until` – on the same line as the statement

```ruby
# if modifier form

a = 5
b = 0

puts "One liner" if a == 5 and b == 0
# => One liner
```

```ruby
# while modifier form

times_2 = 2
times_2 *= 2 while times_2 < 100
puts times_2 # => ?
```

True / False

- `false` and `nil` objects are false

- **Everything else** is true!

```
puts "0 is true" if 0 # => 0 is true
puts "false is true?" if "false" # => false is true?
puts "no way - false is false" if false # => NOTHING PRINTED
puts "empty string is true" if "" # => empty string is true
puts "nil is true?" if "nil" # => nil is true?
puts "no way - nil is false" if nil # => NOTHING PRINTED
```

```
warning: string literal in condition
```

Triple Equal

- Triple Equal: ===
- “Equal” in its own way
  - Sometimes it’s not about being exactly equal

```
if /sera/ === "coursera"
  puts "Triple Equals"
end
# => Triple Equals

if "coursera" === "coursera"
  puts "also works"
end
# => also works

if Integer === 21
  puts "21 is an Integer"
end
# => 21 is an Integer
```

Case Expressions

- Two “flavors”
  1. Similar to a series of “`if`” statements
  2. Specify a target next to `case` and each `when` clause is compared to target

- === is called the case equality operator because it is used in precisely this case!

- **No fall-through logic**

Case Expressions

```ruby
age = 21

case # 1ST FLAVOR
  when age >= 21
    puts "You can buy a drink"
  when 1 == 0
    puts "Written by a drunk programmer"
  else
    puts "Default condition"
end
# => You can buy a drink
```

```ruby
name = 'Fisher'
case name # 2nd FLAVOR
  when /fish/i then puts "Something is fishy here"
  when 'Smith' then puts "Your name is Smith"
end

#=> Something is fishy here
```

For Loop

- Hardly used
- `each` / `times` preferred

```
for i in 0..2
  puts i
end

# => 0
# => 1
# => 2
```

Summary

- Lots of options for flow of control
- Modifier form is an interesting way to be very expressive
- Non-nil and non-false values are always true

What's next?

- Functions / Methods

In this lecture, we will discuss…

✧ Functions / Methods
- Definitions
- How do you call them?
- What and how do they return?
- Default args

✧ How to make methods more expressive

✧ What is “splat”

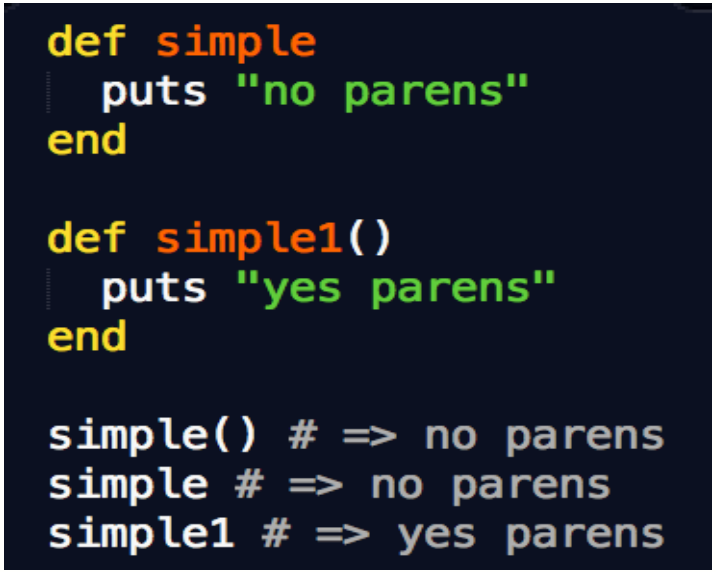
~~Functions~~ and Methods

- ✧ Technically, a **function** is defined **outside** of a class and a **method** is defined **inside** a class
- ✧ In Ruby, **every** function/method has at least one class it belongs to
  - Not always written inside a class

Conclusion: Every function is really a method in Ruby

Methods

- ✧ Parentheses are **optional** both when **defining and calling** a method
  - Used for **clarity**



```ruby
def simple
  puts "no parens"
end

def simple1()
  puts "yes parens"
end

simple() # => no parens
simple # => no parens
simple1 # => yes parens
```

Return

- **No need** to declare type of parameters
- Can return **whatever you want**
- `return` keyword is optional (last executed line returned)

```ruby
def add(one, two)
  one + two
end

def divide(one, two)
  return "I don't think so" if two == 0
  one / two
end

puts add(2, 2) # => 4
puts divide(2, 0) # => I don't think so
puts divide(12, 4) # => 3
```

Expressive Method Names

- Method names can end with:
  - ‘?’ - Predicate methods
  - ‘!’ - Dangerous side-effects (*example later by strings*)

```ruby
def can_divide_by?(number)
  return false if number.zero?
  true
end

puts can_divide_by? 3 # => true
puts can_divide_by? 0 # => false
```

Default Arguments

- ✧ Methods can have **default arguments**
  - If a value is passed in – use that value
  - Otherwise – use the default value provided

```
def factorial (n)
  n == 0? 1 : n * factorial(n - 1)
end

def factorial_with_default (n = 5)
   n == 0? 1 : n * factorial_with_default(n - 1)
end

puts factorial 5 # => 120
puts factorial_with_default # => 120
puts factorial_with_default(3) # => 6
```

Ternary operator:
condition ? true : false

Splat

- * **prefixes parameter inside** method definition
  - Can even apply to **middle parameter**, not just the last

```
def max(one_param, *numbers, another)
  # Variable length parameters passed in
  # become an array
  numbers.max
end

puts max("something", 7, 32, -4, "more") # => 32
```

Summary

- There is **no need** to declare parameter type passed in or returned (dynamic)
- `return` is **optional** – the last executable line is "returned"
- You can construct methods with **variable number** of arguments or default arguments

What's next?

- Blocks

In this lecture, we will discuss…

- ✧ Blocks
- ✧ How they are used
- ✧ How to incorporate them into your own methods

Blocks

✧ Chunks of code

- Enclosed between either curly braces (`{ }`) or the keywords `do` and `end`
- Passed to methods as last “parameter”

Blocks

- **Convention:**
  - Use `{ }` when block content is a single line
  - Use `do` and `end` when block content **spans multiple lines**
- Often used as **iterators**

Blocks

- ✧ Can accept arguments

```ruby
1.times { puts "Hello World!" }
# => Hello World!

2.times do |index|
  if index > 0
    puts index
  end
end
# => 1

2.times { |index| puts index if index > 0 }
# => 1
```

Coding with blocks

✧ Two ways to configure a block in your own method

Implicit

- **Use `block_given?` to see if block was passed in**
- **Use `yield` to "call" the block**

Explicit

- **Use `&` in front of the last "parameter"**
- **Use `call` method to call the block**

Implicit

- ✧ Need to check "`block_given?`"
  - Otherwise, an **exception is thrown**

```ruby
def two_times_implicit
  return "No block" unless block_given?
  yield
  yield
end

puts two_times_implicit { print "Hello "} # => Hello
                                          # => Hello
puts two_times_implicit # => No block
```

Explicit

- Should check if the block is `nil?`

```ruby
def two_times_explicit (&i_am_a_block)
  return "No block" if i_am_a_block.nil?
  i_am_a_block.call
  i_am_a_block.call
end

puts two_times_explicit # => No block
two_times_explicit { puts "Hello"} # => Hello
                                   # => Hello
```

Summary

✧ Blocks are **just code** that you can pass into **methods**

✧ When incorporating into your own methods:

- **Either** use blocks **implicitly**
- **Or** call them **explicitly**

What's next?

