Integrallis

Ruby/Ruby on Rails

Full Steam Ahead

by Chris Irish

Overview
Part 1 - Ruby

In **Part I** you'll be (re)introduced to the Ruby Programming Language

Objectives

- To gain a deeper of **understanding** of **Ruby**
- To **understand** how **Rails** implements some of its **"magic"**
- To become comfortable with the **common tools** used by Rubyists
- To learn and embrace the **practices** of successful Ruby/Rails developers

Overview
Part 1 - Ruby

Part I at a glance:

- The Rubyist Toolset
- Ruby: Laying the Tracks
- IRB: Ruby's Learning Lab
- Ruby Programs Primer
- Requiring & Loading
- Classes & Objects in Depth
- Data Structures
- Lambdas, Procs & Blocks: Portable Code
- Inheritance: OO in Ruby
- RubyGems
- Mixins
- TDD w/ RSpec
- Metaprogramming

Material Conventions
Part 1 - Ruby

- Each training day consists of a mix of lecture time, discussion time and Q & A, guided exercises and labs

- For the guided exercises you will see a green “follow along” sign on the slides

- For the labs you’ll see an orange sign with the lab number

Rubyist Toolset

Laying the Tracks

Rubyist Toolset
Setting up your local Environment

Let's do a sanity check of our environment...

Start by checking that Git is installed

```
/>git --version
git version 1.7.10.2 (Apple Git-33)
```

Rubyist Toolset
Setting up your local Environment

We'll also check that RVM is installed ...

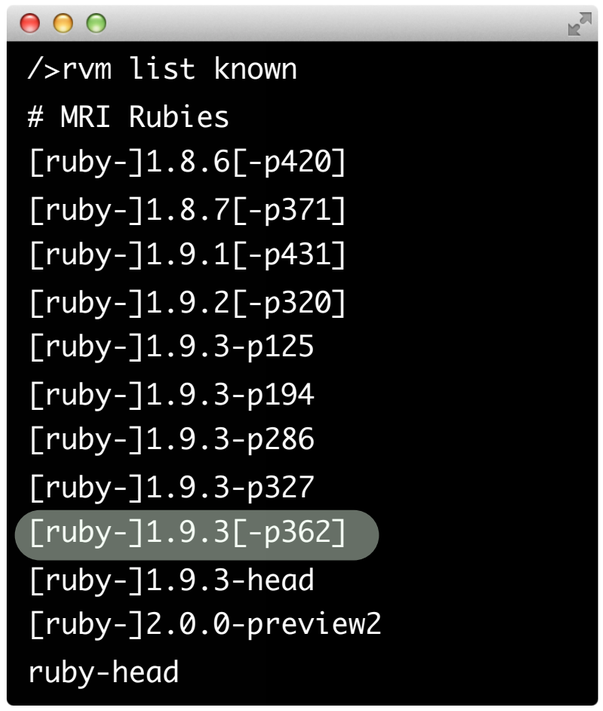
```
/>rvm -v
rvm 1.17.6 (stable) ...

/>rvm info
...
rvm:
    version:      "rvm 1.17.6 (stable) ..."
    updated:      "25 days 21 hours 51 minutes 6 seconds ago"
...
```

Rubyist Toolset
Setting up your local Environment

With RVM in place we check to see that we have Ruby 1.9.3 installed:

```
/>rvm list

rvm rubies

   jruby-1.7.0 [ x86_64 ]
   jruby-1.7.1 [ x86_64 ]
   ruby-1.9.2-p320 [ x86_64 ]
=* ruby-1.9.3-p362 [ x86_64 ]

# => - current
# =* - current && default
#  * - default
```

Rubyist Toolset
Setting up your local Environment



- If Ruby 1.9.3 was not found we'll have to install it via RVM, we'll start by listing the Rubies known to RVM
- To check if our list or Rubies is up to date we can check **ruby-lang.org/en/downloads** for the latest version
- At the time of this writing the latest version is **ruby-1.9.3-p374**
- The latest version shown on the left is **-p362** so we need to refresh RVM's known Rubies list with the command: **rvm get stable**

With RVM list of Rubies up to date we can proceed to install Ruby 1.9.3:

```
/>rvm install 1.9.3
No binary rubies available for: downloads/ruby-1.9.3-p374.
Continuing with compilation.
Please read 'rvm mount' to get more information on binary rubies.
Installing Ruby from source to: ~/.rvm/rubies/ruby-1.9.3-p374, this may take a while depending on your cpu(s)...
ruby-1.9.3-p374 - #downloading ruby-1.9.3-p374, this may take a while depending on your connection...
ruby-1.9.3-p374 - #extracting ruby-1.9.3-p374 to ~/.rvm/src/ruby-1.9.3-p374
```

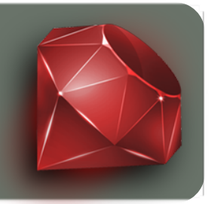
Rubyist Toolset
Setting up your local Environment

Below we can see that we now have the latest Ruby 1.9.3 installed

```
/>rvm use 1.9.3
/>rvm list

rvm rubies

   jruby-1.7.0 [ x86_64 ]
   jruby-1.7.1 [ x86_64 ]
   ruby-1.9.2-p320 [ x86_64 ]
 * ruby-1.9.3-p362 [ x86_64 ]
=> ruby-1.9.3-p374 [ x86_64 ]

# => - current
# =* - current && default
#  * - default
```

- Noticed that the older version (-p362) is still the default Ruby
- To check make the newly installed Ruby the default we use the command: **rvm --default use 1.9.3**

Rubyist Toolset
Setting up your local Environment



We can now check our Ruby version...

```
/>rvm use 1.9.3
Using ~/.rvm/gems/ruby-1.9.3-p374
/>ruby -v
ruby 1.9.3p374 (2013-01-15 revision 38858)
[x86_64-darwin12.2.1]
```

For this course any 1.9.3 release should work

Rubyist Toolset
Setting up your local Environment

We can install Rails while we're at it, so that we are ready for Part 2

```
/>gem install rails
```

The above command should install Rails 3.2.11

Rubyist Toolset
The Command Line

Posix

Text editor

Git - Version control

RVM - Ruby Version Manager

Rubygems - Ruby packages

Bundler - Dependency management

Rubyist Toolset
Setting up your local Environment

Efficient Ruby/Rails Developers are masters of their Shell

On the Mac we recommend iTerm 2

http://www.iterm2.com

On Windows you can install Cygwin

https://cygwin.com/

If you are a Linux user you probably already have a favorite

Git
Version Control

Created by Linus Torvalds

http://git-scm.com/

Extremely powerful

Distributed workflow

Much more approachable than earlier versions

"Merge early, merge often"

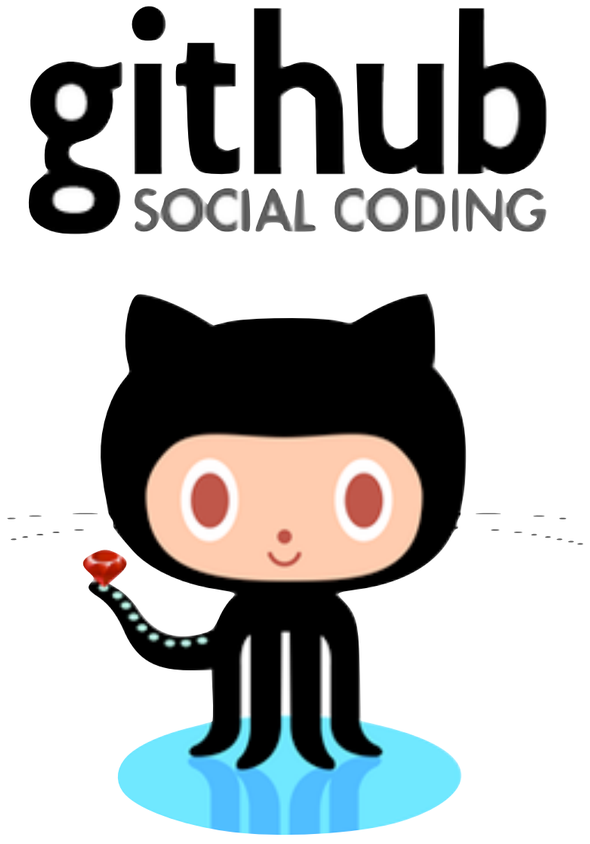
Linus Torvalds

http://www.youtube.com/watch?v=4XpnKHJAok8

Our friendly subset of Git

```
$ git init
$ git add .
$ git status
$ git diff --cached
$ git commit -m "Add x to y"
$ git checkout -b my-feature-branch # new branch
$ git merge my-feature-branch
```

GitHub
Cloud Hosted Version Control



http://github.com/

Code hosting

Collaboration tools

Issue tracking

Wikis

Code review

RVM
Ruby Version Manager

https://rvm.io

Huge positive impact on Ruby developer efficiency

Sandbox dependencies with "gemsets"

Declare project specific configuration in `.rvmrc`

An alternative is rbenv

https://github.com/sstephenson/rbenv)

Our subset of RVM

```
$ rvm get stable # get the latest stable
release
$ echo "rvm use 1.9.3@my_project --create"
> .rvmrc
$ rvm reload
```

RubyGems
Ruby Package Manager

Package manager for Ruby

https://rubygems.org

Provides an executable called `gem`

rubygems.org is an open-source Rails app

https://github.com/rubygems/rubygems.org)

Command reference

http://guides.rubygems.org/command-reference/

Bundler
Managing Application Dependencies

Dependency management for Ruby gems

http://gembundler.com/

Packaged as a Ruby gem

Includes an executable for managing project dependencies

Expects a Gemfile in the root of project

Outputs Gemfile.lock

Gemfile
Declaring Dependencies

```
source :rubygems
```

```
gem 'rest-client', '>= 1.0.4', '< 2'
```

```
gem 'sqlite3', :group => [:development, :test]
# or
group :test do
  gem 'rspec-rails'
  gem 'capybara'
end
```

Using Gem Groups

```
require "bundler"
Bundler.require(:default, MyProject.env)
```

Ruby
Laying the Tracks

Ruby
Some Facts About Ruby

✓ Object-Oriented

✓ Elegant

✓ General Purpose

Ruby

is...

