AUC Ruby Course 2009

Marcus Crafter & Gareth Townsend

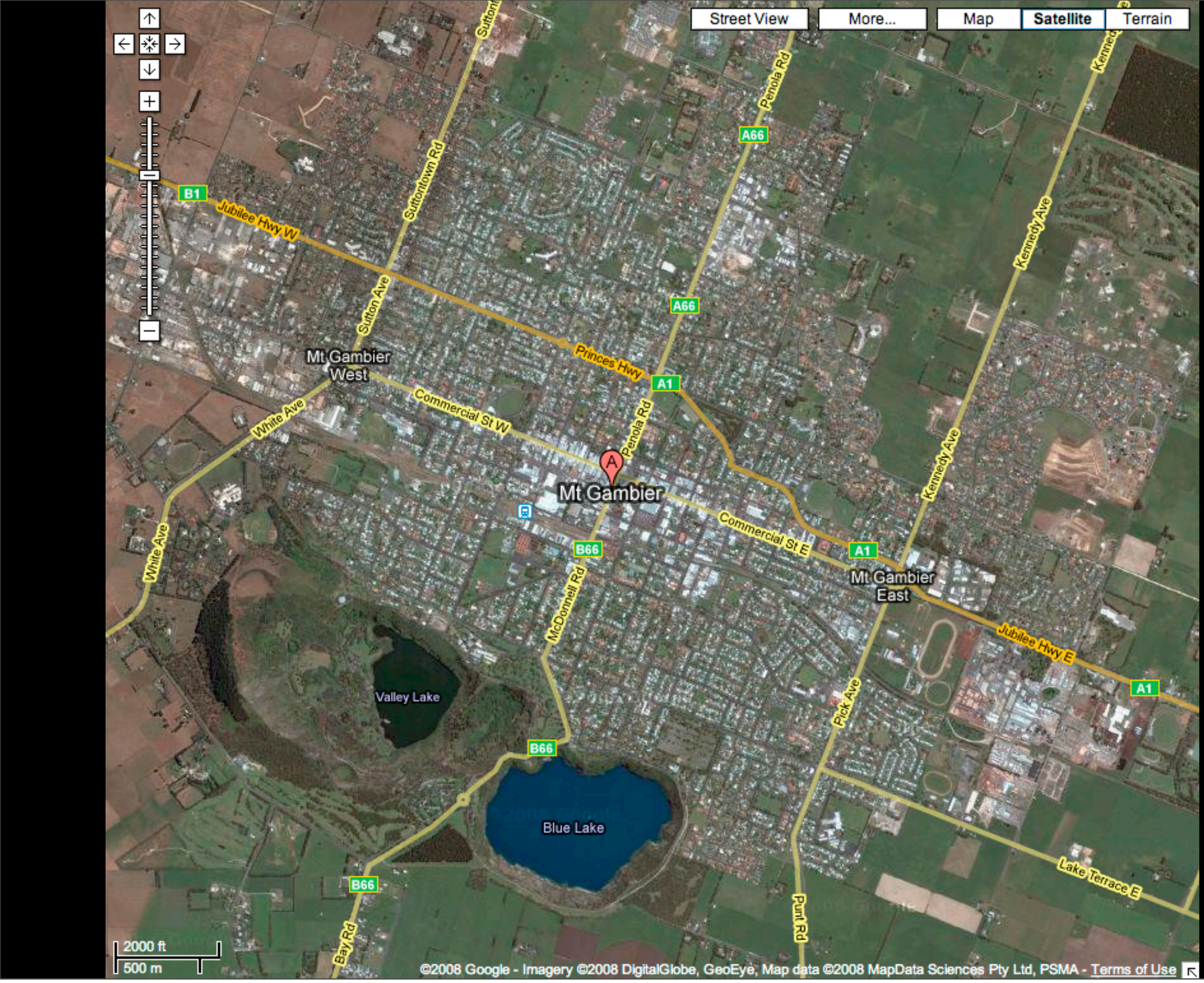
Welcome!

- thanks for coming!
- 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

Introduction, and a bit of background about me :)
- from Melbourne, at least I say I come from Melbourne



Monday, 9 March 2009

Originally From Mount Gambier in South Australia

- Along the border between Victoria and South Australia
- Anyone know where that is?
- ~30,000 people
- Primary and high school life
- Blue Lake, extinct volcano, natural spring water on tap

Monday, 9 March 2009

The Blue Lake

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

Monday, 9 March 2009

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)