✧ Files and Environment Variables

In this lecture, we will discuss…

- ✧ Reading and writing to files
- ✧ Exceptions
- ✧ Reading values from environment variables

Reading from File

```ruby
File.foreach( 'test.txt' ) do |line|
  puts line
  p line
  p line.chomp # chops off newline character
  p line.split # array of words in line
end
```

test.txt

```
The first line of the file
And the second
Third
```

```
The first line of the file
"The first line of the file\n"
"The first line of the file"
["The", "first", "line", "of", "the", "file"]
And the second
"And the second\n"
"And the second"
["And", "the", "second"]
Third
"Third"
"Third"
["Third"]
```

Output

Reading from Non Existing File

```
1  File.foreach( 'do_not_exist.txt' ) do |line|
2    puts line.chomp
3  end
4
5
6
```

```
/Users/kalmanhazins/coursera/code-module2/Lecture5-Files-Environment-
Vars/read_from_file_handle_exceptions.rb:1:in `foreach': No such file
or directory @ rb_sysopen - do_not_exist.txt (Errno::ENOENT)
```

Handling Exceptions

```ruby
begin

  File.foreach( 'do_not_exist.txt' ) do |line|
    puts line.chomp
  end

rescue Exception => e
  puts e.message
  puts "Let's pretend this didn't happen..."
end

```

```
No such file or directory @ rb_sysopen - do_not_exist.txt
Let's pretend this didn't happen...
```

Alternative to Exceptions

```
if File.exist? 'test.txt'

  File.foreach( 'test.txt' ) do |line|
    puts line.chomp
  end

end
```

Writing to File

The file is automatically closed after the block executes

Environment Variables

If needed, revisit Module 1 on how to setup environment variables for your Operating System

```
puts ENV["EDITOR"] # => subl
```

Summary

- Files **automatically closed** at the end of the block
- **Either** use exception handling **or** check for existence of the file before accessing

What's next?

- Strings and Symbols

In this lecture, we will discuss…

- ✧ Different kinds of strings supported by Ruby
- ✧ Many methods supported by the String API
- ✧ Symbols

Strings

- ✧ Single-quote literal strings are **very** literal
  - Allow escaping of `'` with `\`
  - Show (almost) everything else **as is**
- ✧ Double-quoted strings
  - Interpret special characters like `\n` and `\t`
  - Allow string interpolation!

Don't bother concatenating with +

Strings / Interpolation

```
single_quoted = 'ice cream \n followed by it\'s a party!'
double_quoted = "ice cream \n followed by it\'s a party!"

puts single_quoted # => ice cream \n followed by it's a party!
puts double_quoted # => ice cream
                   # =>   followed by it's a party!

def multiply (one, two)
  "#{one} multiplied by #{two} equals #{one * two}"
end
puts multiply(5, 3)
# => 5 multiplied by 3 equals 15
```

Interpolation (only available for double-quoted strings)

More Strings

- ✧String methods ending with `!` modify the existing string
  - Most others just return a new string
- ✧Can also use `%Q{long multiline string}`
  - Same behavior as double-quoted string

Very Important to Master String API

More Strings

```ruby
my_name = " tim"
puts my_name.lstrip.capitalize # => Tim
p my_name # => " tim"
my_name.lstrip! # (destructive) removes the leading space
my_name[0] = 'K' # replace the fist character
puts my_name # => Kim

cur_weather = %Q{It's a hot day outside
                 Grab your umbrellas…}

cur_weather.lines do |line|
  line.sub! 'hot', 'rainy' # substitute 'hot' with 'rainy'
  puts "#{line.strip}"
end
# => It's a rainy day outside
# => Grab your umbrellas...
```

Strings API

include? other_str → true or false

Returns `true` if *str* contains the given string or character.

```
"hello".include? "lo"   #=> true
"hello".include? "ol"   #=> false
"hello".include? ?h     #=> true
```

Symbols

- ✧ `:foo-` highly optimized strings
- ✧ **Constant names** that you don't have to pre-declare
  - **"Stands for something"** string type

Symbols

- ✧ Guaranteed to be **unique** and **immutable**
- ✧ Can be converted to a `String` with `to_s`
  - **Or** from `String` to `Symbol` with `to_sym`

Symbols

```
~$ irb
irb(main):001:0> "hello".methods.grep /case/
=> [:casecmp, :upcase, :downcase, :swapcase, :upcase!, :downcase!, :swapcase!]
```

Summary

- ✧ Interpolation lets you **finish your thought**
- ✧ Strings have a lot of **really useful API**

What's next?

- ✧ Arrays

In this lecture, we will discuss…

- ✧ Arrays
  - How they are **created**
  - How to modify **arrays**
  - **Accessing elements** inside arrays

Arrays

- Collection of **object references** (auto-expandable)
- Indexed using `[]` operator (method)
- Can be indexed with **negative numbers** or **ranges**
- **Heterogeneous types allowed** in the same array
- Can use `%w{str1 str2}` for string array creation

Arrays

```ruby
het_arr = [1, "two", :three] # heterogeneous types
puts het_arr[1] # => two (array indices start at 0)

arr_words = %w{ what a great day today! }
puts arr_words[-2] # => day
puts "#{arr_words.first} - #{arr_words.last}" # => what - today!
p arr_words[-3, 2] # => ["great", "day"] (go back 3 and take 2)

# (Range type covered later...)
p arr_words[2..4] # => ["great", "day", "today"]

# Make a String out of array elements separated by ‘,’
puts arr_words.join(',') # => what,a,great,day,today!
```

Arrays

- ✧ Modifying arrays
  - Append: `push` or `<<`
  - Remove: `pop` or `shift`
  - Set: `[]=` (method)

Arrays

- Randomly pull element(s) out with `sample`
- Sort or reverse with `sort!` and `reverse!`

Arrays

```ruby
# You want a stack (LIFO)? Sure
stack = []; stack << "one"; stack.push ("two")
puts stack.pop # => two

# You need a queue (FIFO)? We have those too...
queue = []; queue.push "one"; queue.push "two"
puts queue.shift # => one

a = [5,3,4,2].sort!.reverse!
p a # => [5,4,3,2] (actually modifies the array)
p a.sample(2) # => 2 random elements

a[6] = 33
p a # => [5, 4, 3, 2, nil, nil, 33]
```

Arrays

- Lots of useful array methods
  - `each` – loop through array
  - `select` – filter array by selecting
  - `reject` – filter array by rejecting
  - `map` – modify each element in the array

Many, many others…

Another Important API

http://ruby-doc.org/core-2.2.0/Array.html

Array Processing

```ruby
a = [1, 3, 4, 7, 8, 10]
a.each { |num| print num } # => 1347810 (no new line)
puts # => (print new line)

new_arr = a.select { |num| num > 4 }
p new_arr # => [7, 8, 10]
new_arr = a.select { |num| num < 10 }
           .reject{ |num| num.even? }
p new_arr # => [1, 3, 7]

# Multiply each element by 3 producing new array
new_arr = a.map {|x| x * 3}
p new_arr # => [3, 9, 12, 21, 24, 30]
```

Summary

- ✧ Arrays API is very **flexible** and **powerful**
- ✧ Lots of ways to **process elements** inside the array

What's next?

