AUC Ruby Course 2011

Marcus Crafter & Gareth Townsend

Welcome!

- thanks for coming!
- great to see big interest in Ruby, Rails, tech, etc.
- GT and I work professionally with Ruby as part of our day job
- very excited about showing everyone Ruby, bit of a glimpse
- people have come from a long way - where is everyone from?
- we have a great few days lined up for you

Marcus Crafter

crafterm@redartisan.com
http://twitter.com/crafterm

First a brief intro
- from Melbourne, at least I say I come from Melbourne

The Blue Lake

- changes colour between winter from a grey to a bright blue in summer

Studied at La Trobe University in Melbourne

- 1992-1996
- Degree in Computer Systems Engineering with Honours
- Specialized in software engineering
    - Thesis was in Cross Platform User Interface Development
    - C/C++
- One of the first students to start developing with Java, released in 4th year
    - at the time thought was fantastic (C++ was the alternative)

OSA/ManageSoft transferred me to Germany in 1998

- If you get the opportunity to work overseas, definitely take it up!
- Frankfurt, financial capital of Germany and with the introduction of the euro in 2000, Europe.
- On site consulting at Dresdner Bank
    - Web development for the banks very wealthy
    - Portfolio management application
- Initial placement was for 9 months
- Ended up living in Germany for 8 years, returning to Australia in 2006
- Continued working with ManageSoft in Australia for a further year...

Ruby

- In 2003, I started learning and using Ruby, and I started to get really excited!
- Initially as a scripting language
    - Was using Java at the time as my primary development platform
    - Started using Ruby similar to how people were using Perl at the time
        - processing text with regular expressions
        - filesystem manipulation and scripting repetitive tasks
- Impressed with its readability, and its ease of use for common tasks
- Ruby use increased each year until I encountered Rails mid-end 2006
    - Became consumed by the technology..
    - Was spending all of my spare time with Rails until...

red () artisan

Early 2007 I founded my own company, Red Artisan.com

- Specializing in Ruby, and Ruby on Rails development
- Worked for most Australian Ruby/Rails shops in Melbourne and several international clients
    - CLEAR Web Solutions
    - Flashden/Envato
    - Square Circle Triangle
    - CSG Solar
    - and more.
- Traveled overseas to various conferences, RailsConf US/EU
- Best fun I've had in my career :)

Now work at NZX

- last client CLEAR, was acquired in 2009
- dot.com companies do happen in Australia with esoteric technology
- mainly financial apps
    - ruby on rails
    - ruby
    - iOS

Gareth Townsend

Software Developer

Background

- From Melbourne
- Lived in Papua New Guinea and Vienna
- Studied Software Engineering at RMIT
- Founded Melbourne CocoaHeads

Work Experience

- 5+ years professional Software Engineering
- Ruby on Rails, some .NET and CocoaTouch
- AUC training courses and Dev World

Work Experience

- Mac user since 2003
- Ruby and Rails since 2005
- Ruby and Rails professionally since 2007

Work Experience

- Hannan IT
- Six Figures
- Box + Dice
- CLEAR
- NZX

Conferences

- WWDC (3x)
- Web Directions South
- Dev World (speaker)
- YOW!

Lets look at a few Ruby apps!

What you can do with Ruby & Rails

Before we get into too many details
- Lets start off with a demo!

CLEAR

Australia’s Grain Exchange

Trading platform for Australia’s deregulated grain industry

Twitter

Instant online communication

- Twitter, instant world wide communication
  - We'll be building something to work with Twitter a bit later on :)
- Started as a Rails app
    - Now Rails, Scala, and other technologies

- New York plane crash, the best photos from the incident were found on Twitter - news pic
- Incredible, billion dollar news companies cant beat the speed of news Twitter can provide

yellowlab.com.au

Yellow Pages Testing Ground

Rails based implementation of Australia's yellow pages
- Pilots features that are added to yellowpages.com.au
- Rails application running under JRuby

Across all those examples
  - Mac OS X used as the development platform
- Ruby is a better citizen on the Mac these days than Java is
- Really good integration with Mac OS X, eg:
  - Domain and Keychain Authentication via Mac OS X/server
  - Hardware accelerated multimedia processing via Core Image/Animation/Video/Audio
  - Addressbook, Calendar and MobileMe application integration

The Ruby Value

- Competitive Edge
  - Increased development speed
  - Smaller development teams, lower costs
- Exciting Companies
  - Early adopters, startups, corporations
- Ruby on Rails has been a big catalyst for Ruby
- Active community conferences, books, user groups, etc

Cool, flexible and productive language
- naturally you can develop faster with it
- saves time, resources, -> money

Rails - huge catalyst for Ruby

Many professional conferences around the world

So many people & companies are getting into it
-> anyone in this room could get a job in ruby, today
-> as an employer finding people is hard
-> companies would hire you to train you in ruby (!!)

Course Agenda

General Overview

So what are we going to do here?

General Overview

- Ruby, the language and platform
- Assume no prior Ruby knowledge and start from the beginning
- Ruby is an easy language to get started with, but has plenty of depth to master
  - Today, language, environment, and get started writing code
  - Tomorrow, advanced topics, meta prog & DSLs

- Ruby, the language and platform (most of today)
  - Assume no knowledge of the language and start from the beginning
  - Will assume you're familiar with software development and have experience with at least one language
  - Ruby is an easy language to get started with, but has plenty of depth to master
    - Today we'll learn the language, standard libraries, and get started writing programs in Ruby
    - Tomorrow we'll cover advanced/awesome topics such as meta programming and domain specific languages

General Overview

- RubyCocoa
  - Ruby integration with Mac OS X/Cocoa
- Ruby on Rails
  - Web framework, catalyst for Ruby's popularity

Few sub-technologies we'll be investigating

- RubyCocoa
  - Ruby's excellent integration with Mac OS X/Cocoa
  - This afternoon
- Ruby on Rails
  - Web framework and a major catalyst for Ruby's popularity
  - Integration with databases
  - Tomorrow afternoon

General Overview

- We'll also be building things along the way
  - Twitter applications, command line & RubyCocoa
  - Rails web apps
- Time to experiment, questions, answers, and breaks

- ... and we'll be building some things along the way
  - Command line Twitter client
  - Mac OS X Desktop Twitter client using RubyCocoa
    - (http party, Twitter gem and a table, perhaps some animation somewhere?)
  - Image processing RubyCocoa application, and will convert it into a
    - Rails application, file uploads reusing the RubyCocoa processing images using Core Image
  - Some free break time to experiment, ask questions

So what is Ruby?