✓ Multi-paradigm

✓ Interpreted

✓ Reflective

✓ Dynamic

✓ Garbage-collected

Ruby
Some Facts About Ruby

- Created by Yukihiro “Matz” Matsumoto
- Developed in 1993, released in 1995
- Blended parts of Matz favorite languages
  - Smalltalk, Lisp, Perl, Sather, Eiffel, CLU and Ada
- More popular than Python in Japan

Ruby
Some Facts About Ruby

- Created by Yukihiro “Matz” Matsumoto
- Developed in 1993, released in 1995
- Blended parts of Matz favorite languages
  - Smalltalk, Lisp, Perl, Sather, Eiffel, CLU and Ada
- More popular than Python in Japan

Ruby
A Language of Careful Balance

- "There is more than one way to do it" (TIMTOWTDO)
- "Everything is an object" (almost)

"Ruby is simple in appearance, but is very complex inside, just like our human body"

Yukihiro "Matz" Matsumoto
Heroku's Chief Architect & Ruby Language Creator

Ruby
Some Facts About Ruby

"I believe people want to express themselves when they program. They don't want to fight with the language. Programming languages must feel natural to programmers"

Yukihiro "Matz" Matsumoto
Heroku's Chief Architect & Ruby Language Creator

Ruby
Some Facts About Ruby

- Ruby Ideals:
  - Programming should be fun!
  - The language should treat you like a responsible adult!
  - Balanced functional programming with imperative programming
  - "There is more than one way to do it" (TIMTOWTDO)
  - "Everything is an object" (almost)

IRB

Ruby's Learning Lab

Ruby Quick Start
Using the Interactive Ruby Shell

In this section, students will get acquainted with the Ruby language using the Interactive Ruby shell (IRB)

- IRB is a command-line shell where you can evaluate Ruby statements
- IRB allows you to experiment with Ruby in real-time
- For most Rubyists IRB is where we learn how to use a feature of the language or learn how to use a new library

Ruby Quick Start
Using the Interactive Ruby Shell

Follow Along

To launch IRB at the command line type: **irb**

```
/>irb
1.9.3p362 :001 >
```

IRB's command-line options can be found at:
http://ruby-doc.org/docs/ProgrammingRuby/html/irb.html

Ruby Quick Start
Using the Interactive Ruby Shell

Follow Along

We can check the current Ruby version within IRB by printing to the console the value of the RUBY_VERSION constant

```
/> irb
> puts RUBY_VERSION
1.9.3
 => nil
```

You probably seen the puts method before, it's part of the Class IO (http://www.ruby-doc.org/core-1.9.3/IO.html#method-i-puts)

Ruby Quick Start
Using the Interactive Ruby Shell

Follow Along

- As you can see in IRB (or in any Ruby script) we can invoke a method (or more accurately we can send a message) as long as we have a receiver

- In the previous example the receiver is the “self” object which at the top-level execution context of a Ruby program is the “main” object

```
/> irb
> self
 => main
> RUBY_VERSION.class
 => String
```

Ruby Quick Start
Using the Interactive Ruby Shell

Let's try some simple examples on IRB:

```
> 1+2
 => 3
> "rUBY rOCKS!".swapcase
 => "Ruby Rocks!"
```

IRB is a shell in the family of read-eval-print-loop (REPL) environments that originated with LISP

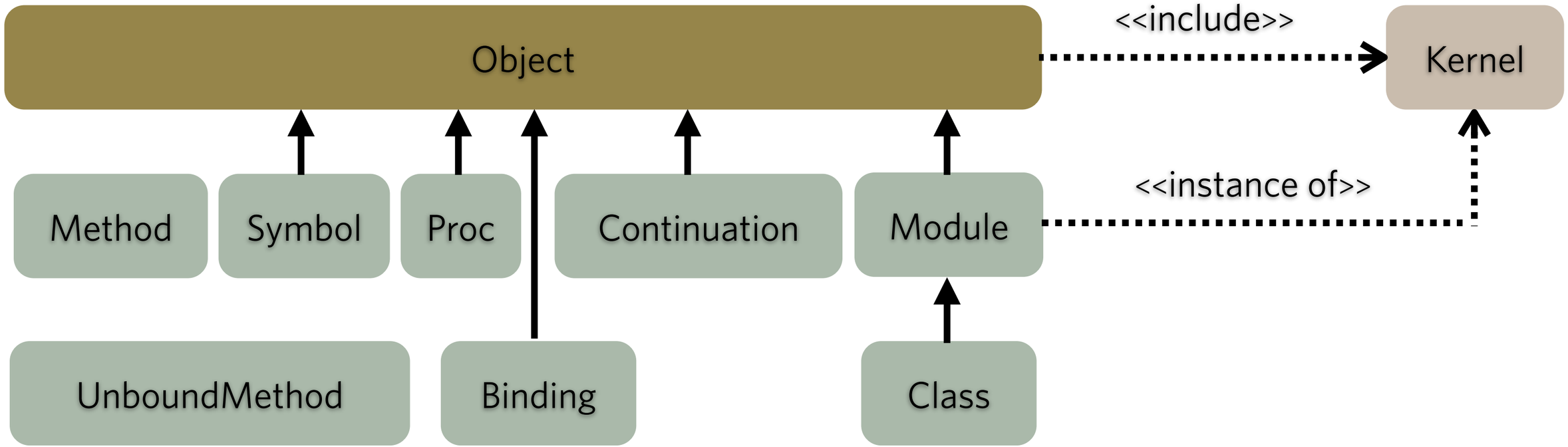
Ruby Quick Start
Using the Interactive Ruby Shell

- Ruby is an Object-Oriented language
- Ruby class hierarchy is single rooted
- At the top of the hierarchy is the **Object** class

```
1.9.3p362 :007 > (1+2).class
 => Fixnum
1.9.3p362 :008 > "Ruby".class
 => String
1.9.3p362 :009 > self.class
 => Object
```

Ruby's Class Hierarchy

The Basics



Ruby Quick Start

Using the Interactive Ruby Shell

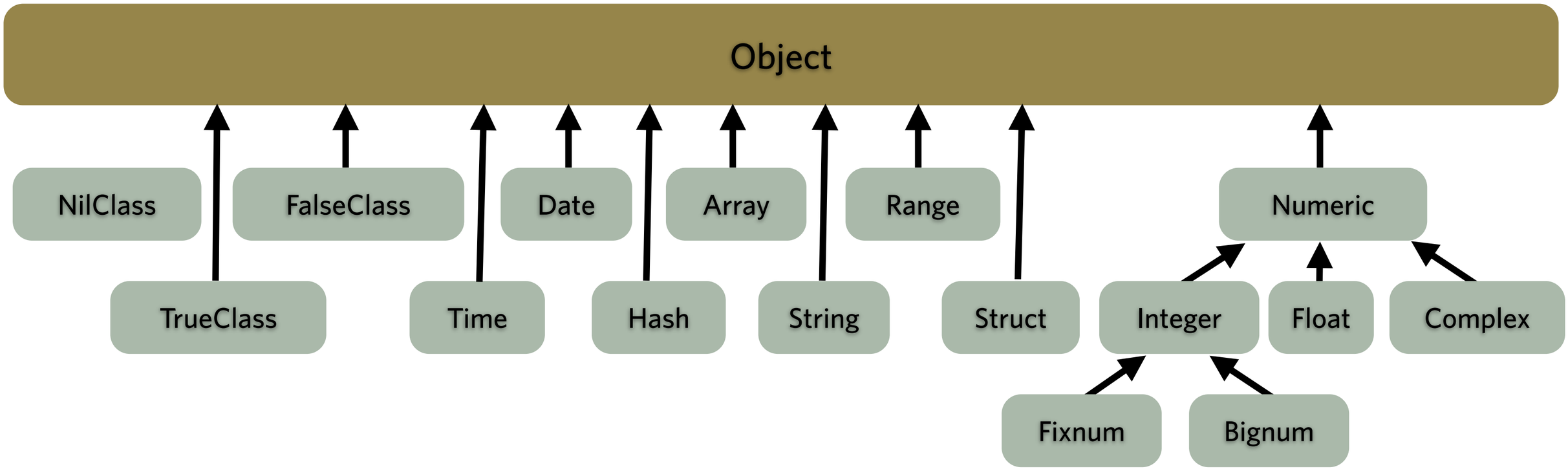
Let's try some not so simple examples on IRB:

```
> 15.class.ancestors
 => [Fixnum, Integer, Numeric, Comparable, Object, Kernel, BasicObject]
> ["lions", "tigers", "bears"].join(" and ") + ", oh my!"
 => "lions and tigers and bears, oh my!"
> gimme = lambda { |item| "Please pass me the #{item}." }
 => #<Proc:0x007fb2a203ac08@(irb):38 (lambda)>
> gimme.call("salt and pepper")
 => "Please pass me the salt and pepper."
```

Don't worry if you don't know what all the above statements do, soon you will!

Ruby's Data Types

The Basics



Programs
Ruby Programs

Ruby Quick Start
Ruby Programs

- Ruby programs are interpreted in a top-down fashion
- A Ruby program doesn't need an explicit entry point
- Ruby programs are usually stored with the .rb extension (the .ru extension is sometimes used by is falling out of style)
- A Ruby program can consist of multiple Class and Module declarations as well as free-standing statements

A Simple Program
Class-less Greeter

- Using a simple text editor, create the simple ruby program shown below
- Save the file as **hello.rb**

```
def greet
  puts "Hello!"
end

greet
```

hello.rb

A Simple Program
Class-less Greeter

- On the command line, change directories to the location of the hello.rb Ruby program and type:

```
/>ruby hello.rb
Hello!
```

- Where does the greet method live? Let's try adding the following lines at the bottom of hello.rb to find out...

```
puts self.methods.include?(:greet)
puts self.class.methods.include?(:greet)
puts self.class.private_methods.include?(:greet)
```

```
false
false
true
```

A Simple Program
Class-less Greeter

Follow Along

- We find it useful to think of communication between objects as **message passing** rather than function calls

- In **hello.rb** let make some changes to reveal the underlying mechanics of method invocation tin Ruby

```ruby
def greet
  puts "Hello!"
end

self.send(:greet)
```

hello.rb

Execute the example from the command line. The output should be identical as the previous version

A Simple Program
Class-less Greeter

- In Ruby we try to think about sending a message to an object rather than calling a method on an object, a subtle but significant difference when it comes to the understanding of the language:

```
p "yo!" #=> "yo!"
p String.new("yo, yo!") #=> "yo, yo!"
p String.send(:new, "yo, yo, ma!") #=> "yo, yo, ma!"
```

message_sending.rb

- In simples cases (like with constructing a String), Ruby provides syntactic sugar to hide verbosity

Error Handling
rescue & raise

- Without specifying an Error Class to rescue from, rescue will rescue from StandardError and any of its descendants

```
begin
   1 + '2'
rescue => e
  puts e.message
end

def times_2(val)
  raise ArgumentError.new("'#{val}' is a string :(") if val.is_a?(String)

  val * 2
rescue => e
  puts e
end

times_2("hello")
```

Error Handling
rescue & raise

- You can rescue from a specific error class:

```
MyCustomError = Class.new(StandardError)

begin
  raise StandardError
rescue MyCustomError => e
  # never gets here
  # will not rescue StandardError
end
```

Requiring & Loading
Composing a System

Composing a System
Requiring & Loading

- Up to this point we have dealt with a single Ruby program that had no external dependencies
- A Ruby program (file) can use code contained in other Ruby programs (files)
- The two main means to use an external Ruby program are the `load` and `require` (and the new `require_relative`) methods

Kernel#load
Loading a Ruby File

- The `Kernel` module contains the `load` method
- We learned that the `Kernel` module is mixed-in or included in the `Object` class therefore making it globally available
- The `load` method "loads" the contents of another Ruby program in place
- At the location of the `load` statement, the code is read in and interpreted