- ✧ Ranges

In this lecture, we will discuss…

- ✧ Ranges
- ✧ How they are useful in Ruby

Ranges

- ✧ Used to express natural **consecutive** sequences
  - `1..20`, `'a'.. 'z'`
- ✧ **Two** dots → **all-inclusive**
  - `1..10` (1 is **included**, 10 is **included**)
- ✧ **Three** dots → **end-exclusive**
  - `1...10` (1 is **included**, 10 is **EXCLUDED**)

Ranges

- ✧ **Efficient**
  - Only **start** and **end** stored
- ✧ Can be **converted** to an array with `to_a`
- ✧ Used for **conditions** and **intervals**

Ranges

```ruby
some_range = 1..3
puts some_range.max # => 3
puts some_range.include? 2 # => true

puts (1...10) === 5.3 # => true
puts ('a'...'r') === "r" # => false (end-exclusive)

p ('k'..'z').to_a.sample(2) # => ["k", "w"]
# or another random array with 2 letters in range

age = 55
case age
  when 0..12 then puts "Still a baby"
  when 13..99 then puts "Teenager at heart!"
  else puts "You are getting older..."
end
# => Teenager at heart!
```

Summary

- ✧ Ranges are **useful** for **consecutive sequences**
- ✧ You can **convert** a range to an array for more **functionality**

What's next?

- ✧ Hashes

In this lecture, we will discuss…

- ✧ Hashes
- ✧ **How** they are used
- ✧ **Why** they are used
- ✧ **Similarity** to blocks

Hashes

- **Indexed collections** of object references
- Created with either `{}` or `Hash.new`
- Also known as **associative arrays**
- Index(key) can be **anything**
  - **Not just an integer** as in the case of arrays

Hashes

- Accessed using the `[]` operator
- Values set using
  - `=>` (creation)
  - `[]` (post creation)

Hashes

```ruby
editor_props = { "font" => "Arial", "size" => 12, "color" => "red"}

# THE ABOVE IS NOT A BLOCK - IT'S A HASH
puts editor_props.length # => 3
puts editor_props["font"] # => Arial

editor_props["background"] = "Blue"
editor_props.each_pair do |key, value|
  puts "Key: #{key} value: #{value}"
end
# => Key: font value: Arial
# => Key: size value: 12
# => Key: color value: red
# => Key: background value: Blue
```

Hashes

- What if you try to **access a value** in the Hash for which an entry **does not exist?**
  - `nil` is returned
- If a Hash is created with `Hash.new(0)` `0` is returned instead

Hashes API is also very important to master!

http://ruby-doc.org/core-2.2.0/Hash.html

Hashes

```ruby
word_frequency = Hash.new(0)

sentence = "Chicka chicka boom boom"
sentence.split.each do |word|
  word_frequency[word.downcase] += 1
end

p word_frequency # => {"chicka" => 2, "boom" => 2}
```

More Hashes

- As of **Ruby 1.9**
  - The order of putting things into `Hash` maintained
  - If using symbols as keys – can use `symbol:` syntax

More Hashes

- If a `Hash` is the **last argument** to a method `{}` are optional

Last argument not including a block!

Hashes

```ruby
family_tree_19 = {oldest: "Jim", older: "Joe", younger: "Jack"}
family_tree_19[:youngest] = "Jeremy"
p family_tree_19
# => {:oldest=>"Jim", :older=>"Joe", :younger=>"Jack", :youngest => "Jeremy"}

# Named parameter "like" behavior...
def adjust_colors (props = {foreground: "red", background: "white"})
  puts "Foreground: #{props[:foreground]}" if props[:foreground]
  puts "Background: #{props[:background]}" if props[:background]
end
adjust_colors # => foreground: red
              # => background: white
adjust_colors ({ :foreground => "green" }) # => foreground: green
adjust_colors background: "yella" # => background: yella
adjust_colors :background => "magenta" # => background: magenta
```

Block and Hash Confusion

```ruby
# Let's say you have a Hash
a_hash = { :one => "one" }

# Then, you output it
puts a_hash # => {:one=>"one"}

# If you try to do it in one step - you get a SyntaxError
# puts { :one => "one" }

# RUBY GETS CONFUSED AND THINKS {} IS A BLOCK!!!

# To get around this - you can use parens
puts ({ :one => "one" }) # => {:one=>"one"}

# Or drop the {} altogether...
puts one: "one"# => {:one=>"one"}
```

Summary

- Hashes are **indexed collections**
- Usage is **very similar to regular arrays**
- Although uncommon, can be **confused for blocks**

What's next?

- Classes

In this lecture, we will discuss…

- ✧ Classes
- ✧ How **objects** are **created**
- ✧ How to **access data** within those objects

OO Review

- ✧Identify things your program is **dealing with**
- ✧**Classes** are **things** (*blueprints*)
  - Containers of methods (*behavior*)
- ✧Objects are **instances** of those things
- ✧Objects contain **instance variables** (*state*)

Instance Variables

- ✧ Begin with **@**
  - Example: `@name`
- ✧ Not declared
  - **Spring into existence** when first used
- ✧ **Available to all instance methods** of the class

Object Creation

- ✧ Classes are **factories**
  - Calling `new` method **creates an instance** of class

> `new` causes `initialize`

- ✧ Object's **state** can be (should be) **initialized** inside the `initialize` method, the "constructor"

Object Creation

```ruby
class Person
  def initialize (name, age) # "CONSTRUCTOR"
    @name = name
    @age = age
  end
  def get_info
    @additional_info = "Interesting"
    "Name: #{@name}, age: #{@age}"
  end
end

person1 = Person.new("Joe", 14)
p person1.instance_variables # [:@name, :@age]
puts person1.get_info # => Name: Joe, age: 14
p person1.instance_variables # [:@name, :@age, :@additional_info]
```

Accessing Data

- ✧ Instance variables are **private**
  - **Cannot be accessed** from **outside** the class
- ✧ Methods have **public access** by default
- ✧ To access instance variables – need to define **“getter” / “setter”** methods