"Ruby is... a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write"

So what is Ruby?
- definition from ruby-lang.org
    - "Ruby is... a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write"
- Syntax we'll get to shortly, lets look at some other factors

http://flickr.com/photos/76128093@N00/542486031

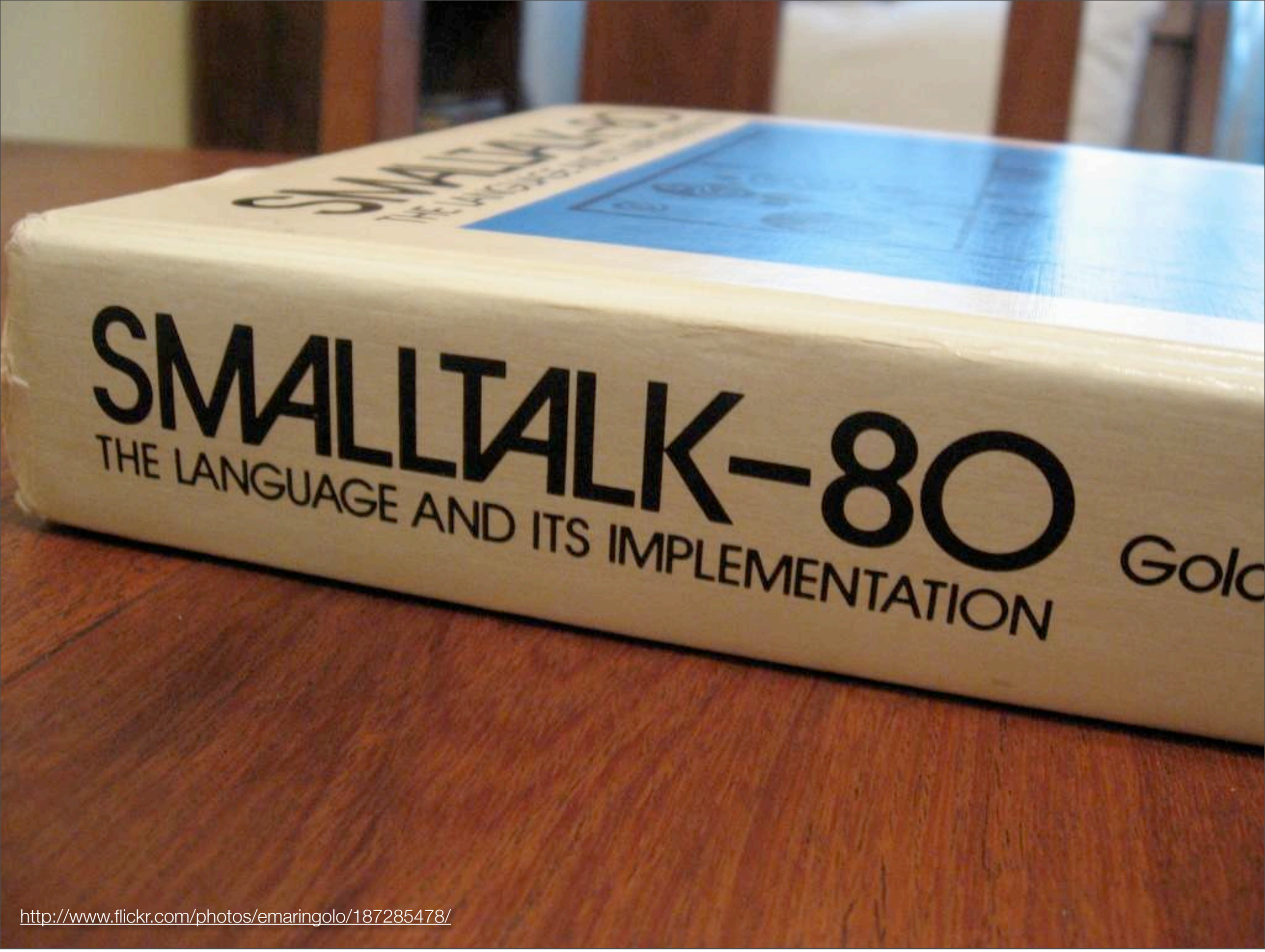
Invented by Yukihiro “matz” Matsumoto
- Simply “Matz”
- Japan 1995
- Started writing it to make his day job easier
- Taken at RubyKaigi 2007 one of the Ruby conferences around the world, in Japan

- “Anyone fancy a business trip to Japan? its on every year!” :)

What is Ruby?

- Interpreted, dynamic language
  - No separate compilation or build phase
- Dynamically typed
  - No declaration of types when you reference objects
  - No compile time type checking
    - All done at runtime
- Flexibility and power, very interesting things! :)

Unlike C/C++, there's no compilation phase to "build" anything
- that's done at runtime
- many things taken away, but lots given back in return



http://www.flickr.com/photos/emaringolo/187285478/

- Any smalltalkers here?
- Under the hood
    - Ruby (& Objective-C) share the same heritage - Smalltalk
    - many features in both languages inspired by Smalltalk - message passing, method missing, etc
    - In Snow Leopard, Objective-C has some more Ruby like features, eg blocks
- Will see why thats useful later on

What is Ruby?