Composing with Load

Loading a Module

- To test how to compose Ruby programs with the `load` method you will create your first Ruby Module containing some reusable code

```ruby
module Financial
  puts "Financial Module loading..."

  def self.calculate_payment(principal, interest, years)
    ip = (interest / 100) / 12
    n = years * 12
    (-principal + (-principal / ((1 + ip) ** n) - 1)) * -ip
  end

  puts "Done loading Financial Module!"
end
```

financial.rb

Composing with Load

Loading a Module

- Next we need to write a Ruby program to load and use the Financial module, we will call this program **payment_calculator.rb**

```
load 'financial.rb'

puts "Welcome to the payment calculator..."
puts "Enter the principal:"
principal = gets.to_f
puts "Enter the interest:"
interest = gets.to_f
puts "Enter the terms in years:"
years = gets.to_i
payment = Financial.calculate_payment(principal, interest, years)
puts "For a principal of $#{principal} with an interest of #{interest}%
for #{years} years they payments will be #{payment}"
```

payment_calculator.rb

Composing with Load

Loading a Module

- On the command line run the **payment_calculator.rb** program:

```
/>ruby payment_calculator.rb
Financial Module loading...
Done loading Financial Module!
Welcome to the payment calculator...
Enter the principal:
50000
Enter the interest:
8.5
Enter the terms in years:
15
For a principal of $50000.0 with an interest of 8.5% for 15 years
they payments will be 453.58467588580226
```

Kernel#require

Requiring a Ruby File

- The `Kernel` module also contains the `require` method
- The `require` method, just as the `load` method does - loads the contents of another Ruby program in place
- It is used just like the load method with the difference being that require does not need the extension of the file being required
- Both `load` and `require` use the ruby load path; a set of directories that ruby contains (and that you can modify if necessary)

Kernel#require
Requiring a Ruby File

- The main difference is that require would NOT reload an already loaded Ruby script
- The load method should be used when changes in the loaded script want to be processed in a reentrant environment
- Rails uses load in development mode to dynamically load any changes to the user code on the next request-response cycle
- Ruby also provides the `require_relative` method which allows you to load a file relative to the file containing the require_relative statement

Classes & Objects

Object-Orientation in Ruby

Classes & Objects
Working with Classes

- Up to this point we have worked with some Ruby objects, invoked methods, we've mixed in a Module into a Ruby program and created our own methods

- Ruby is class-based language. Ruby classes are the recipes or templates used to make objects

- In this section we'll learn how to define classes, add methods, instantiate them and perform some work

Classes & Objects
Working with Classes

Follow Along

- We'll start by creating a simple Ruby program called **greeter.rb**
- The program will define one class called Greeter

```ruby
class Greeter
  def say_hello(name)
    result = "Hi, " + name
    return result
  end
end

greeter = Greeter.new
puts greeter.say_hello("Matz")
```

greeter.rb

The Greeter class, although it is perfectly valid Ruby, is not very idiomatic and ignores what many Rubyists refer to as "The Ruby Way". In the next few sections we'll see how to achieve "The Ruby Way"

- We'll start the refactoring by removing the return statement since in Ruby, the result of the last expression becomes the return value of the method

```ruby
class Greeter
  def say_hello(name)
    result = "Hi, " + name
    result
  end
end

greeter = Greeter.new
puts greeter.say_hello("Matz")
```

greeter.rb

Classes & Objects
Working with Classes

- Since the assignment statement also returns the value assigned we can remove the last line in the method

```ruby
class Greeter
  def say_hello(name)
    result = "Hi, " + name
  end
end

greeter = Greeter.new
puts greeter.say_hello("Matz")
```

greeter.rb

Classes & Objects
Working with Classes

Follow Along

- We can remove the local variable “result” since the result of an assignment is the value:

```ruby
class Greeter
  def say_hello(name)
    "Hi, " + name
  end
end

greeter = Greeter.new
puts greeter.say_hello("Matz")
```

greeter.rb

Classes & Objects
Working with Classes

Follow Along

- Ruby Strings have powerful interpolation capabilities that help keep Ruby code clean:

```ruby
class Greeter
  def say_hello(name)
    "Hi, #{name}"
  end
end

greeter = Greeter.new
puts greeter.say_hello("Matz")
```

greeter.rb

Self
The Current/Default Object

- Let's talk about **self**! Self is the current/default object
- At every point in your program there is one and only one self
- The object that self points to changes based on the execution context

Self
The Current/Default Object

- Let's start at the *Top Level* of a program...

discovering_self.rb

```
puts "Here at the Top Level..."
puts "self is #{self}"
puts "self's class is #{self.class}"
```

```
/>ruby discovering_self.rb
Here at the Top Level...
self is main
self's class is Object
```

- At the Top Level, self is an object named main of class Object

Self
The Current/Default Object

- Now let's add a class definition and see what self points to inside of it...

discovering_self.rb

```
puts "Here at the Top Level..."
puts "self is #{self}"
puts "self's class is #{self.class}"

class OuterMost
  puts "Inside the OuterMost class, self is #{self}"
end
```

```
/>ruby discovering_self.rb
Here at the Top Level...
self is main
Inside the class, self is OuterMost
```

Self
The Current/Default Object

- Let's nest a class inside OuterMost and see what we get ...

discovering_self.rb

```
...
class OuterMost
  puts "Inside the OuterMost class, self is #{self}"
  class InnerOne
    puts "Inside the InnerOne class, self is #{self}"
  end
end
```

```
/>ruby discovering_self.rb
...
Inside the OuterMost class, self is OuterMost
Inside the InnerOne class, self is OuterMost::InnerOne
```

Self
The Current/Default Object

- Using the **def** in combination with the **self** object allow us to add **methods** to that object

- Since the **self** at this level is the **class**, we've effectively added a **class method** (singleton method on the class object)

discovering_self.rb

```
class OuterMost
  ...
  def self.foo
    puts "In Class method OuterMost#foo, self is #{self}"
  end
end

OuterMost.foo
```

```
/>ruby discovering_self.rb
...
In Class method OuterMost#foo, self is OuterMost
```

Self
The Current/Default Object

- Now, let's add an instance method to the **OuterMost** class
- Using **def** without an explicit receiver inside of a **class** creates an **instance method** inside it self points to the object

discovering_self.rb

```
class OuterMost
  ...
  def bar
    puts "In Instance method OuterMost#bar, self is #{self}"
  end
end

an_outer_most = OuterMost.new
an_outer_most.bar
```

```
/>ruby discovering_self.rb
...
In Instace method OuterMost#bar, self is #<OuterMost:0x007fe4e8902de0>
```

Ruby Classes
Working with Classes

- As we seen in previous examples, the **new** method returns an instance of the class that it invoked against

- By now you probably noticed that parenthesis are optional on a method invocation

```
class Person
end

p = Person.new

puts p #=> #<Person:0x007fa7098ad718>
puts p.class # Person
```

Ruby Classes
Initialization

- Ruby doesn't have formal constructors like Java or C++
- Instead Ruby provides the **initialize** method, invoked after a call to **new**

```
class Person
  def initialize(name)
    @name = name
  end

  def name
    @name
  end
end

p = Person.new("Michael")
puts p.name #=> "Michael"
```

Ruby Classes
Attribute Reader

- Ruby's **attr_reader** class method takes a symbol and provides the enclosing class instances with an instance variable and a reader (getter) method both named after the symbol
- Instance variables are prefixed with @ symbol and are only accessible inside instance methods*

```
class Person
  attr_reader :name

  def initialize(name)
    @name = name
  end
end

p = Person.new("Michael")
puts p.name #=> "Michael"
```

Ruby Classes
Attribute Writter

- Ruby's **attr_writer** class method provides a writer (setter) method:

```ruby
class Person
  attr_reader :name
  attr_writer :name

  def initialize(name)
    @name = name
  end
end

p = Person.new("Michael")
puts p.name #=> "Michael"
p.name = "Michael Scott"
puts p.name #=> "Michael Scott"
```

Ruby Classes
Attribute Writter

- When both **attr_reader** and **attr_writer** are required you can instead use **attr_accessor**:

```ruby
class Person
  attr_accessor :name

  def initialize(name)
    @name = name
  end
end

p = Person.new("Michael")
puts p.name #=> "Michael"
p.name = "Michael Scott"
puts p.name #=> "Michael Scott"
```

Ruby Classes
Instance Methods

- Instance methods like **intro** are invoked on an instance:

```ruby
class Person
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  def intro
    "Hello, my name is #{name}"
  end
end

p = Person.new("Michael")
puts p.intro #=> "Hello, my name is Michael"
```

Singleton Object
Instance Specific Behaviors

- In Ruby you can enhance a particular instance of a class
- Ruby uses a proxy object known as the singleton
- The singleton object for a Class object is typically referred as the meta-class
- In the Metaprogramming section of the Course we'll learn more about singletons

Ruby Classes
Singleton Methods

- Singleton methods can be added to any object using dot notation:

```ruby
class Person
  attr_accessor :name

  def initialize(name); @name = name; end
end

mike = Person.new("Michael")
pam = Person.new("Pam")

def mike.speak(phrase)
  puts phrase
end

mike.speak("Hello everybody.") #=> "Hello everybody."
pam.speak("Hello Michael.") #=> NoMethodError: undefined method 'speak'
```

Ruby Classes
Method Lookup

- The frozen? method is defined in the Object class
- The Object class is the superclass or parent class of String making the method available to all instances of the class String

```
str = "lorem ipsum"

# frozen? is not defined on the class String
# It's defined on the superclass of String, the Object class
puts str.frozen?
```

Ruby Classes
Class Names as Constants

- A loaded class name becomes a constant, we can use the `define?` method with any Ruby expression to tests whether is defined and what kind of expression it is (literal object, local variable, method or in the case below a constant)

```
class Person
end

puts defined?(Person) #=> constant
puts Person.class #=> Class
```

Ruby Classes
Singleton Methods

- Previously we defined class methods directly inside the class definition
- Ruby's classes are open so we can use dot (.) notation outside of the class definition to add a class method

```
class Person
end

def Person.intro
  puts "I am the Person class"
end

Person.intro #=> "I am the Person class"
```

Ruby Classes
class << obj

- The << operator is overloaded when used with the class keyword and an object
- The effect is that of opening the passed object **singleton** class
- Methods defined in the block are available only to the passed object

```
str = "lorem ipsum dolor"
otherstr = "my other string"

# Another way of defining singleton methods
class << str
  def shuffle_words
    split.shuffle.join(' ')
  end
end

puts str.shuffle_words #=> "ipsum dolor lorem"
puts otherstr.shuffle_words #=> NoMethodError: undefined method 'shuffle_words'
```

Ruby Classes
class << obj

- Since classes are instances of the Class class the code below uses the technique previously outlined to add a class method to the Person object (which is a object of class Class :-)

```
class Person
end

class << Person
  def intro
    puts "I am the Person class"
  end
end

Person.intro #=> "I am the Person class"
```