Accessing Data

```ruby
class Person
  def initialize (name, age) # "CONSTRUCTOR"
    @name = name
    @age = age
  end
  def name
    @name
  end
  def name= (new_name)
    @name = new_name
  end
end

person1 = Person.new("Joe", 14)
puts person1.name # Joe
person1.name = "Mike"
puts person1.name # Mike
# puts person1.age # undefined method `age' for #<Person:
```

Accessing Data (Continued)

- ✧ Many times the getter/setter logic is **simple**
  - **Get existing** value / **Set new** value
- ✧ There should be an **easier way** instead of actually defining the getter/setter methods…

Accessing Data (Continued)

- Use `attr_*` form instead
  - `attr_accessor` – getter and setter
  - `attr_reader` – getter only
  - `attr_writer` – setter only

Accessing Data (Continued)

```ruby
class Person
  attr_accessor :name, :age # getters and setters for name and age
end

person1 = Person.new
p person1.name # => nil
person1.name = "Mike"
person1.age = 15
puts person1.name # => Mike
puts person1.age # => 15
person1.age = "fifteen"
puts person1.age # => fifteen
```

Accessing Data (Continued)

- **Two problems** with the previous example
  1. Person is in an **uninitialized state** upon creation (without a name or age)
  2. We probably **want to control** the maximum age assigned

Accessing Data (Continued)

Solution: Use a **constructor** and a **more intelligent** age **setter**

But first, we need to talk about `self`…

self

- Inside instance method, `self` refers to the **object itself**
- Usually, using `self` for **calling other methods of the same instance** is **extraneous**

self

- ✧ At other times, using `self` is **required**
  - Ex. - When it could mean a **local variable assignment**
- ✧ Outside instance method definition, `self` refers to the **class itself**

self

```ruby
class Person
  attr_reader :age
  attr_accessor :name

  def initialize (name, ageVar) # CONSTRUCTOR
    @name = name
    self.age = ageVar # call the age= method
    puts age
  end
  def age= (new_age)
    @age = new_age unless new_age > 120
  end
end

person1 = Person.new("Kim", 13) # => 13
puts "My age is #{person1.age}" # => My age is 13
person1.age = 130 # Try to change the age
puts person1.age # => 13 (The setter didn't allow the change)
```

Why do we need `self` here?

Summary

- ✧ Objects are **created** with `new`
- ✧ Use the **short form** for setting/getting data (`attr_`)
- ✧ **Don’t forget** `self` when required

What’s next?

- ✧ Class inheritance and class methods

In this lecture, we will discuss…

- ✧ The “||” operator
- ✧ Class methods and class variables
- ✧ Class inheritance

var = var || something

- || operator **evaluates the left side**
  - If **true** – **returns** it
  - **Else** – **returns** the **right side**
  - `@x = @x || 5` will return `5` the **first** time and `@x` the **next** time

- Short form
  - `@x ||= 5` – same as above

var = var || something

```ruby
class Person
  attr_reader :age
  attr_accessor :name

  def initialize (name, age) # CONSTRUCTOR
    @name = name
    self.age = age # call the age= method
  end
  def age= (new_age)
    @age ||= 5 # default
    @age = new_age unless new_age > 120
  end
end
person1 = Person.new("Kim", 130)
puts person1.age # => 5 (default)
person1.age = 10 # change to 10
puts person1.age # => 10
person1.age = 200 # Try to change to 200
puts person1.age # => 10 (still)
```

Class Methods and Variables

- ✧ **Invoked** on the **class** (as opposed to instance of class)
  - Similar to `static` methods in Java
- ✧ `self` **OUTSIDE** of the **method definition** refers to the `Class` object

Class Methods and Variables

- **Three** ways to **define class methods** in Ruby
- Class variables **begin** with `@@`

Class Methods and Variables

```ruby
class MathFunctions
  def self.double(var) # 1. Using self
    times_called; var * 2;
  end
  class << self # 2. Using << self
    def times_called
      @@times_called ||= 0; @@times_called += 1
    end
  end
end
def MathFunctions.triple(var) # 3. Outside of class
  times_called; var * 3
end

# No instance created!
puts MathFunctions.double 5 # => 10
puts MathFunctions.triple(3) # => 9
puts MathFunctions.times_called # => 3
```

`self` outside of method refers to `Class` object

Class Inheritance

- ✧ Every class **implicitly inherits** from `Object`
  - `Object` itself inherits from `BasicObject`
- ✧ **No** multiple inheritance
  - **Mixins** are used instead

Inheritance

```ruby
class Dog
  def to_s
    "Dog"
  end
  def bark
    "barks loudly"
  end
end
class SmallDog < Dog
  def bark # Override
    "barks quietly"
  end
end

dog = Dog.new # (btw, new is a class method)
small_dog = SmallDog.new
puts "#{dog}1 #{dog.bark}" # => Dog1 barks loudly
puts "#{small_dog}2 #{small_dog.bark}" # => Dog2 barks quietly
```

Implicitly inherits from `Object`

< Denotes inheritance

Summary

- **Class inheritance** lets you **override** parent's behavior
- Class methods **do not** need an instance of object in order to be **called**
- Class variables begin with `@@`

What's next?

- Modules

In this lecture, we will discuss…

- ✧ Modules
  - As Namespaces
  - As Mixins
- ✧ Using built-in Ruby modules, especially `Enumerable`
- ✧ `require_relative`

Module

- **Container** for classes, methods and constants
  - Or other **modules**…
- Like a `Class`, but cannot be **instantiated**
  - `Class` inherits from `Module` and adds `new`

Module

- ✧ Serves **two** main purposes:
  1. Namespace
  2. Mix-in

Module as Namespace

```ruby
module Sports
  class Match
    attr_accessor :score
  end
end

module Patterns
  class Match
    attr_accessor :complete
  end
end

match1 = Sports::Match.new
match1.score = 45; puts match1.score # => 45

match2 = Patterns::Match.new
match2.complete = true; puts match2.complete # => true
```

Note the use of :: operator

Module as Mixin

- ✧ Interfaces in **OO**
  - **Contract** - define what a class **“could”** do
- ✧ Mixins provide a way to **share** (mix-in) ready code among **multiple classes**

You can include built-in modules like `Enumerable` that can do the hard work for you!

Module as Mixin

```ruby
module SayMyName
  attr_accessor :name
  def print_name
    puts "Name: #{@name}"
  end
end

class Person
  include SayMyName
end
class Company
  include SayMyName
end

person = Person.new
person.name = "Joe"
person.print_name # => Name: Joe
company = Company.new
company.name = "Google & Microsoft LLC"
company.print_name # => Name: Google & Microsoft LLC
```

Enumerable Module

- ✧ `map`, `select`, `reject`, `detect` etc.
- ✧ Used by `Array` class and **many others**
- ✧ You can **include it** in your **own class**
- ✧ Provide an **implementation** for `each` method

All the other functionality of `Enumerable` is magically available to you!