- Blend of Matz's favourite languages such as Perl, Smalltalk, Eiffel, Ada, Lisp, and others
- Open Source
  - Free, as in speech, not beer
- Dual Licensed under the GPL or the Ruby license
  - Can be used commercially, modified, and distributed

Commercial advantage
- you can embed it in your product, or deploy it on your server
- Google Sketchup
- Mac OSX
  - one of the reasons Ruby is such a well looked after citizen on the mac

What is Ruby?

- Object Oriented - absolutely everything is an object
  - Including "*fundamental*" types

```
5.times { print "We *love* Ruby!" }
```

- Message based, unlike bounded languages
  - Object#send(:symbol) is the identical to a method invocation, similar to objc_msgSend
- Designed for humans, not machines

Everything is an object
- in low level languages like C, fundamental types often matched the underlying hardware (eg 32bit ints, 1 byte chars, etc)
    - in Ruby, there's no system level fundamental types, they're all objects
        - in older systems this would be slow, but modern hardware makes this a reality
- Many advantages to object based fundamental types
    - no overflow, underflow with ints - the equivalent object type, Fixnum/Bignum scale to the size of available memory
- Message based
    - messages aren't bound at compile time, but are realized at runtime
        - lower level languages bind a method call to a jump in memory
    - there's even a method .send, which lets you send a message programatically to an object
        - very interesting, lets us do a lot of powerful things that we'll see later

What is Ruby?

- Dynamic runtime
  - Define classes and methods at runtime, on the fly
- Can create new domain specific languages, eg:
  - Software construction (rake) or testing (rspec)
  - Software deployment (sprinkle)
  - Database manipulation (active record)

- Dynamic runtime
  - Can define classes and methods at runtime, and on the fly
    - Will see some of that tommorrow
    - Can even catch calls for methods that don't exist and define them as needed
  - Can write (new) domain specific languages that better capture a business domain
    - music (arx)
    - software construction (rake)
    - software deployment (sprinkle)
    - software testing (rspec)
    - database schemas and queries (active record queries and migrations)
    - other DSLs? the common part - its all Ruby code

Rake - Construction

```
desc "Source installer task"
task :install => :environment do |t|
    FileUtils.mkdir_p @target
    FileUtils.cp_r @source, @target
end
```

```
$> rake install
```


Software construction
- building environments
- compiling native code
- running tests
- performing system tasks

RSpec - BDD Testing

```
describe MyClass, 'when created' do
    it 'should be invalid without a name' do
        @instance.should validate_presence_of(:name)
    end
end
```

```
$> spec spec/models/myclass_spec.rb
```


Testing, behaviour driven development
- that reads just like english, but its real code

Sprinkle - Deployment

```
package :ruby do
    description 'Ruby Virtual Machine'
    version '1.8.7'
    source "ftp://ftp.ruby..org-#{version}-p111.tar.gz"
    requires :ruby_dependencies
end
```

```
$> sprinkle -c -s deployment.rb
```


Automating software deployment

Databases

```
class User < ActiveRecord::Base
    validates_presence_of :name
    validates_uniqueness_of :name
    validates_numericality_of :age, :greater_than => 0
end


User.find :all, :conditions => { :name => 'Steve' }
```


Active Record, ORM – object relational mapper

Here we're finding all the users with the name 'Steve'

All examples were pure Ruby

All of what we've just seen were pure Ruby

- no extensions
- just using Ruby language to define a new language via meta programming.

- We'll work through some of this advanced meta programming tomorrow.

Ruby Flavours

MRI, Rubinius, MacRuby, JRuby & IronRuby

There are actually several different flavours of Ruby available
- used in different environments, why?
    - deployment and technology environments
    - politics
    - reuse of existing code and libraries
    - etc

MRI - “Matz” Ruby

- Official source tree
- 1.8.x stable release, 1.9.x under constant development
- Interpreter, language and runtime written in C
- Standard library written in Ruby
- Interesting legacy and history
- 1.8.x green threads, 1.9.x native threads

The official Matz Ruby
- definitive source of what Ruby is and isn’t
- 1.8.x green threads
    - interesting, means operations are threaded ruby interpreter not the operating system
    - you could do Ruby multithreaded programming under MS-DOS :)
- 1.9.x released just last week
    - native threads, still work being done on the standard library
- considered slow compared to other interpreters, eg. Java

JRuby - Java Ruby

- First Ruby 1.8 port to a new environment
- First Ruby environment to offer native threads via JVM
- Runs Rails in production well
  - Deployment of Rails application as WAR files
- Competitive MRI performance

First new 1.8 port to a new environment
- JVM native threads
- Can be used to run Rails applications in production well
- For certain uses is faster than MRI
- Advantage for Java shops
    - politics
    - departments with entire java staff, etc

MacRuby

- Ruby 1.9 implementation written with Objective-C and CoreFoundation/Cocoa
- Very exciting, Ruby types are Cocoa types
- Cocoa garbage collection
- Sponsored by Apple, @lrz
- In active development
  - Mac side great, Ruby side improving, iOS tweets
- http://www.macruby.org

Implementation of Ruby on top of the Cocoa/Core Foundation API's
- Very exciting and interesting project
- With bridges you have to convert method invocations and types from one language to another, this can be a big performance penalty
    - Ruby classes are Objective-C classes, Ruby and Objective-C code can work with each other
    - Ruby types are Objective-C types (eg. String is NSString) - no conversion penalty required

Ruby Environment

/usr/bin/ruby, /usr/bin/irb, /usr/bin/ri, gems

So what do you “get” when you have Ruby?