Ruby Classes
Class Methods

Follow Along

- Inside of a class definition we've learn that self point to the class object, therefore adding a class method to the Person class (object)
- By now, Ruby's "Almost Everything is an Object"* should be becoming more clear

```ruby
class Person
  class << self
    def intro
      puts "I'm still the Person class!!!"
    end
  end
end

Person.intro #=> "I'm still the Person class!!!"
```

Ruby Classes
Class Methods

- Singleton attributes (or class attributes) can be created using the class << object idiom:

```
class Vehicle
  class << self
    attr_accessor :api_endpoint
  end
end

Vehicle.api_endpoint = "vehicle-api.example.com"
puts Vehicle.api_endpoint
```

Ruby Classes
Cloning and Duping

- We can duplicate objects with the **clone** and **dup** methods

```ruby
class Vehicle
  def initialize(make, model)
    @make = make
    @model = model
  end
end

civic = Vehicle.new("Honda", "Civic")

def civic.title
  "#{@make} #{@model}"
end

puts "Original title: #{civic.title}" #=> "Original title: Honda Civic"

cloned_civic = civic.clone
duped_civic = civic.dup

puts cloned_civic.title #=> "Honda Civic"
puts duped_civic.title #=> NoMethodError: undefined method 'title'
```

Ruby Classes
Method Arity

- Method declarations take a formal arguments list:

```
def comma_separate(a, b)
  a + ", " + b
end

puts comma_separate("foo","bar")

begin
  puts comma_separate("foo", "bar", "baz")
  #=> ArgumentError: wrong number of arguments (3 for 2)
rescue ArgumentError => e
  puts e.message
end
```

Ruby Classes
Method Arity & Splat Args

- The Splat(*) as part of the last method argument allows you gather up any remaining arguments

```ruby
def other_comma_separate(a, *b)
  a + ", " + b.join(", ")
end

puts other_comma_separate("p", "q", "r", "s", "t")
#=> p, q, r, s, t

def best_comma_separate(*args)
  args.join(", ")
end

puts best_comma_separate(1,2,3,4,5,6,7,8)
#=> 1, 2, 3, 4, 5, 6, 7, 8
```

Ruby Classes
Method Arity & Splat Args

- The splat operator can be used to turn an array into a list of arguments

```ruby
def greeting(title, first_name, last_name)
  "Hello #{title} #{first_name} #{last_name}."
end

info = ["Mr.", "Dwight", "Shrute"]

begin
  greeting(info) #=> ArgumentError: wrong number of arguments (1 for 3)
rescue ArgumentError => e
  puts "Oops: #{e.message}"
end

puts greeting(*info) #=> Hello Mr. Dwight Shrute.
```

Ruby Classes
Default Argument Values

- Method arguments can take default values:

```
def do_it(a, b=2, c=3)
  puts a * b * c
end

do_it(1, 2, 3) #=> 6
do_it(1)       #=> 6
```

Data Structures
Arrays, Hashes & Structs

Ruby Arrays
Ruby's Workhorse

- Arrays are ordered collections of objects
- In Ruby, Arrays are just like any other class
- The language does provide some syntactic sugar in terms of convenient constructors and operators (operators are just methods in Ruby)
- Ruby Arrays are type-less, that is, you can put objects of any type into an Array (mix and match as needed)

Ruby Arrays
Ruby's Workhorse

- Arrays are enclosed in brackets ([])
- Let's fire up IRB and try a few of the possible ways to create an Array:

```
:009 > my_array =  [1, 2, 3, 4]
 => [1, 2, 3, 4]
:010 > my_array[0]
 => 1
:011 > my_array[3]
 => 4
:012 > my_array[4]
 => nil
:013 > my_array.first
 => 1
:014 > my_array.last
```

Ruby Arrays
Ruby's Workhorse

- Accessing Array elements can be doing via the array dereference operator ([]) or via several available methods:

```
:001 > Array.new
 => []
:002 > Array.new(2)
 => [nil, nil]
:003 > Array.new(3, 'blah')
 => ["blah", "blah", "blah"]
:004 > Array.new(2, Hash.new)
 => [{}, {}]
:005 > ['a', 'b', 'c', 'd', 1.0]
 => ["a", "b", "c", "d", 1.0]
:006 > Array.[](1, 'a', 2+1)
 => [1, "a", 3]
:007 > Array[1, "a", 3]
 => [1, "a", 3]
```

Ruby Arrays
Ruby's Workhorse

- Arrays can be added to one another, sorted, duplicates can be removed, arrays of arrays can be flatten amongst other standard operations

```
:001 > one = [1, 2, "hey"]
 => [1, 2, "hey"]
:002 > two = [2, 3, 4, "you"]
 => [2, 3, 4, "you"]
:003 > one + two
 => [1, 2, "hey", 2, 3, 4, "you"]
:004 > one - two
 => [1, "hey"]
:005 > two - one
 => [3, 4, "you"]
:006 > (one + two).uniq
 => [1, 2, "hey", 3, 4, "you"]
```

Enumerable
Collections Utilities

- The Enumerable module is included in the Array class
- Enumerable is a Mixing that provides several methods to deal with collection classes, such as the `each` and `each_with_index` method

```ruby
my_array = [1,2,3,4,5]

my_array.each do |val|
  puts "The value is #{val}"
end

my_array.each_with_index do |val, i|
  puts "The value at index #{i} is #{val}"
end
```

Enumerable
Collections Utilities

- The Enumerable module provides many set operation methods such as `all?` and `any?`
- For a full list see http://ruby-doc.org/core-1.9.3/Enumerable.html

```ruby
puts [2,4,6].all? { |n| n % 2 } #=> true

puts ["foo", "bar", "baz"].any? { |str| str == "baz" }
#=> true
```

Enumerable
Enumerable Mutation

- Several methods allow you to mutate the values of an Enumerable
- In the example below we are using the map! method
- In Ruby, traditionally methods ending in a bang (!) mutate the receiver

```ruby
arr = [1,2,3]
arr.map! { |val| val*2 }

p arr #=> [2, 4, 6]
```

Enumerable
Enumerable Module

Follow Along

- The Enumerable module can be mixed in into your own classes to provide them with Array-like capabilities

```
class User
  attr_reader :name
  def initialize(name)
    @name = name
  end
end
```

```
class UserCollection
  include Enumerable

  def initialize(*users)
    @users = users
  end

  def each(&block)
    @users.each(&block)
  end
end
```

Enumerable
Enumerable Module

Follow Along

- Now the `UserCollection` class can be iterated and manipulated just like an Array:

```
jim = User.new("Jim")
phyllis = User.new("Phyllis")
creed = User.new("Creed")
pam = User.new("Pam")
oscar = User.new("Oscar")

user_collection = UserCollection.new(jim, phyllis, creed, pam, oscar)

names = user_collection.map { |user| user.name }
p names #=> ["Jim", "Phyllis", "Creed", "Pam", "Oscar"]

user_with_shortest_name = user_collection.min { |user| user.name.length }
puts user_with_shortest_name.name #=> "Jim"
```

Hashes
Working with Key-Value Pairs

- Hashes are collections of key-value pairs
- The keys can be an arbitrary object type
- In Ruby is common practice to use symbols as keys in a Hash

```ruby
hsh = Hash.new

hsh[:name] = "Erin"
hsh[:employer] = "Dunder Miflin"

p hsh #=> {:name=>"Erin", :employer=>"Dunder Miflin"}

hsh[:some_key] #=> nil
```

Hashes
Working with Key-Value Pairs

- Hashes like Arrays include Enumerable:

```ruby
h = {
  :name => "Michael",
  :email => "mike@example.com",
  :age => 44
}

h.each do |key, value|
  puts "The key '#{key}' has the value '#{value}'."
end

p h.keys #=> [:name, :email, :age]
p h.values #=> ["Michael", "mike@example.com", 44]
p h.to_a #=> [[:name, "Michael"], [:email, "mike@example.com"], [:age, 44]]
```

Hashes
Working with Key-Value Pairs

- Keys can be of mixed types and elements can be accessed using the dereference operator ([]) or the `fetch` method

```ruby
h = {
  "foo" => 1,
  "bar" => 10,
  :baz  => 100
}

p h["baz"] #=> nil
p h[:baz] #=> 100

puts h.fetch(:quux) { "Default value" }
```

Hashes
Working with Key-Value Pairs

- You can nest Hashes and access their values using chained []

```
params = {
  :user => {
    :name => "Mike Wallace",
    :email => "wallace@example.com"
  }
}

p params[:user] #=> {:name=>"Mike Wallace", :email=>"wallace@example.com"}
p params[:user][:name] #=> "Mike Wallace"
```

Hashes
Working with Key-Value Pairs

- Hash keys can be any object, regardless of its complexity:

```
class User < Struct.new(:name)
end

toby = User.new("Toby")

hsh = {
  toby => 137.9
}

p hsh[toby] #=> 137.9
```

Structs
Working with Key-Value Pairs

- A Struct is a lightweight collection of attributes providing accessor methods, without having to write an explicit class:

```
class Address < Struct.new(:street, :city, :state, :zip)
end

address = Address.new("123 Mulberry Ln", "Scranton", "PA", "18504")

p address
#=> #<struct Address street="123 Mulberry Ln", city="Scranton", state="PA", zip="18504">

p address.members #=> [:street, :city, :state, :zip]
```

Structs
Working with Key-Value Pairs

- Structs like Hashes and Arrays include Enumerable:

```
address.each do |value|
  puts value
end
#=> 123 Mulberry Ln
#=> Scranton
#=> PA
#=> 18504
```

Lambdas, Procs & Blocks

Portable Code

Lambda
Portable Code

- In Ruby the line between code and data is blurred. The lambda method allows us assign a code of block to a variable also knows as a Proc:

```ruby
squared = lambda { |n| n ** 2 }

puts squared.call(4) #=> 16

puts squared.call(3) #=> 9
```

Lambda & Proc.new

Portable Code

- More lambda examples:

```ruby
hello = lambda { puts "Hi" }
hello.call #=> "Hi"

gimme = lambda { |item| "Please pass me the #{item}." }
puts gimme.call("green beans") #=> "Please pass me the green beans."

concatenator = lambda { |a, b, c| a+b+c }
puts concatenator.call("lions", "tigers", "bears") #=> "lionstigersbears"

exclamator = lambda do |phrase|
  phrase + "!!!"
end
puts exclamator.call("I <3 Ruby") #=> "I <3 Ruby!!!"
```

Lambda
Portable Code

- A Proc can take multiple parameters:

```
shuffle = lambda do |a, b|
  a.zip(b).flatten
end

p shuffle.call([1, 2], ["a", "b"]) #=> [1, "a", 2, "b"]
```

Lambda
Portable Code

Follow Along

```ruby
def takes_a_block(&the_block)
  the_block.call
end

takes_a_block { puts "Hello" } #=> Hello

def passes_a_proc( the_proc )
  takes_a_block( &the_proc )
end

my_proc = lambda { puts "Howdy!" }

passes_a_proc( my_proc ) #=> Howdy
```

- Once you have a proc you can pass it around to a method
- The & in a method, declares the parameter as a proc passed as a block
- To further pass a proc parameter we need to “dereference” it with &