Player

```ruby
# name of file - player.rb
class Player

  attr_reader :name, :age, :skill_level

  def initialize (name, age, skill_level)
    @name = name
    @age = age
    @skill_level = skill_level
  end

  def to_s
    "<#{name}: #{skill_level}(SL), #{age}(AGE)>"
  end

end
```

Team

```ruby
# team.rb
class Team
  include Enumerable # LOTS of functionality

  attr_accessor :name, :players
  def initialize (name)
    @name = name
    @players = []
  end
  def add_players (*players) # splat
    @players += players
  end
  def to_s
    "#{@name} team: #{@players.join(", ")}"
  end
  def each
    @players.each { |player| yield player }
  end
end
```

Enumerable in Action

require_relative allows importing other .rb files!!!

```
require_relative 'player'
require_relative 'team'

player1 = Player.new("Bob", 13, 5); player2 = Player.new("Jim", 15, 4.5)
player3 = Player.new("Mike", 21, 5) ; player4 = Player.new("Joe", 14, 5)
player5 = Player.new("Scott", 16, 3)

red_team = Team.new("Red")
red_team.add_players(player1, player2, player3, player4, player5) # (splat)

# select only players between 14 and 20 and reject any player below 4.5 skill-level
elig_players = red_team.select {|player| (14..20) === player.age }
                       .reject {|player| player.skill_level < 4.5}
puts elig_players # => <Jim: 4.5(SL), 15(AGE)>
                  # => <Joe: 5(SL), 14(AGE)>
```

Summary

- Modules allow you to **“mixin” useful code** into other **classes**
- `require_relative` is useful for **including other ruby files** relative to the **current ruby code**

What’s next?

- Scope

In this lecture, we will discuss…

- ✧ **Scope** of variables
- ✧ **Constants**
- ✧ How the **scope** works with **blocks**

Scope

- Methods and classes begin **new scope** for variables
- Outer scope variables **do not** get carried over to the inner scope
- Use `local_variables` method to see which variables are in (and which are not in) the current scope

Scope

```ruby
v1 = "outside"

class MyClass
  def my_method
    # p v1 EXCEPTION THROWN - no such variable exists
    v1 = "inside"
    p v1
    p local_variables
  end
end

p v1 # => outside
obj = MyClass.new
obj.my_method # => inside
              # => [:v1]
p local_variables # => [:v1, :obj]
p self # => main
```

Scope: Constants

- **Constant** is any reference that begins with **uppercase**, including classes and modules
- Constants' scope rules are **different** than variable scope rules
- Inner scope **can see** constants defined in outer scope and **can also override** outer constants
  - Value remains **unchanged** outside!

Scope - Constant

```ruby
module Test
  PI = 3.14
  class Test2
    def what_is_pi
      puts PI
    end
  end
end
Test::Test2.new.what_is_pi # => 3.14
```

```ruby
module MyModule
  MyConstant = 'Outer Constant'
  class MyClass
    puts MyConstant # => Outer Constant
    MyConstant = 'Inner Constant'
    puts MyConstant # => Inner Constant
  end
  puts MyConstant # => Outer Constant
end
```

Remains unchanged outside

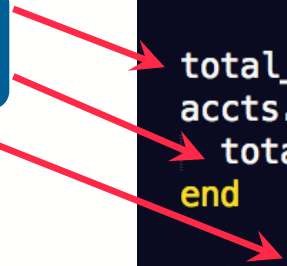
Scope: Block

- Blocks **inherit** outer scope
- Block is a **closure**
  - **Remembers** the context in which it was defined and **uses** that context whenever it is called

Scope - Block

```ruby
class BankAccount
  attr_accessor :id, :amount
  def initialize(id, amount)
    @id = id
    @amount = amount
  end
end

acct1 = BankAccount.new(123, 200)
acct2 = BankAccount.new(321, 100)
acct3 = BankAccount.new(421, -100)
accts = [acct1, acct2, acct3]

total_sum = 0
accts.each do |eachAcct|
  total_sum += eachAcct.amount
end

puts total_sum # => 200
```

Same scope



Block – Local Scope

- Even though blocks **share** the outer scope – a variable created inside the block is **only available** to the block
- **Parameters** to the block are **always local** to the block – **even if** they have the **same name** as variables in the outer scope
- Can **explicitly declare** block-local variables after a **semicolon** in the block parameter list

Block: Local Scope

```ruby
arr = [5, 4, 1]
cur_number = 10
arr.each do |cur_number|
  some_var = 10 # NOT available outside the block
  print cur_number.to_s + " " # => 5 4 1
end
puts # print a blank line
puts cur_number # => 10
```

```ruby
adjustment = 5
arr.each do |cur_number;adjustment|
  adjustment = 10
  print "#{cur_number + adjustment} " # => 15 14 11
end
puts
puts adjustment # => 5 (Not affected by the block)
```

Summary

- ✧ Methods and classes **start** a new scope
- ✧ Constants **maintain** scope
- ✧ Blocks **inherit** outer scope
  - Could be **overridden**

What's next?

- ✧ Access Control

In this lecture, we will discuss…

- ✧ **Three levels** of access control
- ✧ **Controlling** access
- ✧ How **private** is private access?

Access Control

- When **designing** your class – **important** to think about **how much** of it you will be **exposing** to the world
- **Encapsulation**: try to hide the **internal representation** of the object so you **can change it** later
- **Three** levels: `public`, `protected` and `private`

Encapsulation

```ruby
class Car
  def initialize(speed, comfort)
    @rating = speed * comfort
  end

  # Can't SET rating from outside
  def rating
    @rating
  end
end

puts Car.new(4, 5).rating # => 20
```

Details of how rating is calculated are kept inside the class

Specifying Access Control

- **Two** ways to **specify** access control:
  1. **Specify** `public`, `protected` or `private`
     - **Everything** until the **next access control keyword** will be of that access control level
  2. **Define** the methods regularly and then specify `public`, `private`, `protected` access levels and **list** the comma-separated methods **under** those levels using **method symbols**

Specifying Access Control

```ruby
class MyAlgorithm
  private
    def test1
      "Private"
    end
  protected
    def test2
      "Protected"
    end
  public
    def public_again
      "Public"
    end
end
```

```ruby
class Another
  def test1
    "Private, as declared later on"
  end
  private :test1
end
```

Access Control

- `public` methods – **no** access control is enforced
  - **Anybody** can call these methods
- `protected` methods – **can be invoked** by the objects of the defining class or its subclasses
- `private` methods – **cannot be invoked** with an explicit receiver
  - **Exception**: Setting an attribute can be invoked with an explicit receiver

Private Access

```ruby
class Person
  def initialize(age)
    self.age = age # LEGAL - EXCEPTION
    puts my_age
    # puts self.my_age # ILLEGAL
                       # CANNOT USE self or any other receiver
  end

  private
    def my_age
        @age
    end
    def age=(age)
      @age = age
    end
end

Person.new(25) # => 25
```

Summary

- ✧ `public` and `private` access controls used the **most**
- ✧ `private` methods are **not callable** from outside or inside the class with an explicit receiver

What's next?