/usr/bin/ruby

- Main Ruby binary
- Runs your Ruby program on the command line
- Generally named in the *she-bang* path of scripts

  #!/usr/bin/env ruby

/usr/bin/irb

- Interactive ruby console
- Allows you to write Ruby code directly into the interpreter
- Very useful for testing
- Provides syntax and command completion

IRB
- When I was coding Java I wished I had something like this.
- Easy to type in some simple code values and check things out without having to write a complete class/program
- Full access to the Ruby API

```
require ‘osx/cocoa’
diggy = OSX::NSSound.alloc.initWithContentsOfFile_byReference("tiger.m4a", false)
diggy.play
```

/usr/bin/ri & /usr/bin/rdoc

- Inbuilt documentation system
- Similar to manpages for Ruby methods, classes, etc

Gems

- Ruby library deployment system
- Packaging
- Dependencies
- Networking, remote installation

Similar to cpan for perl, ports for BSD, or easy-install for python
- sudo gem update --system
- sudo gem install twitter

- Bundler and RVM are further advancements in this area too

Ruby First Glance

Straight in the deep end!

Make sure everyone has a Mac?

Hello World

```
class Hello
  def say(rant)
    puts "hello #{rant}"
  end
end

Hello.new.say('world')

# => "hello world"
```


Lets start with a simple bit of code
- we'll type it into irb/textmate, students can also follow

There's a few things here:
- defined a class called 'Hello'
- defined a method that is called 'say'
- string interpolation, variable substitution

- What do you guys notice the most about the code? pretty colours?
- Notice no semi-colons at the end of each line

Some Comparisons?

Ruby

```
require 'book'
require 'glossy'
require 'softcover'

class Magazine < Book
  include Glossy, SoftCover

  attr_accessor :title, :content

  def initialize(title, content)
    @title = title
    @content = content
  end

  def to_s
    "#{@title}: #{@content}"
  end

end
```


- require brings other files into scope - can also be done dynamically via const_missing/ active support
- single class inheritance
    - include modules, traits based programming
- attr_accessor for defining accessors/getters
- initialize is the constructor
- to_s automatic string conversion

Objective-C

```objc
#import "book.h"
#import "glossy.h"
#import "softcover.h"

@interface Magazine : Book <Glossy, SoftCover>

@property (nonatomic, retain) NSString * title;
@property (nonatomic, retain) NSString * content;

- (id)initWithTitle:(NSString *)title andContent:(NSString *)content;

@end

@implementation Magazine
@synthesize title, content;

- (id)initWithTitle:(NSString *)aTitle andContent:(NSString *)someContent {
    if (self = [super init]) {
      self.title = aTitle;
      self.content = someContent;
    }
    return self;
}

- (NSString *)description {
  return [NSString stringWithFormat:@"%s: %s", self.title, self.content];
}

@end
```


- Implementation and interface are in separate files
- named parameters
- finer grained property generation control
- String concats are more verbose via stringWithFormat
- description is like to_s in Ruby
- we're actually missing something here in the Objective-C version - dealloc/release of the instance variables
    - in an ARC world things are different.

Java

```
import com.company.books.Book;
import com.company.books.printing.styles.Glossy;
import com.company.books.covers.SoftCover;

public class Magazine extends Book implements Glossy, SoftCover {

  public Magazine(String title, String content) {
    this.title = title;
    this.content = content;
  }

  public void setTitle(String title) {
    this.title = title;
  }

  public String getTitle() {
    return title;
  }

  // ... accessors/getters for String content ...

  public String toString() {
    return title + ": " + content;
  }

  private String title;
  private String content;
}
```


- get/setters define manually (does Java support this yet?)
- access control defined on each method/variable
- Similar to Objective-C with interfaces/protocols
- constructor names the parameters
    - Ruby has default parameter names, less verbose
- toString is like to_s in Ruby

CoffeeScript

```
class Magazine extends Book
  constructor: (@title, @content) ->

  to_s: ->
    “#{title}: #{content}”
```