Lambda & Proc.new

Portable Code

- A Proc can take multiple parameters:

```ruby
incrementer = Proc.new { |num| num + 1 }
puts incrementer.call(9) #=> 10

decrementer = Proc.new do |val|
  val - 1
end
puts decrementer.call(50) #=> 49

# Don't use `proc` anymore
halve = proc { |num| num / 2 }
puts halve.call(900) #=> 450
```

Lambda & Proc.new

Portable Code

- Lambda is strict about argument counts:

```
dashit = lambda { |a, b, c| "#{a}-#{b}-#{c}" }

puts dashit.call("foo", "bar", "baz") #=> "foo-bar-baz"

dashit.call("just one arg") #=> ArgumentError: wrong number of arguments (1 for 3)
dashit.call(1,2,3,4) #=> ArgumentError: wrong number of arguments (4 for 3)
```

Lambda & Proc.new

Portable Code

- Proc.new is lenient about argument counts:

```
underscorer = Proc.new { |a,b,c| [a,b,c].join('_') }

puts underscorer.call("foo", "bar", "baz") #=> "foo_bar_baz"

# Parameters without arguments are given as `nil`
puts underscorer.call("just one") #=> "just one__"

# Extra arguments are ignored
puts underscorer.call("l","m","n","o","p") #=> l_m_n
```

Lambda & Proc.new

Portable Code

- A lambda returns from only it's own scope:

```
def some_method
  puts "start"
  l = lambda { return }
  l.call
  puts "finish"
end

some_method

#=> start
#=> finish
```

Lambda & Proc.new

Portable Code

- Proc.new returns from the enclosing scope:

```
def another_method
  puts "start"
  p = Proc.new { return }
  p.call
  puts "finish"
end

another_method

#=> "start"
# Never get to "finish" though!
```

Block Arguments
Portable Code

- Any method can take a block as an argument
- “Implicit block” argument

```ruby
def do_what_you_want
  yield #=> "I'm here!"
end

do_what_you_want do
  puts "I'm here!"
end
```

Block Arguments
Portable Code

- Explicit block arguments are declared with the & operator

```
def block_caller(&blk)
  blk.call #=> "Some custom behavior"
end

block_caller do
  puts "Some custom behavior"
end
```

Blocks
Portable Code

- We can yield control to the passed block
- ...and yield an object to it:

```
def string_me
  yield "Here is my string"
end

string_me do |str|
  puts str
end
#=> Here is my string
```

Lambda & Proc.new

Portable Code

- Yield self to a given block:

```ruby
class Employee
  attr_accessor :name, :college

  def initialize(name, college)
    @name = name
    @college = college
  end

  def gimme_self
    yield self
  end
end
```

```ruby
andy = Employee.new("Andy", "Cornell")

andy.gimme_self do |the_self|
  the_self.college += " University"
end

puts andy.name #=> "Andy"
puts andy.college #=> "Cornell University"
```

Lambda & Proc.new

Portable Code

- Check if a block was passed:

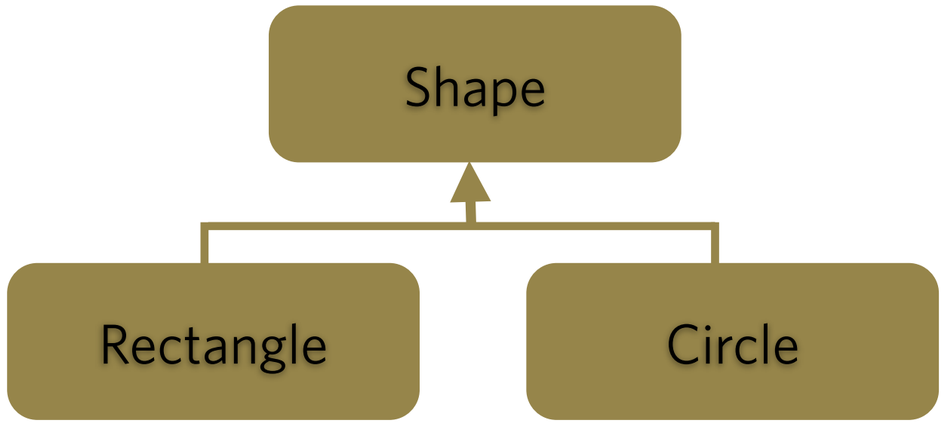
```ruby
def not_sure
  if block_given?
    yield "My name is"
  else
    puts "No block given"
  end
end

not_sure do |val|
  puts val + " Angela" #=> "My name is Angela"
end

not_sure #=> "No block given"
```

Inheritance

Object-Orientation in Ruby

Inheritance
Extending a Class

- In this exercise we will build a simple example to explore and revisit basic OO concepts in Ruby
- We'll do a Ruby version of the traditional "Shapes" example



Inheritance
Extending a Class

- Let's start by creating a base class in shape.rb containing the Shape class:

```
class Shape
  attr_accessor :x, :y, :color

  def initialize(init_x, init_y, color)
    @x, @y, @color = init_x, init_y, color
  end

  def move_to(new_x, new_y)
    @x, @y = new_x, new_y
  end

  def move_by(new_x, new_y)
    move_to(new_x + @x, new_y + @y)
  end

  def draw
    raise "generic shape can't draw itself"
  end

  def area
    raise "generic shape doesn't have a defined area"
  end
end
```

Inheritance
Extending a Class

- Let's test the Shape class:

```
my_shape = Shape.new(42, 33, 'Ruby Red')
p my_shape #=> #<Shape:0x007fa1f090a7b0 @x=42, @y=33, @color="Ruby Red">
p my_shape.x #=> 42
p my_shape.y #=> 33
p my_shape.draw #=> shape.rb:17:in `draw': generic shape can't draw itself (RuntimeError)
```

Inheritance
Extending a Class

- With the Shape class in place let's create the Circle subclass

```ruby
require_relative 'shape'

class Circle < Shape
  attr_accessor :radius

  def initialize(x, y, radius, color)
    super(x, y, color)
    @radius = radius
  end

  def draw
    print("Drawing a Circle at:(#@x, #@y), radius #@radius and color #@color\n")
  end

  def area
    Integer(Math::PI * radius ** 2)
  end
end
```

Inheritance
Extending a Class

- Let's test the Circle class:

```
my_circle = Circle.new(7, 11, 13, 'PC Beige')
p my_circle #=> #<Circle:0x007fa74a084640 @x=7, @y=11, @color="PC Beige", @radius=13>
p my_circle.x #=> 7
p my_circle.y #=> 11
p my_circle.radius #=> 13
my_circle.draw #=> Drawing a Circle at:(7, 11), radius 13 and color PC Beige
p my_circle.area #=> 530
```

Lab 1.0
Inheritance & Polymorphism

- In Lab 1.0 students will:
  - Create a simple Ruby program containing a Rectangle class that inherits from Shape
  - Write a simple test program to test polymorphism in Ruby, e.g. write a method that takes an object and asks for it to draw itself
  - Create an Array of Shapes, iterate over the shapes and invoke various “shape” methods

Lab 1.1
Inheritance & Polymorphism

- In Lab 1.1 students will:
  - Create a Color class that contains RGB integer values and use it in the draw method
  - Discover Duck Typing: Create the Outlaw class that provides a draw and area method but is not part of the Shape hierarchy. Add it to the test for Polymorphism created in Lab 1.0

RubyGems

Tapping into the Ruby Open Source Ecosystem

RubyGems
Sharing Packaged Ruby Code

- Ruby comes with a pretty complete standard library but when the standard library is not enough you can count on Ruby's Open Source Ecosystem to provide the functionality you are looking for in the form of a Ruby Gem
- A gem is a packaged Ruby application/library under a given name (e.g. rails) and a version (e.g. 3.2.11)
- The Ruby program RubyGems is the ruby packaging system, with it you can install, remove, require and query gem packages

RubyGems
Using the Gem Command

- RubyGems is similar to the package management tools RPM and apt-get
- RubyGems provides the gem command
- To see the version of RubyGems installed in your system use:
  - gem -v

```
/> gem -v
1.8.24
```

RubyGems
Using the Gem Command

- To see the list of gems installed in your system use:
  - gem list

```
/> gem list

*** LOCAL GEMS ***

bundler (1.2.3)
nokogiri (1.5.6)
rake (10.0.3)
rubygems-bundler (1.1.0)
rvm (1.11.3.5)
```

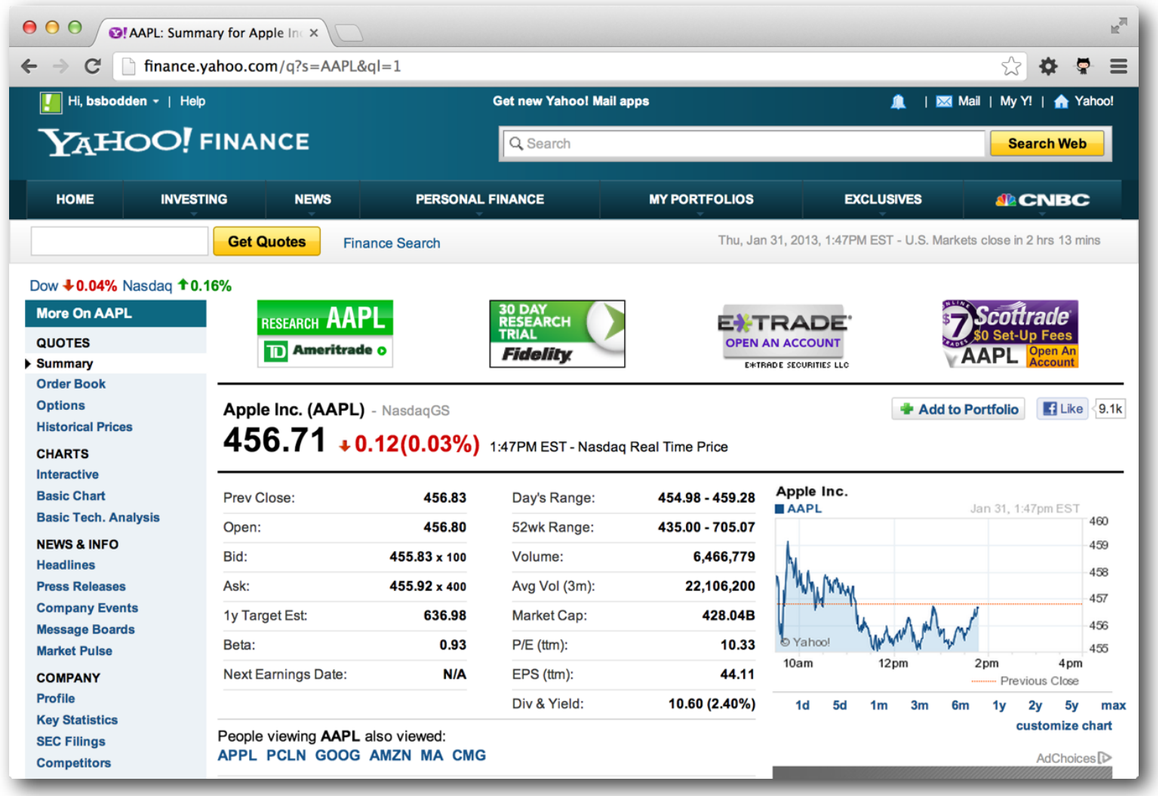
RubyGems
Installing a Gem

- In the upcoming labs we will use the popular Nokogiri gem (http://nokogiri.org/)
- Nokogiri is a simple XML/HTML parser for Ruby
- To install a gem we can use them gem install command (or Bundler as we'll do in this sections' Lab)
- gem install nokogiri

```
/> gem install nokogiri
Building native extensions.  This could take a while...
Successfully installed nokogiri-1.5.6
1 gem installed
Installing ri documentation for nokogiri-1.5.6...
Installing RDoc documentation for nokogiri-1.5.6...
```