- Introduction to Unit Testing

In this lecture, we will discuss…

- ✧ Unit testing

Why Write Tests?

- How do you know that your code **works**?
  - You really have **no idea** until you **run it**…
- How do you **refactor with confidence** that you didn't **break anything**?
- Serves as **documentation** for developers

Enter Test::Unit

- Ruby takes testing **very seriously** and ships with Test::Unit
- In Ruby 1.8 – Test::Unit was **bloated with extra libraries** that included unnecessary code
- Ruby 1.9 **stripped** Test::Unit to a minimum

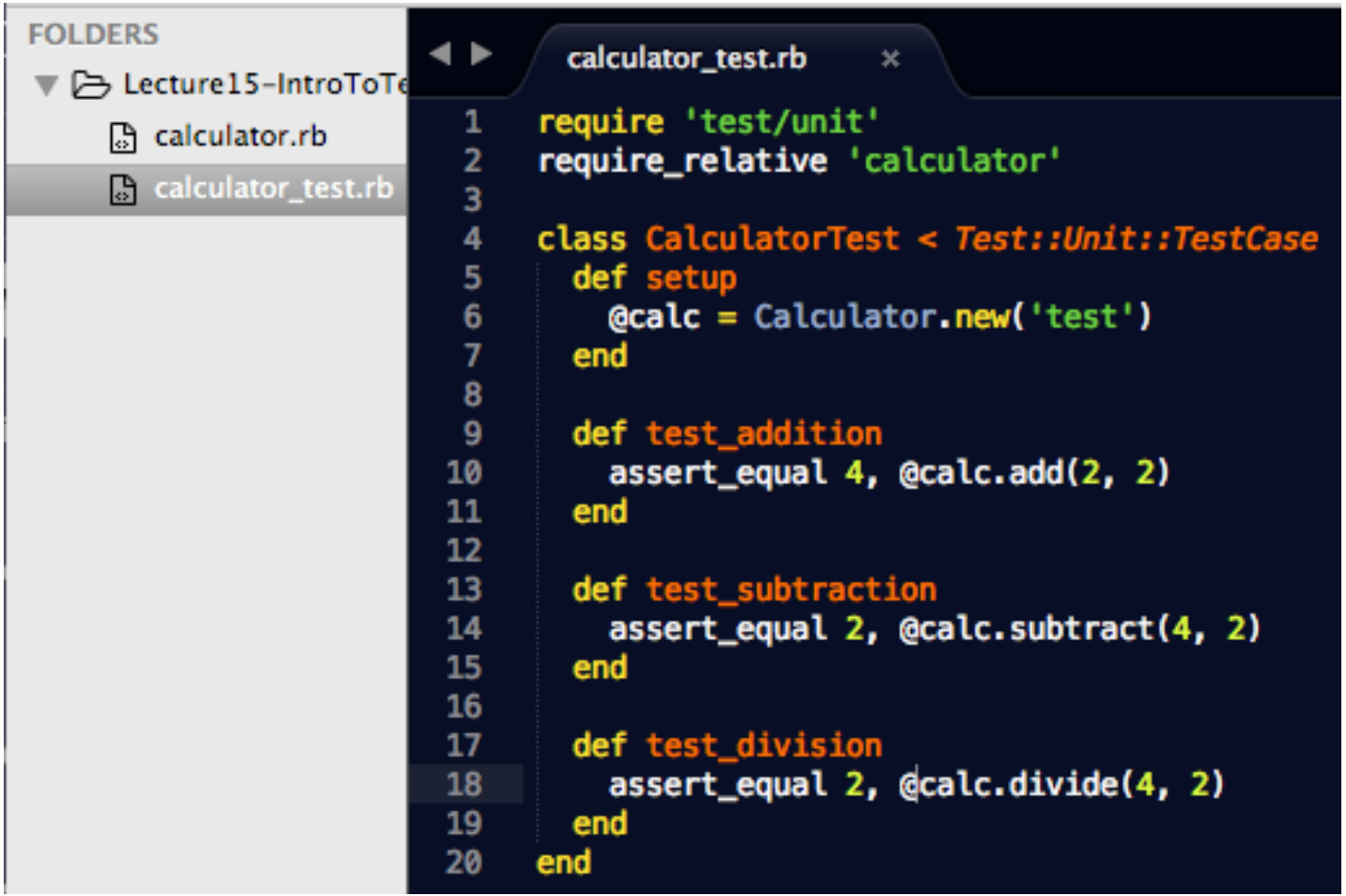
(The new framework is officially called **MiniTest**, but it's a **drop-in replacement**, so **no changes are required** to Test::Unit code)

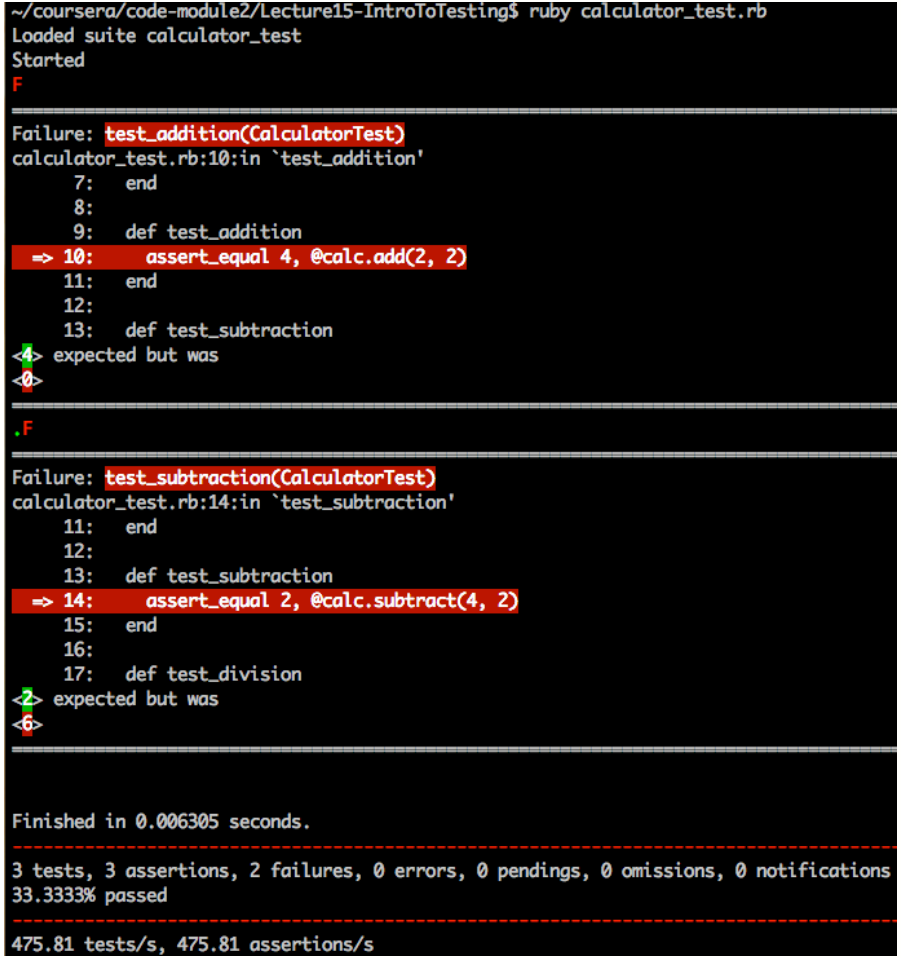
Enter Test::Unit

- Member of the **XUnit** family (JUnit, CppUnit)
- **Basic Idea** – extend Test::Unit::TestCase
- Prefix method names with **test_**
- If one of the methods **fails** – others **keep going** (this is a good thing)
- Can use **setup()** and **teardown()** methods for setting up behavior that will execute before **every** test method