Ask Gareth

- CoffeeScript, Ruby inspired generator for JavaScript

JavaScript

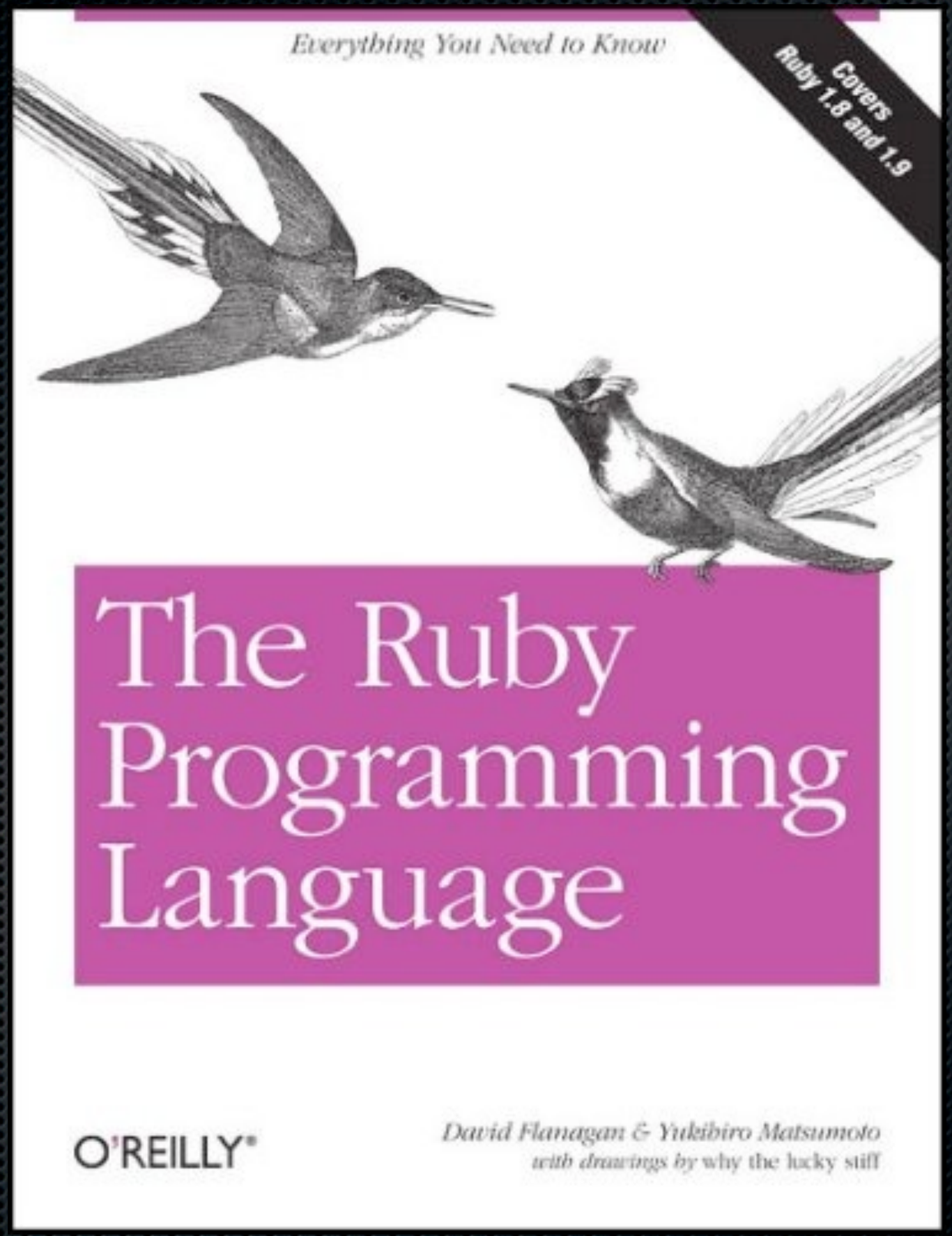
```
var Magazine;
var __hasProp = Object.prototype.hasOwnProperty, __extends = function(child,
parent) {
  for (var key in parent) { if (__hasProp.call(parent, key)) child[key] =
parent[key]; }
  function ctor() { this.constructor = child; }
  ctor.prototype = parent.prototype;
  child.prototype = new ctor;
  child.__super__ = parent.prototype;
  return child;
};
Magazine = (function() {
  __extends(Magazine, Book);
  function Magazine(title, content) {
    this.title = title;
    this.content = content;
  }
  Magazine.prototype.to_s = function() {
    return “;
  };
  return Magazine;
})();
```


Ask Gareth

- CoffeeScript, Ruby inspired generator for JavaScript

Ruby The Language

The Details!



This is the book we're using for the course
- written by the creator of Ruby, Matz and David Flanagan
- the most definitive resource for the ruby language
- up to date, covers Ruby 1.8 and 1.9, the language, environment and standard library
- does not cover applications, other libraries like RubyCocoa, MacRuby, etc

Bookshelf Example

- Domain is easy, everyone is familiar with a bookshelf
- We'll reuse it again tomorrow too
- We'll define a Bookshelf class, which holds all our books
    - then we'll define the items that go in it
- line by line, together, taking time to explain parts of the language

```
class Bookshelf
end
```


- descend from Object automatically, everything does
- single inheritance only
- methods defined are public by default
- conventions are to have an uppercase letter for classes

```
class Bookshelf < Object
end
```


- < specifies inheritance
- Bookshelf is a sub class of Object
    - is-a relationship
- but for subclasses of Object, its optional

```
class Bookshelf
end
```


- so we can leave it with this

```
class Bookshelf
  def initialize
    @items = Array.new
  end
end
```


- Constructor, always called initialize
  - called automatically when building an instance
- 0 or more arguments, here we have none
- notice the optional parentheses
- @ defines an instance variable
  - notice no type information
  - variables appear as required - no declarations required
  - memory management is automatic, garbage collected, no retain/release
- Creating an instance of an object
  - send ‘new’ to the class

```
class Bookshelf
  def initialize
    @items = []
  end
end
```


- [ ] shorthand for creating a new instance of an Array

- also notice instance variables are private by default

```
class Bookshelf
  def initialize
    @items = []
  end

  def add(item)
    @items << item
  end

  alias << add
end
```


- def, defines a method, just like the constructor
- methods return the value of the last statement
    - unless a return statement is used or an exception raised
- methods are public by default
- everything in Ruby passed by reference, no pass by value

- << is an operator, and you can overload them as well like in C++
- here, << is pushing an object referenced by ‘item’ into the @items array

- alias lets us define another name for the same method,
    - here we define << to also mean the same as ‘add’
    - subtle, you can alias an existing method on another class, and insert some code before or after it

```
class Bookshelf

  def contents
    descriptions = Array.new
    for item in @items
      descriptions << item.description
    end
    descriptions
  end


end
```


- Before typing this all in, we'll talk about it first..
- Array.new is the same as [ ]
- for loop, iterating over all the elements in the @items
- calling the method 'description' on the item
- storing the result in a 'descriptions array
- the return value is the last line of the method

- 'collecting' the results
- valid ruby, but not often written

```
class Bookshelf

  def contents
    descriptions = Array.new
    @items.each do |item|
      descriptions << item.description
    end
    descriptions
  end

end
```


- ‘each’ is an enumerator, part of the Enumerable module on collections
- accepts a block of code as a parameter
    - yields each element to the block
- blocks in ruby are similar to those in Objective-C
    - an optional ‘code’ parameter to a method
    - lisp & smalltalk, very powerful, we’ll see more in the DSL work tomorrow

- also valid ruby, but we’re essentially collecting the results, which is a common pattern
    - there’s a method for that

```
class Bookshelf

  def contents
    @items.collect do |item|
      item.description
    end
  end

end
```


- ‘collect’ is like ‘each’
  - it stores the result of each operation in an array
  - exactly the same as the previous slide
  - abstracted the ‘collection’ of values from a collection (!)