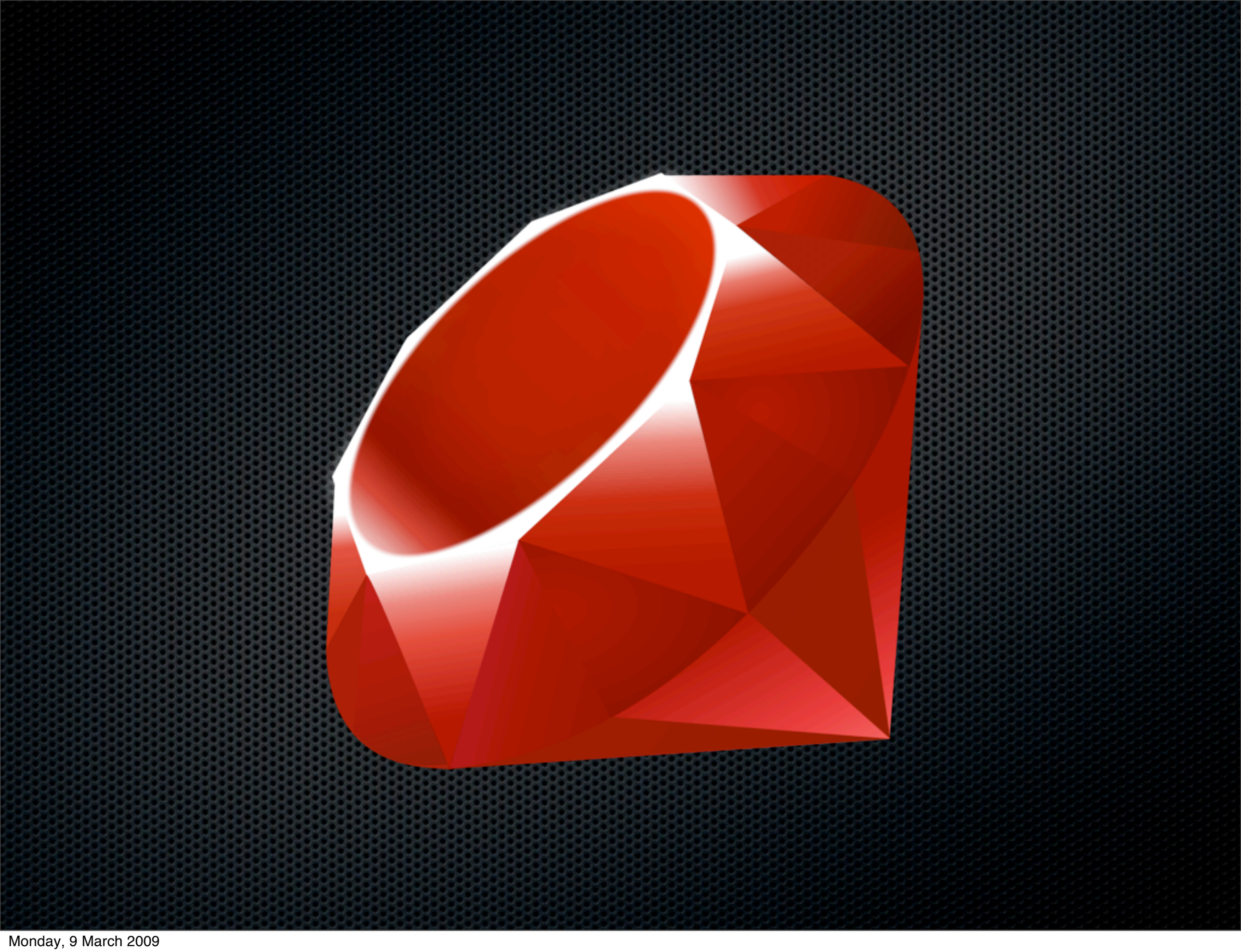
Monday, 9 March 2009

My first job was with Open Software Associates
- Ringwood, Melbourne
- Later renamed company to ManageSoft following product rebranding
- Specialized in enterprise level software deployment
- Worked in their cross platform development branch
- Java and..

(need to find a better image somewhere)

Monday, 9 March 2009

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

crafterm@redartisan.com

Monday, 9 March 2009

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

- Specializing in Ruby, and Ruby on Rails development
- Worked for several Australian and 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 :)

Monday, 9 March 2009

Little bit of computer history for me! :)

First Computer - Texas Instruments TI99/4A, that one didn't run Ruby :)

Monday, 9 March 2009

Second Computer - very boutique, Sharp MZ 700 - this one didn't run Ruby either :)

Monday, 9 March 2009

Amstrad PC2086, this one would have run Ruby 1.4 or 1.6... :)
- Ran DOS!