calculator.rb

```ruby
class Calculator

  attr_reader :name

  def initialize(name)
    @name = name
  end

  def add(one, two)
    one - two
  end

  def subtract(one, two)
    one + two
  end

  def divide(one, two)
    one / two
  end
end
```


FOLDERS
Lecture15-IntroToTe
calculator.rb
calculator_test.rb
calculator_test.rb
1 require 'test/unit'
2 require_relative 'calculator'
3
4 class CalculatorTest < Test::Unit::TestCase
5 def setup
6 @calc = Calculator.new('test')
7 end
8
9 def test_addition
10 assert_equal 4, @calc.add(2, 2)
11 end
12
13 def test_subtraction
14 assert_equal 2, @calc.subtract(4, 2)
15 end
16
17 def test_division
18 assert_equal 2, @calc.divide(4, 2)
19 end
20 end


~/coursera/code-module2/Lecture15-IntroToTesting$ ruby calculator_test.rb
Loaded suite calculator_test
Started
F
Failure: test_addition(CalculatorTest)
calculator_test.rb:10:in `test_addition'
7: end
8:
9: def test_addition
=> 10: assert_equal 4, @calc.add(2, 2)
11: end
12:
13: def test_subtraction
<4> expected but was
<0>
.F
Failure: test_subtraction(CalculatorTest)
calculator_test.rb:14:in `test_subtraction'
11: end
12:
13: def test_subtraction
=> 14: assert_equal 2, @calc.subtract(4, 2)
15: end
16:
17: def test_division
<2> expected but was
<6>
Finished in 0.006305 seconds.
3 tests, 3 assertions, 2 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
33.3333% passed
475.81 tests/s, 475.81 assertions/s

Testing Failures

Summary

- ✧ Assertions allow you to exercise your code
- ✧ Unit tests give you confidence to restructure/refactor your code

What's Next?

RSpec

In this lecture, we will discuss…

- ✧ RSpec

Testing With RSpec

- **Test::Unit** “does the job”, but it would be nice if tests would be more **descriptive**, more **English-like**
- The **writing** of the tests is more **intuitive** as well as the **output** from running the tests

Installing RSpec

```
~/coursera/code-module2/Lecture16-RSpec$ gem install rspec
Fetching: rspec-3.3.0.gem (100%)
Successfully installed rspec-3.3.0
Parsing documentation for rspec-3.3.0
Installing ri documentation for rspec-3.3.0
Done installing documentation for rspec after 0 seconds
1 gem installed
~/coursera/code-module2/Lecture16-RSpec$ rspec --init
  create    .rspec
  create    spec/spec_helper.rb
```

Creates a spec directory where “specs” go

describe()

- Set of **related tests** (a.k.a. *example group*)
- Takes either a **String** or **Class** as argument
- All specs **must be inside a describe block**
- No class to subclass
  - Unlike Test::Unit which always subclasses TestCase class

before() and after() methods

- `before()` and `after()` methods are similar to `setup()` and `teardown()` in MiniTest
- Can pass in either `:each` or `:all` (infrequently used) to **specify** whether the block will run **before/after each test** or **once before/after all tests**
- `before :all` **could be useful**, if for example you only want to connect to DB once

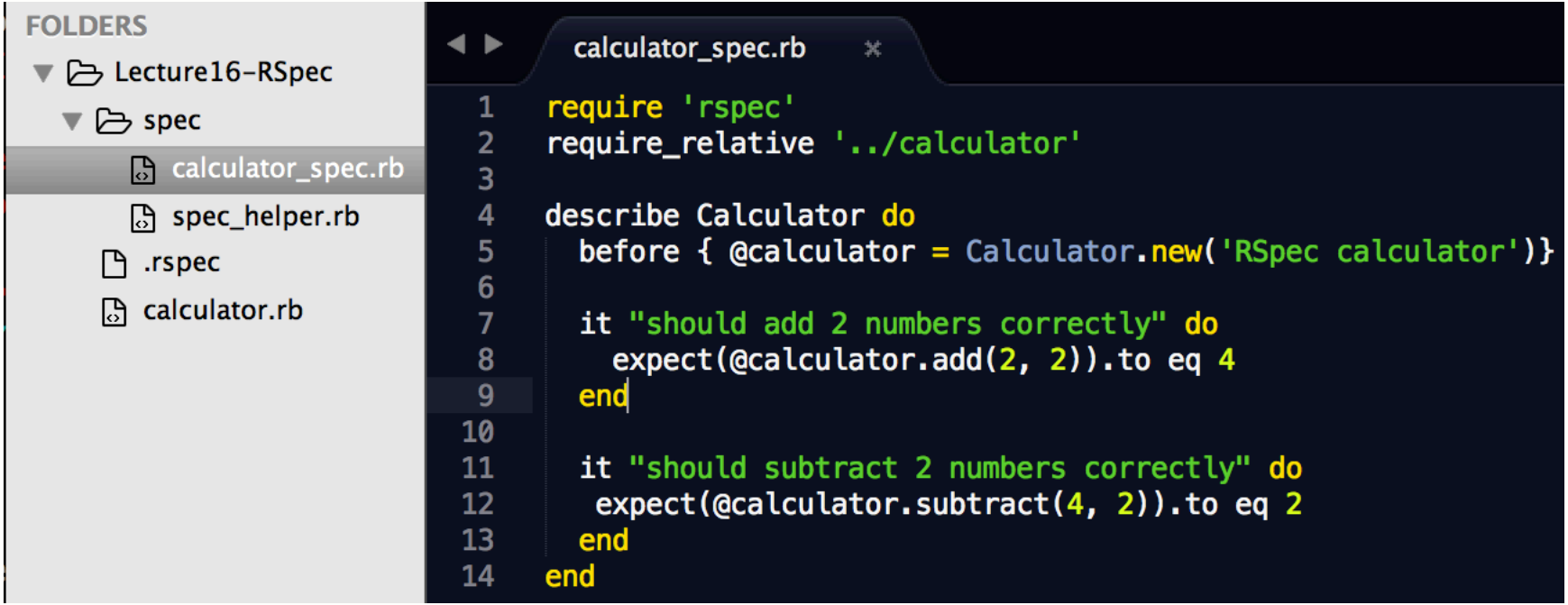
it() method

- Used to define the **actual** RSpec specifications/ examples
- Takes an **optional string** that **describes the behavior** being tested