- remember the return value rule
- whats the last line of code in the ‘contents’ method?
  - collect is, this is actually one line of code
- collect results are returned from ‘contents’

```
class Bookshelf

  def contents
    @items.collect { |item| item.description }
  end

end
```


- block's can be written with 'do/end' or { }
- much like Objective-C
- good (ruby?) code is often only a few lines
    - see the code we're not writing
    - this one you can type in

- its possible to get shorter again
    - @items.collect(&:description)

- ok we have our bookshelf

```
class Bookshelf
  def initialize
    @items = []
  end

  def add(item)
    @items << item
  end
  alias << add

  def contents
    @items.collect { |item| item.description }
  end
end
```


- should have something like this

- we might want access to our items outside of the bookshelf?

- you might want to set or get the items directly
    - ie. getters and setters

```
class Bookshelf

  def items
    @items
  end

  def items=(items)
    @items = items
  end

end
```


- this would be how you'd write them
    - again, don't type this one in just yet
- this is abstracted as well

```
class Bookshelf

  attr_reader :items
  attr_writer :items

end
```


- getter
    - attr_reader

- setter
    - attr_writer

- if you want both:

```
class Bookshelf

  attr_accessor :items

end
```


- see how we keep reducing code back to only a few lines

- now, whats that funny : before the items
- this is actually a symbol
    - symbol is a constant unique fast lookup string
    - like having a reference to a string value of the same name

```
class Bookshelf
  attr_reader :items

  def initialize
    @items = []
  end

  def add(item)
    @items << item
  end
  alias << add

  def contents
    @items.collect { |item| item.description }
  end
end
```


- ok, should have something like this - our Bookshelf class
    - notice how small it is

- we need to put something into it

```
class Book
end
```


- again class, descending from Object

```
class Book
  attr_accessor :title, :pages, :publisher


end
```


- we'll define 3 attributes
- title, string
- pages, integer count
- publisher, string

```
class Book

  attr_accessor :title, :pages, :publisher

  def initialize(title, pages, publisher = 'Unknown')
    @title = title
    @publisher = publisher
    @pages = pages
  end


end
```


- another constructor

- notice, methods can accept default values
  - publisher is optional
  - this object can accept, 2 or 3 parameters

```
class Book


  attr_accessor :title, :pages, :publisher


  def initialize(title, pages, publisher = 'Unknown')
    @title = title
    @publisher = publisher
    @pages = pages
  end


  def description
    "#{self.class.name}: #{@title}, published by #{@publisher}, #{@pages} pages"
  end


end
```


- “ “ returns a string
- #{ } is variable interpolation

- so what is this ‘self’ thing
    - self refers to the current object

- self is also the default receiver, meaning its mostly optional
- here we call a method called ‘class’ to get the ‘Class’ object, and its name
    - like in Objective-C a Class object is an object that represents a type

```
class Bookshelf # .....
end


class Book # .....
end
```

```
shelf = Bookshelf.new

shelf << Book.new('The Ruby Programming Language', 350, 'OReilly')

shelf.add(Book.new('The Ruby Way', 201))

puts shelf.contents
```


- Ok, so now we've defined our Bookshelf, and Book class

- lets create an instance
  - our bookshelf
  - 2 books

- notice we're using the default value for the 'The Ruby Way', we don't know who published it
- notice we can use .add or << to put a book on the shelf, both do the same thing
  - << is perhaps a bit more readable

```
$> ruby bookshelf.rb

Book: The Ruby Programming Language, published by OReilly, 350 pages
Book: The Ruby Way, published by Unknown, 201 pages
```


- Ok, lets run it
    - should have some output like this

- notice, Book: is printed, because we dynamically asked what the name of the class was

- Ok, lets keep going
    - I also have some DVD's on my bookshelf

```
class Dvd
end
```


- again class, descending from Object

```
require ‘date’

class Dvd
  attr_accessor :title, :director, :release_date

end
```


- we’ll define 3 attributes
- title, string
- director, string
- release_date, date

- what’s this ‘require’ business
    - lets you bring in other libraries, code from gems, standard library, etc
    - by default, ‘date’ is in date.rb, and there’s a load path to find these files

```
require ‘date’

class Dvd
  attr_accessor :title, :director, :release_date

  def initialize(title, director, release_date = Date.today)
    @title, @director, @release_date = title, director, release_date
  end


end
```


define a constructor
- default release_date attribute of today
- notice the one liner with parallel assignment
    - ruby lets you define and assign multiple values at once

- you can usually bring that one out to show off when pair programming

```
require ‘date’


class Dvd
  attr_accessor :title, :director, :release_date


  def initialize(title, director, release_date = Date.today)
    @title, @director, @release_date = title, director, release_date
  end


  def description
    "#{self.class.name}: #{title}, directed by #{director}, released #{release_date}"
  end


end
```


- define the description method
  - notice anything different
  - no @title, its just ‘title’
  - this is subtle, but we’re actually using the accessor rather than the ivar direct.
  - remember self is the default receiver
    - checks whether title is a local variable or a method automatically

```
class Bookshelf # .....
end


class Book # .....
end


class Dvd # .....
end


shelf = Bookshelf.new
shelf << Book.new('The Ruby Programming Language', 350, 'OReilly')
shelf << Book.new('The Ruby Way', 201)
shelf << Dvd.new('Inception', 'Christopher Nolan')
```


- notice we use the default release date value for the DVD

```
$> ruby bookshelf.rb

Book: The Ruby Programming Language, published by OReilly, 350 pages

Book: The Ruby Way, published by Unknown, 201 pages

Dvd: Inception, directed by Christopher Nolan, released 2011-07-07
```


Should see something like this:

- Great we have Books and DVD's.
- There's something subtle going on here between Books and DVD's which might not be obvious
    - can anyone tell?
- Lets step back and analyse some code

```
class Book


  attr_accessor :title, :pages, :publisher


  def initialize(title, pages, publisher = 'Unknown')
    @title = title
    @publisher = publisher
    @pages = pages
  end


  def description
    "#{self.class.name}: #{@title}, published by #{@publisher}, #{@pages} pages"
  end


end
```