Monday, 9 March 2009

Amiga 500 - This one runs Ruby :)
- even today, can still run Ruby on an Amiga

Mac OS X Powerbook, MacBook Pro,
- Ruby screams on this one

Monday, 9 March 2009

A few final words before we get started

- ..Brave, we have lots of material
  - perhaps even too much, Gareth and I will adjust appropriately
- ..Excited, we're learning cutting edge tech, really exciting language
  -
- ..Afraid, you may never look at programming languages quite the same again
- ..Challenged, we have many subjects to cover, some quite complex and challenging, hang on, and you'll learn a lot :)
- ..Sober, we'll grab a few drinks tonight, lets make sure we're sober tomorrow morning when we tackle meta programming
-

Gareth Townsend

Software Developer

Background

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

Work Experience

- 3 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

Conferences

- WWDC (twice)
- Web Directions South
- Dev World (speaker)

Lets start with a Demo!

What you can do with Ruby

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

Archaeopteryx

MIDI control device for DJ'ing

Uses Ruby to define a meta language for controlling generation of music sequences thats fed into a MIDI player

CLEAR

Australia's Grain Exchange

Monday, 9 March 2009

Trading platform for Australia's deregulated grain industry
- Gareth and my current employer

Twitter

Instant online communication

- Twitter, instant world wide communication
  - We'll be building something to work with Twitter a bit later on :)

Monday, 9 March 2009

- 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

Monday, 9 March 2009

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

Mac OS X used as the development platform for all these products

- Allows for great 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

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
- Large community conferences, books, user groups, etc

Cool, flexible and productive language
- naturally you can develop faster with it
- saves time, resources, -> money
So many people are getting into it
Rails - huge catalyst for Ruby
Many professional conferences around the world

Course Agenda

General Overview

Monday, 9 March 2009

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
  - Database integration

- 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
    - Rails Address Book manipulation application
  - Time to experiment, questions and answers

- ... 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

Todays Plan

Thursday 19th February

- Start 9am
  - Intro, demos, agenda
- Morning Tea 10.30 am
  - Ruby environment, language and platform
- Lunch 12-1 pm
  - Hello World, Command line Twitter Client
- Afternoon Tea 3 pm
  - Further Ruby language, RubyCocoa && Twitter Client
- Finish 5 pm

Monday, 9 March 2009

- Todays Plan (approx. times)
  - Start 9am
  - Intro, agenda, demos
- Morning Tea 10.30
  - Ruby environment
  - Ruby the language and platform
- Lunch 12.00
- Reconvene 1pm
  - Hello World
  - Twitter 1.0 - command line twitter client
- Afternoon Tea 3pm
  - Ruby the language
  - RubyCocoa
  - Twitter 2.0 - RubyCocoa twitter client
- Finish 5pm

Tomorrows Plan

Friday 20th February

- Start 9am
  - Short review, advanced ruby, cool cats code
- Morning Tea 10.30 am
  - RubyCocoa Twitter
- Lunch 12-1 pm
  - Ruby on Rails, first Rails application
- Afternoon Tea 3 pm
  - Address Book processing via RubyCocoa
  - Summary, Q&A, feedback

Monday, 9 March 2009

Plenty of time along the way for questions, etc, please feel free to ask

- Start 9am
  - Quick review thus far, Q&A
  - Advanced Ruby - the cool cat's code
- Morning Tea 10.30
  - Ruby on Rails, the web framework
  - Hello World Rails App
- Lunch 12.00
  - Ruby integration with Databases under Mac OS X
  - Core Image processing via RubyCocoa
- Reconvene 1pm
  - Rails app with Core Image
  - Summary, questions & answer, feedback, etc.
- Finish 5pm

- Still needs a bit of work for the Friday to ensure we've covering all content we've planned for, more useful content from the RubyCocoa book might be more appropriate