calculator.rb

```ruby
class Calculator

  attr_reader :name

  def initialize(name)
    @name = name
  end

  def add(one, two)
    one - two
  end

  def subtract(one, two)
    one + two
  end

  def divide(one, two)
    one / two
  end
end
```

calculator_spec.rb



Output

```
~/coursera/code-module2/Lecture16-RSpec$ rspec
FF

Failures:

  1) Calculator should add 2 numbers correctly
     Failure/Error: expect(@calculator.add(2, 2)).to eq 4

       expected: 4
            got: 0

       (compared using ==)
     # ./spec/calculator_spec.rb:8:in `block (2 levels) in <top (required)>'

  2) Calculator should subtract 2 numbers correctly
     Failure/Error: expect(@calculator.subtract(4, 2)).to eq 2

       expected: 2
            got: 6

       (compared using ==)
     # ./spec/calculator_spec.rb:12:in `block (2 levels) in <top (required)>'

Finished in 0.02073 seconds (files took 0.08271 seconds to load)
2 examples, 2 failures

Failed examples:

rspec ./spec/calculator_spec.rb:7 # Calculator should add 2 numbers correctly
rspec ./spec/calculator_spec.rb:11 # Calculator should subtract 2 numbers correctly
```

Summary

- ✧ RSpec makes testing more intuitive

What's next?

- ✧ RSpec Matchers

In this lecture, we will discuss…

- ✧ Rspec Matchers

Rspec Matchers

- RSpec “hangs” **to** and **not_to** methods on **all outcome of expectations**
- to()/not_to() methods take **one parameter** – a **matcher**
- Matcher examples:
  - be_true / be_false
  - eq 3
  - raise_error(SomeError)

Be_predicate – boolean

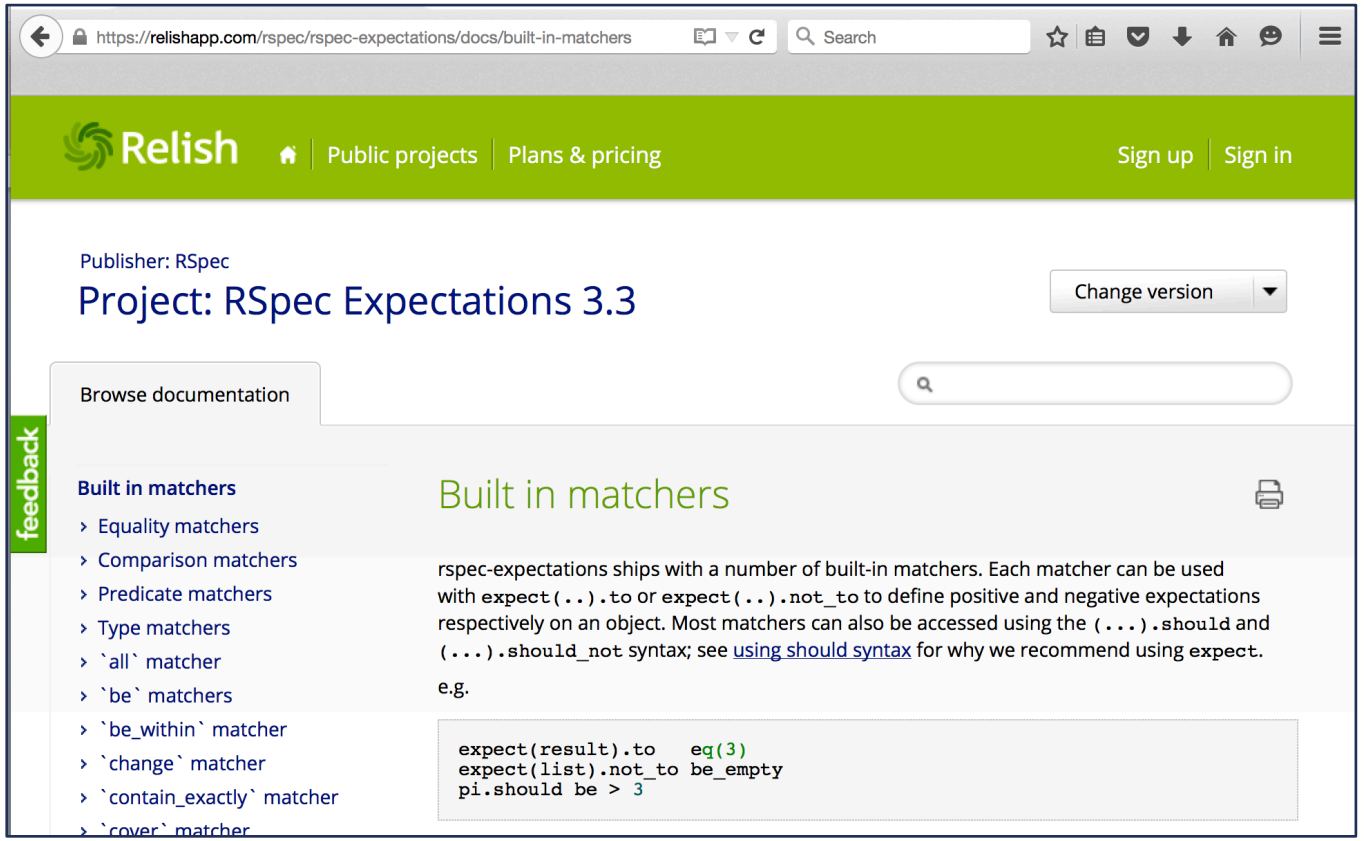
- If the **object** on which the **test is operating** has a **predicate (boolean) method** – you **automatically** get a **be_predicate** matcher
- So, for example **be_nil** is a **valid matcher** since every Ruby object has a :nil? method

```
it "should sum two odd numbers and become even" do
 expect(@calculator.add(3, 3)).to be_even
 expect(@calculator.add(3, 3)).not_to be_odd
end
```

Be_predicate – boolean

```
~/coursera/code-module2/Lecture16-RSpec$ rspec
...

Finished in 0.00169 seconds (files took 0.08464 seconds to load)
3 examples, 0 failures

~/coursera/code-module2/Lecture16-RSpec$ rspec --format documentation

Calculator
  should add 2 numbers correctly
  should subtract 2 numbers correctly
  should sum two odd numbers and become even

Finished in 0.00187 seconds (files took 0.08772 seconds to load)
3 examples, 0 failures
```

More Matchers



Summary

- RSpec has a lot of built-in matchers readily available for simplifying writing tests

What's next?

- Module 3: Introduction to Ruby on Rails