Lab 1.2
Screen Scrapping with Nokogiri

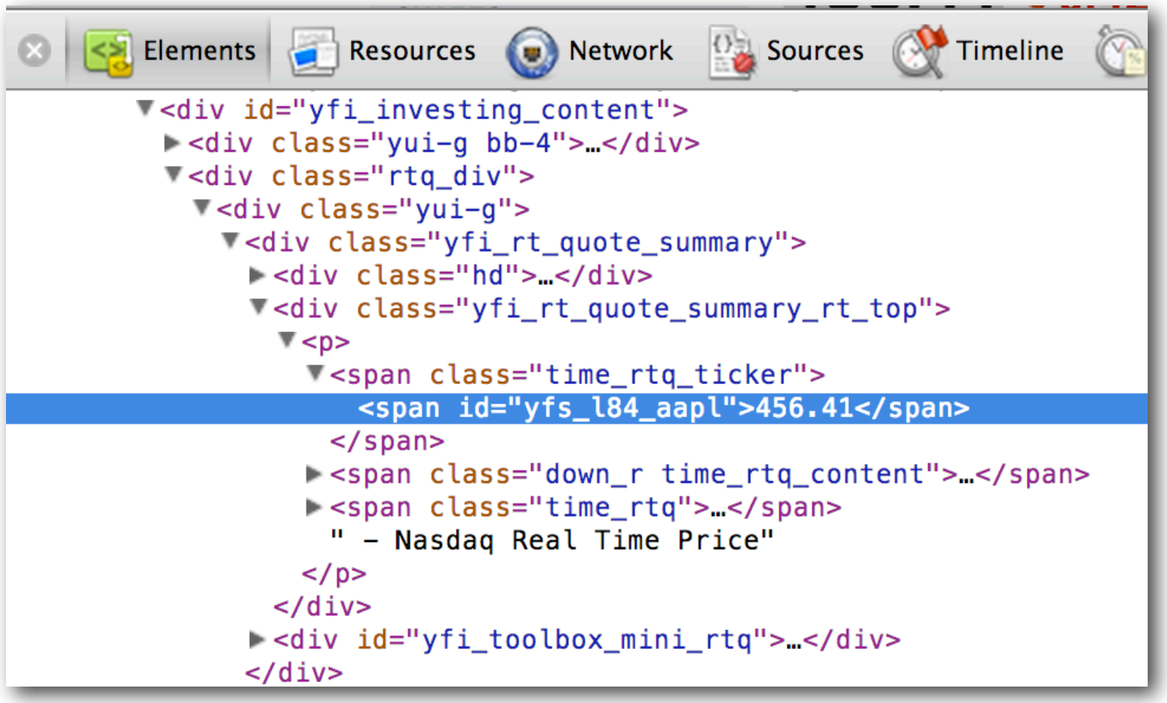
- In Lab 1.2 students will create a simple Ruby program using Nokogiri to parse stock prices from Yahoo! Financial



The web page above shows the stock page for Apple (AAPL) for which the URL is: **http://finance.yahoo.com/q?s=AAPL**

Lab 1.2
Screen Scrapping with Nokogiri

- Nokogiri is a tool well-suited for HTML scrapping
- Examination of the HTML source for the Yahoo! Financial page reveals the structure of the HTML elements we are interested in extracting



Lab 1.2
Screen Scrapping with Nokogiri

- The goal of the lab is to retrieve the stock value (highlighted below)
- The key pattern to note is that the stock price is in a element that follows a span with the css class “time_rtq_ticker”:

```
<div class="yfi_rt_quote_summary_rt_top">
▼<p>
  ▼<span class="time_rtq_ticker">
     <span id="yfs_l84_aapl">456.41</span>
   </span>
  ▶<span class="down_r time_rtq_content">…<
```

Lab 1.2
Screen Scrapping with Nokogiri

- The expected output of the **nokogiri_quotes.rb** program is shown below:

```
/> ruby nokogiri_quotes.rb
"Looking up http://finance.yahoo.com/q?s=AAPL..."
AAPL =>  456.70
```

- The output should show the stock symbol (AAPL hardcoded for now) followed by a “=>” and the stock price

Lab 1.2
Screen Scrapping with Nokogiri

- Suggested Lab Steps:
  - Create a new directory for your application
  - Add a .rvmrc file with: rvm use 1.9.3@nokogiri_quotes --create
  - Add a Gemfile requiring nokogiri and run the bundle command
  - Create a new Ruby program called nokogiri_quotes.rb
  - Require rubygems, nokogiri and open-uri

Lab 1.2
Screen Scrapping with Nokogiri

- Suggested Lab Steps (cont.):
  - Load the HTML using open-uri:
    - content = open(uri)
  - Create a Nokogiri document
    - doc = Nokogiri::HTML(content)
  - Search for the quote value using XPath or CSS
    - Find the element with the class time_rtq_ticker"
    - Grab the inside and print its inner_text

Mixins
Reusable Behaviors

Mixins
Reusable Behaviors

- Rubyist prefer composition over inheritance. Modules can be used to group behaviors that can be mixed in a class using the **include** method

```ruby
module Horn
  def honk
    puts "!!!!!!!!!!!!!!!!!!!"
  end
end

class Vehicle
  include Horn
end

v = Vehicle.new
v.honk #=> "!!!!!!!!!!!!!!!!!!!"
```

Mixins
Reusable Behaviors

- Mixin behavior can be shared:

```
class Vehicle
  include Horn
end

class Boat
  include Horn
end

v = Vehicle.new
v.honk #=> "!!!!!!!!!!!!!!!!!!!"

b = Boat.new
b.honk #=> "!!!!!!!!!!!!!!!!!!!"
```

Mixins
Reusable Behaviors

- If we use extend a module methods become class methods instead:

```
module Scary
  def boo
    puts "BOOOOOOOO!!!!!!!"
  end
end

class MovieStar
  extend Scary
end

MovieStar.boo #=> "BOOOOOOOO!!!!!!!"
```

Mixins
Reusable Behaviors

- Modules provide hook methods that respond to certain events
- For example, **included** is invoked when our module is included

```
module Introductions
  def self.included(klass)
    puts "We're getting included, yay!"
    klass.extend ClassMethods
  end

  module ClassMethods
    def intro
      puts "I am the #{self} class"
    end
  end

  def intro
    puts "I am an instance of the #{self.class} class"
  end
end
```

Mixins
Reusable Behaviors

- If we create a new class and include the Introductions module we should see the included hook being invoked and the methods added to the Metal class at the designated scopes:

```
class Metal
  include Introductions #=> "We're getting included, yay!"
end

Metal.intro #=> "I am the Metal class"

m = Metal.new
m.intro #=> "I am an instance of the Metal class"
```

TDD w/ RSpec

Test Driven & Behavior Driven Development with RSpec

BDD
Behavior Driven Development

- BDD focuses TDD to deliver the maximum value possible to stakeholders
- BDD is a refinement in the language and tooling used for TDD
- As the name implies with BDD we focus on behavior specifications
- Typically BDD works from the outside in, that is starting with the parts of the software whose behavior is directly perceive by the user
- We say BDD refines TDD in that there is an implicit decoupling of the tests and the implementation (i.e.. don't tests implementation specifics, test perceived behavior)

BDD
Behavior Driven Development

- BDD focuses on “specifications” that describe the behavior of the system
- In the process of fleshing out a story the specifications start from the outside and might move towards the inside based on need
- In the context of a Web Application this Outside-In approach typically means that we are starting with specifications related to the User Interface
- If we are talking about a software component then we mean the API for said component

BDD
Behavior Driven Development

- BDD helps us figure out what to test, where to start and what to ignore (or what to make a target of opportunity)
  - What to test? ➜ Use Cases or User Stories, test what something **does (behavior)** rather than what something **is (structure)**
  - Where to start? ➜ From the outer most layer
  - What to ignore? ➜ Anything else... Until proven that you can't

BDD
Behavior Driven Development

- BDD focuses on **getting the words right**, the resulting specifications become a runnable/self-verifying form of documentation

- BDD specifications follow a format that makes them easy to be driven by your system's User Stories

*As a [**role**], I want [**goal**] so that [**benefit**]*

User Story

*Given [**role and its state**]*
*When [**an event/action occurs**]*
*Then [**the benefit**]*

Specification

RSpec
Ruby's BDD Framework

- RSpec is the most popular BDD framework for Ruby
- Created by Steven Baker in 2005, enhanced and maintained by David Chelimsky until late 2012
- RSpec provides a DSL to write executable examples of the expected behavior of a piece of code in a controlled context

RSpec
Ruby's BDD Framework

- RSpec uses the method **describe** to create and Example Group
- Example groups can be nested using the **describe** or **context** methods

```ruby
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

RSpec
Ruby's BDD Framework

- RSpec uses the method **describe** to create and Example Group
- Example groups can be nested using the **describe** or **context** methods

```ruby
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

Example Group

RSpec
Ruby's BDD Framework

- RSpec uses the method **describe** to create and Example Group
- Example groups can be nested using the **describe** or **context** methods

```ruby
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

RSpec
Ruby's BDD Framework

- RSpec uses the method **describe** to create and Example Group
- Example groups can be nested using the **describe** or **context** methods

```ruby
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

Example

RSpec
Ruby's BDD Framework

- RSpec uses the method **describe** to create and Example Group
- Example groups can be nested using the **describe** or **context** methods

```ruby
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

RSpec
Ruby's BDD Framework

- RSpec uses the method **describe** to create and Example Group
- Example groups can be nested using the **describe** or **context** methods

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)          Expectation
  end
end
```

RSpec
Ruby's BDD Framework

- RSpec uses the method **describe** to create and Example Group
- Example groups can be nested using the **describe** or **context** methods

```ruby
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

RSpec
Ruby's BDD Framework

- RSpec uses the method **describe** to create and Example Group
- Example groups can be nested using the **describe** or **context** methods

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

Matcher

RSpec
Ruby's BDD Framework

- RSpec uses the method **describe** to create and Example Group
- Example groups can be nested using the **describe** or **context** methods