- Book has a description method

```
require ‘date’

class Dvd

  attr_accessor :title, :director, :release_date

  def initialize(title, director, release_date = Date.today)
    @title, @director, @release_date = title, director, release_date
  end


  def description
    "#{self.class.name}: #{title}, directed by #{director}, released #{release_date}"
  end


end
```


- Dvd has a description method

- both Book and DVD are different Classes
  - they’re subclasses of Object, but there’s no ‘description’ method defined on Object

```
class Bookshelf

  def contents
    @items.collect { |item| item.description }
  end

end
```


- yet, when we put both Books & DVD's into the @items, we can call 'description' on either

- how does this work?
  - java, obj-c you'd need a common abstract class with a method on it
    - because you'd need to cast the object to a type, and then invoke the method

- In ruby, this works because its dynamic, there's no types and due to something special

- Duck typing

If it walks like a duck, and smells like a duck, its a duck!

- if it walks like a duck, and smells like a duck, then its a duck.

- in ruby this means
  - its not important what object type something is
  - its important that the object responds to the message
  - if the object responds to the message, its a duck (ie. good enough), and ruby will call it

- really useful, particularly when marrying up classes in libraries you cant change, etc
  - in obj-c, java you'd have to write proxy or adaptor classes, etc

Ok, your turn :)

- Implement some other things on the bookshelf
- Wine Bottles
  - Brand, Vintage, Grape, etc
- Magazines
  - Similar to books, but with a magazine issue number

- If you're bookshelf is like mine there wine
- .. and magazines

- 10 minutes, and a break

```
class Magazine

  attr_accessor :title, :pages, :publisher, :issue

  def initialize(title, pages, issue, publisher)
    @title = title
    @pages = pages
    @issue = issue
    @publisher = publisher
  end

end
```


- Ok, taking a last look at the bookshelf
    - one option is to build it like this

- very similar to a Book, in fact, the only addition is the ‘issue’
- a Magazine is-a Book
- we can use inheritance to extend a Book into a Magazine

```
class Magazine < Book
end
```


- this time we’re descending from Book

```
class Magazine < Book

  attr_accessor :issue

  def initialize(title, pages, issue, publisher)
      super title, pages, publisher
    @issue = issue
  end


end
```


- Constructor
  - notice we re-use the constructor from the Book
    - ‘super’

```
class Magazine < Book

  attr_accessor :issue

  def initialize(title, pages, issue, publisher)
    super title, pages, publisher
    @issue = issue
  end

  def description
    super << ", issue #{issue}"
  end

end
```


- notice in ‘description’
    - we extend the implementation of Book’s description, by adding issue to the end

- class based inheritance

Extra bits & pieces

Data types, classes, method, etc

Integers

- Integer, base class for all integers
- Fixnum, if it fits within 31 bits

```
5.class # => Fixnum
```

- Bignum, if larger

```
12345678890.class # => Bignum
```

- Fixnum to Bignum converted automatically

Integer is the base class, concrete classes are

- Fixnum, (usually) up to 31 bit representation
- Bignum, can be as big a memory in the system

Different to other languages, particular compiled languages
- Integers often 16, or 32 bits specific depending on platform
- Program can work on one platform and not another
- No overflow in Ruby

Floats

- Floating point numbers
  - IEEE-754 specification
    - Binary representation for efficiency
      - Not precise (example)
- BigDecimal class
  - Decimal representation, unlimited size
  - Financial applications, etc

Floats

- Are represented in binary so have inherent precision errors
- demo 0.4 - 0.3 == 0.1 # => false

BigDecimal
- Uses decimal representation and can be used in applications where precision/round is important, eg, financial applications

Hashes

- Unordered key/values pairs
- {}

```
hash = { :dog => ‘fred’, :cat => ‘bonny’ }
```

- Hash.new

```
hash = Hash.new
hash[:dog] = ‘fred’
hash[:cat] = ‘bonny’
```

- {} are optional, sometimes

```
pets.define(:dog => ‘fred’, :cat => ‘bonny’)
```


- Hashes, very useful, used all the time
  - machine order, not the order you specify
- {} creation, short form for Hash.new
- {} can be optional at times
  - We’re calling a method on an object called pets, defining or dog and cat names
  - we’ll see a bit later but that statement can actually be further reduced to:
    - pets.define :dog => ‘fred’, :cat => ‘bonny’ - parenthesis are also at times optional

Usual Suspects

- Common hash operations supported

```
h = { :dog => 'fred', :cat => 'bonny' }
h.each do |key, value|
  puts "My #{key} name is #{value}"
end

h[:turtle] = 'preston'

h.has_key? :rabbit # => false
h.size             # => 3
h.values           # => [ 'fred', 'bonny', 'preston' ]
```


Check Ruby docs for more details

- enumeration, usually via blocks, we'll get to that construct soon in Enumerable
- accessing items
- adding/removing items
- objects used as keys and values, eg. numbers also supported, nil, etc.

- notice the '?' on the has_key

True, False and nil

- true, false and nil are all keywords in Ruby
- Everything is true, except for the keyword false and nil
- There is no Boolean class in Ruby, when required
  - nil behaves like false
  - any object except for nil and false behave as true

Examples of truth

```
# hash returns nil if no key matches
name = pets[:dog]
puts "i have a dog, its name is #{name}" if name

# any object can be tested if its nil?
puts "no dogs unfortunately" if name.nil?

# string evaluating to a boolean
if 'a'
  puts "the string 'false' evaluates to true"
end
```


First one, conditionally print a string if the name exists

Second one, checks for nil?, prints a string if no name exists

Third one, contrived example, 'false' will evaluate to string
- makes sense though, only the keyword false and nil evaluate to false, everything else is considered to be true

This is Ben :)

For a light break :)

- There's a picture of my dog while I was growing up
- golden cocker spaniel :)
- anyone here have any dogs? :)