```ruby
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

Test-Driven Development
Drive your Development with Tests

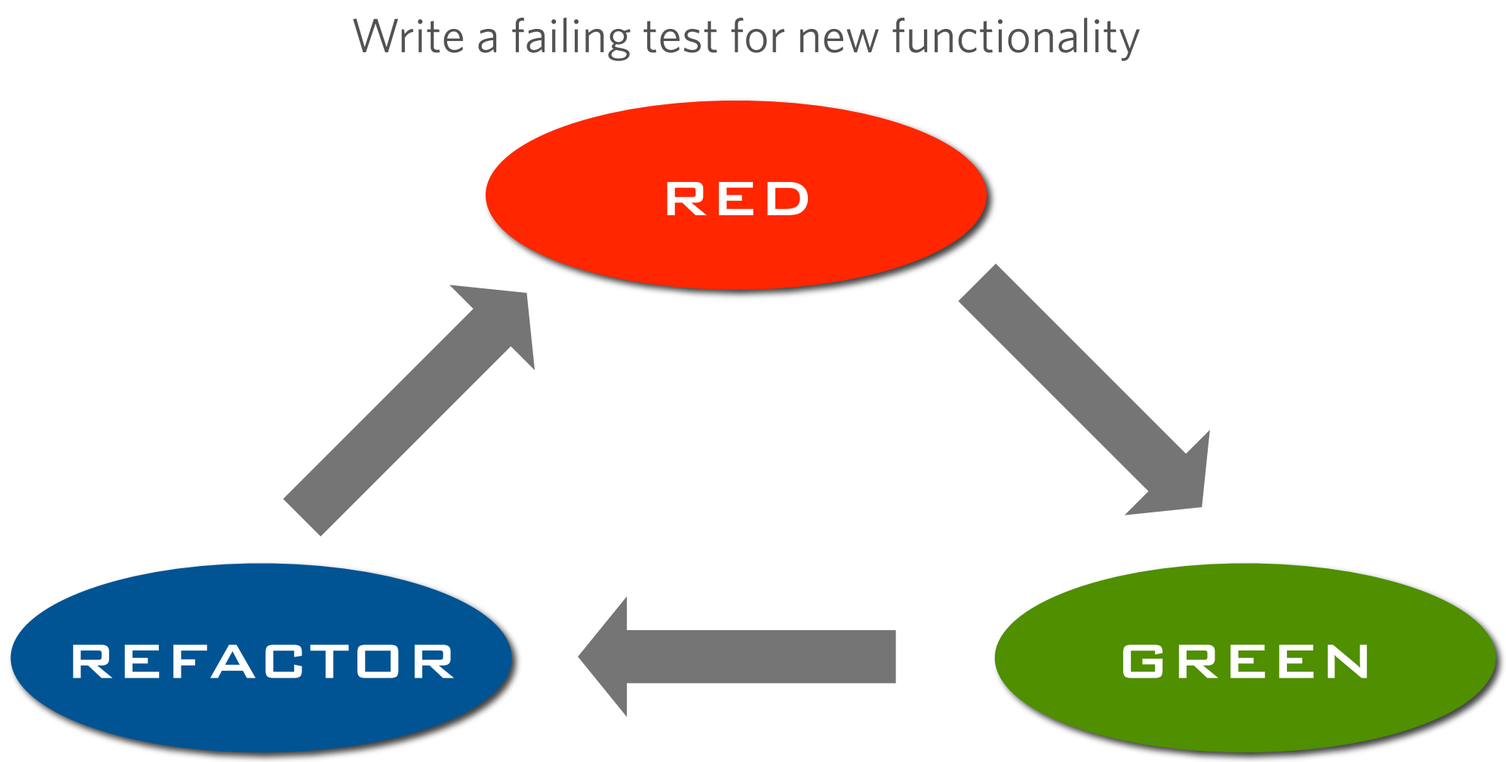
- TDD is not *really* about testing
- TDD is a design technique
- TDD leads to cleaner code with separation of concerns
- Cleaner code is more reliable and easier to maintain

Test-Driven Development
Drive your Development with Tests

- TDD creates a tight loop of development that cognitively engages us
- TDD gives us lightweight rigor by making development, goal-oriented with a clear goal setting, goal reaching and improvement stages
- The stages of TDD are commonly known as the Red-Green-Refactor loop

Test-Driven Development
Drive your Development with Tests

- The Red-Green-Refactor Loop:



Test-Driven Development
Evolve your Code with Tests

Follow Along

- Let's work through a simple TDD/BDD exercise using RSpec
- We'll design a simple shopping cart class
- We'll start by creating a new folder for our exercise and adding a .rvmrc file and a Gemfile

```
/>mkdir rspec-follow-along
/>cd rspec-follow-along
/>echo 'rvm use 1.9.3@rspec-follow-along' > .rvmrc
/>touch Gemfile
/>mkdir spec
/>mkdir lib
```

```
source :rubygems

group :test do
  gem 'rspec'
end
```

Test-Driven Development
Evolve your Code with Tests

- With our project configure for RVM and with a Gemfile in place we can reenter the directory to activate the Gemset and run the bundle command:

```
/>cd ..
/>cd rspec-follow-along/
Using /Users/user/.rvm/gems/ruby-1.9.3-p374 with gemset rspec-follow-along
bsb in ~/Courses/code/rspec-follow-along using ruby-1.9.3-p374@rspec-follow-along

/> bundle
Using diff-lcs (1.1.3)
Using rspec-core (2.12.2)
Using rspec-expectations (2.12.1)
Using rspec-mocks (2.12.2)
Using rspec (2.12.0)
Using bundler (1.2.3)
Your bundle is complete! Use `bundle show [gemname]` to see where a bundled gem is installed.
```

Test-Driven Development
Evolve your Code with Tests

Follow Along

- We'll start the RGR loop with the simplest possible failure: *There is no Cart!*
- Create the file cart_spec.rb in the spec directory with the following contents:

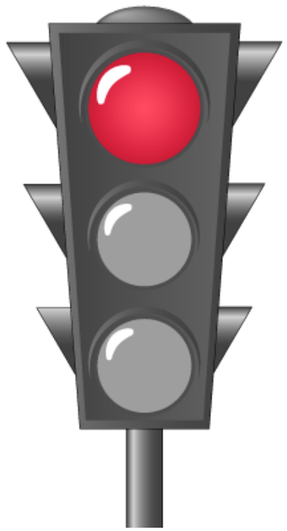
```
describe Cart do

end
```

Test-Driven Development
Evolve your Code with Tests

Follow Along

- Let's run the specs using the rspec command and passing the spec directory as an argument

- Have we arrived at the RED state in our red-green-refactor cycle?



```
/>rspec spec
/Users/bsb/Courses/code/rspec-follow-along/spec/cart_spec.rb:1:in `<top (required)>':
uninitialized constant Cart (NameError)
```

Hint: if you have a failure with no tests it typically means that you need a test (but let's ignore that for a second...)

Test-Driven Development
Evolve your Code with Tests

Follow Along

- Let's create the Cart class in a /lib folder and require it in our spec:

```
class Cart
end
```

```
require_relative '../lib/cart.rb'

describe Cart do
end
```

- Now we have no failures but also we have no specs...

```
/> rspec spec
No examples found.


Finished in 0.00006 seconds
0 examples, 0 failures
```

Test-Driven Development
Evolve your Code with Tests

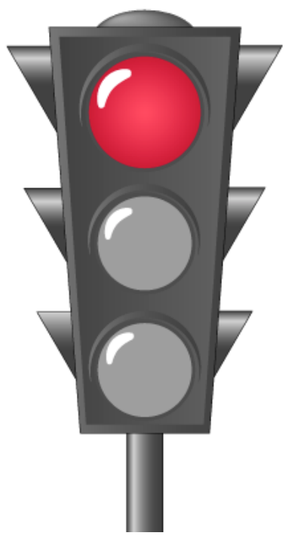
- Let's craft our first real test to drive the development of the Cart
- The spec to tackle is: *"An instance of Cart when new contains no items"*

```
require_relative '../lib/cart.rb'

describe Cart do
  context "a new cart" do
    it "contains no items" do
      @cart.should be_empty
    end
  end
end
```

Test-Driven Development
Evolve your Code with Tests

Follow Along

- If we run the specs we can see a failure:



```
/> rspec spec
F

Failures:
  1) Cart a new cart contains no items
     Failure/Error: @cart.should be_empty
     NoMethodError:
       undefined method `empty?' for nil:NilClass
     # ./spec/cart_spec.rb:6:in `block (3 levels) in <top (required)>

Finished in 0.00243 seconds
1 example, 1 failure
```

Now we have our first "real" test-driven failure
(and that is a good thing!)

Test-Driven Development
Evolve your Code with Tests

- One of the mantras of BDD is to “get the words right”
- If you noticed on the last run the spec output read as *“Cart a new cart contains no items”*
- RSpec is flexible enough to allow us to pass a string to be prefixed to the describe block to make tailor the output to our needs

```
require_relative '../lib/cart.rb'

describe "An instance of", Cart do
  context "a new cart" do
    it "contains no items" do
      @cart.should be_empty
    end
  end
end
```

Test-Driven Development
Evolve your Code with Tests

- If we run the specs we can see that the output now matches the desire spec wording

```
/> rspec spec
F

Failures:

  1) An instance of Cart when new contains no items
     Failure/Error: @cart.should be_empty
     NoMethodError:
       undefined method `empty?' for nil:NilClass
     # ./spec/cart_spec.rb:6:in `block (3 levels) in <top (required)>'

Finished in 0.00154 seconds
1 example, 1 failure

Failed examples:

rspec ./spec/cart_spec.rb:5 # An instance of Cart when new contains no items
```

Test-Driven Development
Evolve your Code with Tests

- The output shows that we have two failures, one implicit and one explicit
- Explicit Failure: We are assuming that a cart has an empty? method
- Implicit Failure: The instance variable @cart has not been initialized

```
Failures:

  1) An instance of Cart when new contains no items
     Failure/Error: @cart.should be_empty
     NoMethodError:
       undefined method `empty?' for nil:NilClass
     # ./spec/cart_spec.rb:6:in `block (3 levels) in <top (required)>'
```

Test-Driven Development
Evolve your Code with Tests

Follow Along

- The output shows that we have two failures, one implicit and one explicit
- Explicit Failure: We are assuming that a cart has an empty? method
- Implicit Failure: The instance variable @cart has not been initialized

```
Failures:

  1) An instance of Cart when new contains no items
     Failure/Error: @cart.should be_empty
     NoMethodError:
       undefined method `empty?' for nil:NilClass
     # ./spec/cart_spec.rb:6:in `block (3 levels) in <top (required)>'
```

Test-Driven Development
Evolve your Code with Tests

- We'll start by addressing the fact that our test fixture hasn't been setup
- Just adding the line `@cart = Cart.new` wouldn't be very TDDish
- What we should do is first make the failure explicit by writing a test for it!

```
require_relative '../lib/cart.rb'

describe "An instance of", Cart do

  it "should be properly initialized" do
    @cart.should be_a(Cart)
  end

...
```

Remember our initial failure with no tests?