Blocks

Smalltalk'isms in Ruby

Blocks

- Fundamental to Ruby's style and conventions
- Segment of code, that can be passed as an argument to any method call
- Are closures over references outside the blocks scope
- Can accept variables, and return a value
- Accessible by the developer using yield or block.call

Segment of code that can be passed as an argument to any method call
- smalltalk and lisp have this concept,
    - code is data
- somewhat similar to anonymous methods
- we've seen a few examples

Examples

```
cars.each do |vehicle|
  vehicle.send :register
end
```

```
# ...

content = “some book text”

File.open('book.txt','w') do |f|
  f << content
end
```


Some example method calls that use blocks

- first 2 are iteration
- third is IO manipulation
    - example advantage of accepting a block of code
    - IO opens the file descriptor, yields the block, automatically closes the file descriptor
    - framework writers love it, and you’ll see it all through the Ruby API

Block Development

```
class Car
  def parts
    @parts.each do |part|
      yield part if block_given?
    end
  end

ferrari = Car.new

ferrari.parts do |part|
  puts "My ferrari has #{part}"
end
```


Example class that defines a method that accepts a block
- yield invokes the code inside the block, passing any arguments through to the block
- here we're defining a method that enumerates all the parts of a car, calling the block for each part.
    - notice the enumeration remains inside the car code, not outside in user code which you would have to do for java, etc.

Enumerable

- select
  - Find all members where the block evaluates to true
- reject
  - Find all members where the block evaluates to false
- collect
  - Invoke the block on all members, return results array
- inject
  - Inject an object into a block called on all members

Enumerable is a module that includes many methods for operating on collections
- uses blocks a lot
- we'll see some examples next

Select

```
cassie = Dog.new('Cassie')
alf    = Cat.new('Alf')
ben    = Dog.new('Ben')

pets   = [ cassie, alf, ben ]

pets.select { |pet| pet.is_a? Dog }

# => [ cassie, ben ]
```


Notice here we're selecting those elements of the array that are of class Dog
- reject is the opposite
- and inject is for wizards

Inject is very powerful and magical :)

Inject

```
cassie = Dog.new('Cassie', ‘cocker spaniel’)
alf    = Cat.new('Alf', ‘house cat’)
ben    = Dog.new('Ben', ‘spaniel’)

pets   = [ cassie, alf, ben ]

pets.inject({}) {|m, pet| m[pet.breed] = pet; m}

# => { 'cocker spaniel' => cassie,
       'house cat' => alf,
       'spaniel' => ben }
```


Usually use it when you want to enumerate a collection and keep a running total/value.

Here we’re creating a new hash, where they key of the hash is the breed of the pet.

Inject takes an object as a parameter, and calls the block for each object in the collection being enumerated, passing the block the object from the collection and the value passed as an argument, the return value of the block becomes the next iterations argument value.

After all these examples involving pets, you've met Ben, this is Alf btw, he lives in Germany :)

Methods

Definitions, Visibility, Constructors, etc

Method Conventions

- Method names are lowercase
- Method names use underscore notation
  - define_method, not defineMethod
- Method names may end in ? or !
  - ? indicates a boolean return type
  - ! indicates an direct modification of the receiver
- Parenthesis are optional for declaration and invocation

```
public class Car {

  public static final String DEFAULT_MAKE  = "Audi";
  public static final String DEFAULT_MODEL = "TTS";

  public Car() {
    this(DEFAULT_MODEL);
  }

  public Car(String make) {
    this(make, DEFAULT_MODEL);
  }

  public Car(String make, String model) {
    this(make, model, new Date());
  }

  public Car(String make, String model, Date year) {
    this.make = make;
    this.model = model;
    this.year = year;
  }

  private String make;
  private String model;
  private Date year;
}
```

Java Chaining

Default arguments help a lot with constructors
- here some example Java code i used to have to write all the time
- same in Objective-C with init methods
- 4 constructors, supporting different styles of parameters
- chained constructors, so that only one does the real work the others pass values through.

Ruby

```
class Car
  attr_accessor :make, :model, :year
  def initialize(make = 'Audi', model = 'TTS', year = Time.now)
    @make  = make
    @model = model
    @year  = year
  end
end
```


Exact same behaviour in Ruby

- actually does more with the auto generation of getters/setters

Dynamically adding methods

```
class String
  def parenthesize
    "(#{self})"
  end
end

"Ruby!".parenthesize # => “(Ruby!)”

# or

1.day.ago
5.days.from_now
```


Here we open up the existing String class
- define a new method called parenthesize that adds parenthesis to the string
- inside variable interpolation, to_s is called automatically
- all strings, including existing ones inherit this new feature

- in Java you would usually use a StringUtils class to do this
- Objective-C has a similar feature via Categories

- also known as monkey patching

Ruby Koans

More code!

– GT

Twitter 1.0

Command Line Version

- Social networking micro-blogging platform
  - 140 character limit on posts
- Thousands of contributors around the world
- Fastest networking platform available

Interface

- Command Line Client
- Twitter Gem
  - Abstracts net/http networking access to Twitter
  - README.md is in /Library/Ruby/Gems/1.8/gems/twitter-1.6.0
- Access the public timeline
  - Display the 50 most recent public tweets

Design

- Require rubygems and the twitter gem
- Create a Twitter class, with a method public_tweets
- Use the Twitter gem to access the public timeline
- Display the tweets to the console

Example

```
require 'rubygems'
require 'twitter'

class Tweeter

  def public_tweets
    fetch_tweets.each do |s|
      puts "#{s.user.name}: #{s.text}"
    end
  end

  private

    def fetch_tweets
      Twitter::Client.new.public_timeline
    end

end

Tweeter.new.public_tweets
```

Further enhancements

- Design document available on public shared drive
- Support for reading a particular users tweets
- Support for posting new tweets from the command line
  - Requires valid credentials/account