Test-Driven Development
Evolve your Code with Tests

- Now we have two valid failing tests to pass, let's get on with it!

```
...

  1) An instance of Cart should be properly initialized
     Failure/Error: @cart.should be_a(Cart)
       expected nil to be a kind of Cart
     # ./spec/cart_spec.rb:6:in `block (2 levels) in <top (required)>'

  2) An instance of Cart when new contains no items
     Failure/Error: @cart.should be_empty
     NoMethodError:
       undefined method `empty?' for nil:NilClass
     # ./spec/cart_spec.rb:11:in `block (3 levels) in <top (required)>'

Finished in 0.00233 seconds
2 examples, 2 failures
```

Test-Driven Development
Evolve your Code with Tests

- We'll pass the test by adding the line `@cart = Cart.new` in a before-each block:

```
describe "An instance of", Cart do

  before :each do
    @cart = Cart.new
  end
```

```
...
  1) An instance of Cart when new contains no items
     Failure/Error: @cart.should be_empty
     NoMethodError:
       undefined method `empty?' for #<Cart:0x007fc98316cb60>

...
```

Test-Driven Development
Evolve your Code with Tests

- Let's add a skeleton empty? method to the Cart class:

```
class Cart
  def empty?
    nil
  end
end
```

```
...
  1) An instance of Cart when new contains no items
     Failure/Error: @cart.should be_empty
       expected empty? to return true, got nil
     # ./spec/cart_spec.rb:15:in `block (3 levels) in <top (required)>'

...
```

Test-Driven Development
Evolve your Code with Tests

- Now we can comply with the spec by providing an implementation of the internal of our Cart
- In this case we are using a Hash to hold our items and delegating to the @items#empty? method

```ruby
class Cart

  def initialize
    @items = {}
  end

  def empty?
    @items.empty?
  end
end
```

```
/> rspec spec
..

Finished in 0.00196 seconds
2 examples, 0 failures
```

We've reached the GREEN state

Test-Driven Development
Drive your Development with Tests

- In the REFACTOR state we concentrate on making the current implementation better, cleaner and more robust

- It is very likely that early on in the development there won't be much to refactor

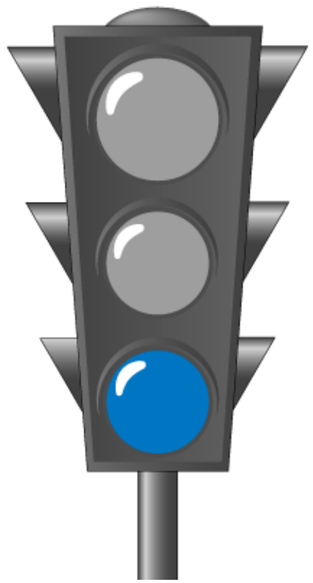
- The need for refactoring is a side-effect of increasing complexity and interaction between classes and subsystems

- Refactoring can also introduce implementation specific specs or reveal holes in your previous specs

Test-Driven Development
Evolve your Code with Tests

Follow Along

- Let's use Ruby's Forwardable module to simplify the delegation of the collection methods to the @items Hash:



```ruby
class Cart
  extend Forwardable
  def_delegator :@items, :empty?
  def initialize
    @items = {}
  end
end
```

```
/>rspec spec
..

Finished in 0.00365 seconds
2 examples, 0 failures
```

Lab 1.3
TDD & BDD

- Lab 1.3 consists of 4 specs to be implemented in a TDD fashion:
  - An new and empty cart total value should be $0.0
  - An empty cart should no longer be empty after adding an item
  - An cart with items should have a total value equal to the sum of each items' value times its quantity
  - Increasing the quantity of an item should not increase the Carts' unique items count

Metaprogramming

Code As Data

Metaprogramming
Code As Data

- Meta-Programming...
  - ... is about programs that write programs
  - ... it's a superb tool for building frameworks
  - ... it's the key ingredient for building domain-specific languages

Metaprogramming
Code As Data

- Ruby is a great vehicle for meta-programming because:
  - ... it is dynamic and reflexible
  - ... it is open and malleable
  - ... code is data and data is code
  - ... has a clean syntax that is low on ceremony

Metaprogramming
Code As Data

- Ruby on Rails
  - ... uses meta-programming to bring the language closer to the problem
  - ... could be defined as a tightly integrated collection of domain-specific languages (DSL) for building Web Applications

Metaprogramming
eval

- Ruby provides the eval family of methods for runtime execution of code
- For example the eval method can evaluate any string containing valid Ruby
- Let's fire up IRB and play around with the eval method:

```
:001 > eval "2*2"
 => 4
:002 > eval "'foo'.upcase"
 => "FOO"
```

Metaprogramming
instance_eval

- The method **instance_eval** can evaluate a string or a block:

```
:001 > a = [1, 2, 3]
 => [1, 2, 3]
:002 > a.instance_eval { inject(0) { |s, v| s+v }}
 => 6
```

Metaprogramming
instance_eval

- The method **instance_eval** can evaluate a string or a block in the context of the receiver:

```
:001 > a = [1, 2, 3]
 => [1, 2, 3]
:002 > a.instance_eval { inject(0) { |s, v| s+v }}
 => 6
```

Metaprogramming
instance_eval

Follow Along

- The module below defines a class method to create class attribute accessor methods:

```
module ClassAccessor
  def cattr_accessor(name)
    # Add method to the class
    instance_eval <<-EOS
      def #{name}=(val)
        @#{name} = val
      end
      def #{name}
        @#{name}
      end
    EOS
  end
end
```

```
class APIWrapper
  extend ClassAccessor
  cattr_accessor :timeout
end

APIWrapper.timeout = 12
puts APIWrapper.timeout #=> 12
```

Metaprogramming
class_eval

- The method **class_eval** can evaluate a string or a code block in the context of the or module that it is invoked on:

```
module ImmutableAttributes
  class Error < StandardError
  end

  def immutable(name)
    class_eval <<-EOS
      def #{name}=(val)
        if @#{name}
          raise ImmutableAttributes::Error.new("#{name} is immutable")
        else
          @#{name} = val
        end
      end
    EOS
  end
end
```

Metaprogramming
class_eval

- Let's test the ImmutableAttributes module on the class User:

```
class User
  extend ImmutableAttributes
  attr_accessor :name
  immutable :name # Order matters here
end

u = User.new

# Set the name once
u.name = "Stanley"
puts u.name #=> "Stanley"

# Immutable after that
u.name = "Kevin" #=> ImmutableAttributes::Error: name is immutable
```

Metaprogramming
Dynamic Method Definition

Follow Along

- The define_method method allows us to dynamically add methods to a class:

```
class Aircraft
  define_method(:wingspan) do
    @wingspan
  end

  define_method(:wingspan=) do |wingspan|
    @wingspan = wingspan
  end
end

a = Aircraft.new
a.wingspan = 230
puts a.wingspan
```

Metaprogramming
Dynamic Method Definition

Follow Along

- For example, we can have method that creates methods:

```
class Lightbulb
  def self.flash(interval)
    define_method("flash_#{interval}") do
      while true
        puts "FLASHING BRIGHTLY <(((((((("
        sleep interval
      end
    end
  end
end

Lightbulb.flash(2)

bulb = Lightbulb.new
bulb.flash_2
```

Lab 1.4
Mimicking Rails

- Lab 1.4 we will mimic a technique used in Rails ActiveRecord to create class methods on subclasses using the singleton class, class_eval and a mixin:

```
class Project < ActiveRecord::Base
  belongs_to :owner
```

- For example, inheriting from the ActiveRecord::Base class adds the belongs_to method which takes a symbol as a parameter and adds a great deal of utility methods to both the class and its instances

Lab 1.4
Mimicking Rails

- In our mock version of belongs_to we want the extending class to behave as shown below:

```ruby
class Project < MyActiveRecord::Base
  belongs_to :owner
  belongs_to :company

  attr_accessor :name
end

project = Project.new
project.name = "Fire Gareth"
project.owner = "David Brent"
project.company = "Wernham Hogg"
p "The project is #{project.name}, owned by #{project.owner} at #{project.company}"
```

```
"The project is Fire Gareth, owned by David Brent at Wernham Hogg"
```

Lab 1.4
Mimicking Rails

- Use the skeleton implementation of the MyActiveRecord module, containing the Base class and the belongs_to method:

```
module MyActiveRecord
  class Base
    def self.belongs_to(clazz)
      # Your code goes here!
    end
  end
end
```

Metaprogramming
method_missing and friends

- Ruby provides several hook methods that can be used to create custom behaviors:
  - method_missing
  - method_added
  - method_removed

Metaprogramming
method_missing and friends

- Ruby's Markup Builder (http://builder.rubyforge.org/) is a example of what can be achieved with method missing:

```ruby
require 'builder'

page = Builder::XmlMarkup.new

page.html {
  page.head {
    page.title "Resume"
  }
  page.body {
    page.h1 "Matz Resume"
    page.comment! "Yukihiro Matsumoto's Resume"
    page.ul {
      page.li "Created Ruby!"
      page.li "You want more?"
    }
  }
}

p page
```

```
/> ruby builder_example.rb
<html>
  <head>
    <title>Resume</title>
  </head>
  <body>
    <h1>Matz Resume</h1>
    <!-- Yukihiro Matsumoto's Resume -->
    <ul>
      <li>Created Ruby!</li>
      <li>You want more?</li>
    </ul>
  </body>
</html>
```

Metaprogramming
method_missing and friends

- The method method_missing is a hook that can inform us when an invoked method is not found in the receiver:

```
class Media
  def method_missing(name)
    puts "I don't respond to the method `#{name}`"
  end
end

m = Media.new
m.do_something #=> "I don't respond to the method `do_something`"
```

Metaprogramming
method_missing and friends

- Let's use method_missing for delegation in order to police the methods that can be invoked on a class
- Let's say that we wanted to control access to the methods in the Media class...

```
class Media
  def play
    puts "Playing..."
    @playing = true
  end

  def stop
    puts "Stopping..."
    @playing = false
  end

  def destroy
    puts "Destroying myself!!!!!"
  end
end
```

Metaprogramming
method_missing and friends

- We could filter the calls to a media object via a delegator that uses **method_missing**:

```
class Delegator
  def initialize(target, *approved_methods)
    @target = target
    @approved_methods = approved_methods
  end

  def method_missing(name, *args, &block)
    if @approved_methods.include?(name)
      @target.send name, *args, &block
    else
      super
    end
  end
end

m = Media.new
dm = Delegator.new(m, :play, :stop)

dm.play #=> Playing...
dm.stop #=> Stopping...
dm.destroy #=> NoMethodError: undefined method 'destroy'
```

Lab 1.5
method_missing greeter

- Lab 1.5 you are asked to complete the implementation of method missing so that our Greeter class instances can dynamically respond to greet_in_* method calls

```ruby
class Greeter
  GREETINGS = {
    :english => 'Hello',
    :french => 'Bon Jour',
    :dutch => 'Hallo',
    :spanish => 'Hola'
  }

  def method_missing(method_name, *args)
   # Your code goes here!
  end
end

greeter = Greeter.new
p greeter.greet_in_dutch
p greeter.greet_in_spanish
```

```
"Hallo!"
"Hola!"
```

Thanks

http://integrallis.com