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

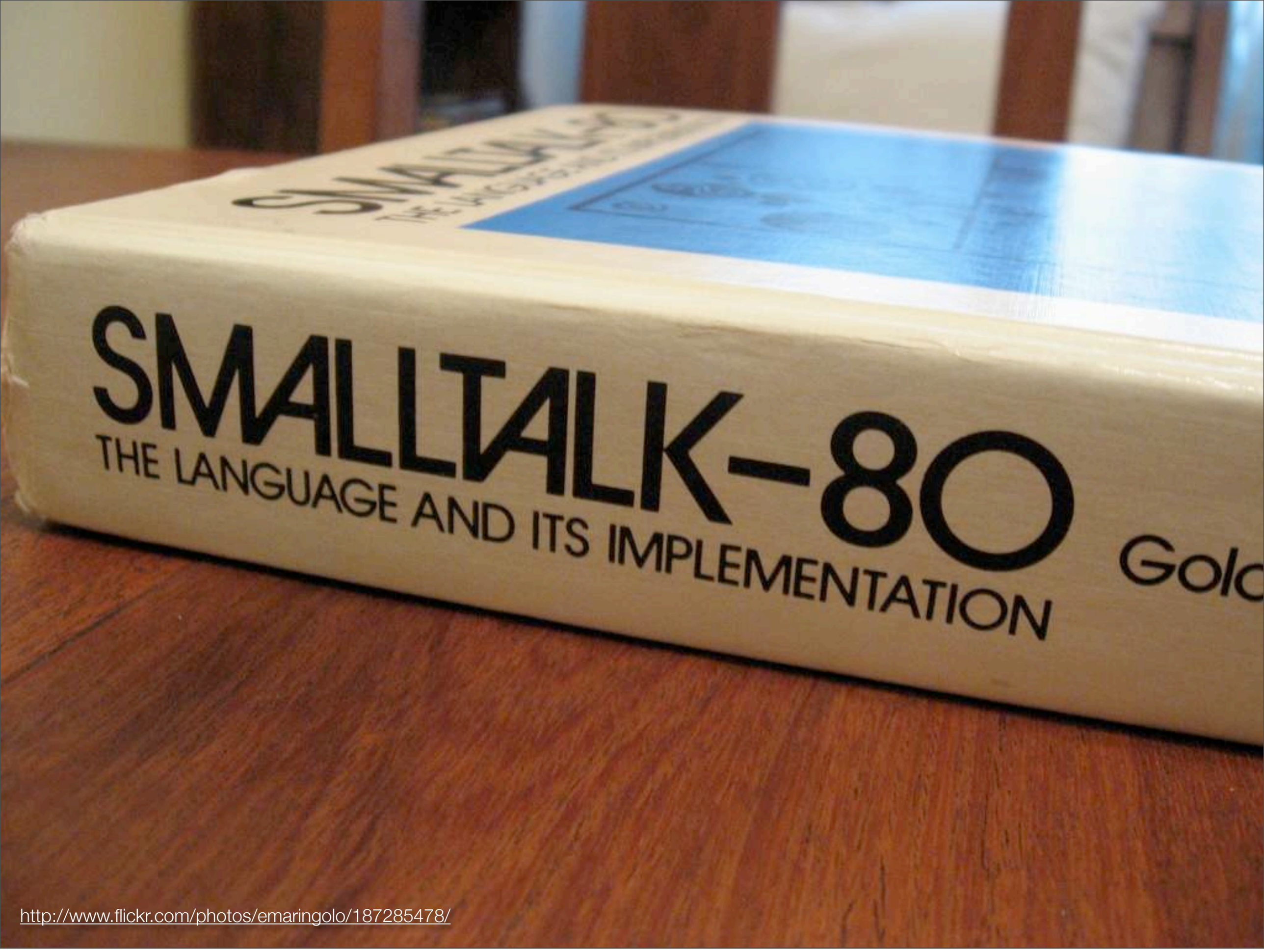
Invented by Yukihiro “matz” Matsumoto – Simply “Matz”

- 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

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
- thats done at runtime
- many things taken away, but lots given back in return



- Any smalltalkers here?
- Under the hood
    - Objective-C and Ruby share the same heritage - Smalltalk
    - many features in both languages inspired by Smalltalk - message passing, method missing, etc
    - In Snow Leopard, Objective-C getting some more Ruby like features
- 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
- A new language that balanced functional programming with imperative programming
- Students will notice features from other languages along the way, particularly Smalltalk and Perl

Matz blended together concepts and features from many other languages
- Functional programming, based on mathematical functions and declarative syntax
- Imperative programming, based on state and mutable data

What is Ruby?

- Open Source
  - Free, as in speech, not beer
- Licensed under the GPL or the Ruby license
  - Can be used commercially, modified, and distributed

What is Ruby?

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

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

- Message based
  - Object#send(:symbol) is the identical to a method invocation

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
        - need to be careful with this power, TDD/BDD is important

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
    - 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
```

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 in 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

Rubinius - Smalltalk inspired

- Completely new Ruby 1.8 VM written based on the Smalltalk Blue Book architecture
- Small fast and tight VM written in C++
- Language, standard library and environment written in Ruby
- Created the Ruby Test project to define what the Ruby language is
- Can run a test Rails application

- Exciting project
- Creating a new Ruby VM based on the Smalltalk blue book implementation
- Before Rubinius there was no official definition of what Ruby was
    - No spec, test suite, only what ran in Ruby 1.8.x
    - Ruby Test is an RSpec based project that aims to write a test suite that defines what the language, standard library and environment is
    - Now split into a separate project, reused by JRuby, etc

IronRuby

- Ruby 1.8 implementation based on Microsoft's Dynamic Language Runtime
- Sponsored by Microsoft
- Still in active development
- http://www.ironruby.net/

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
  - Deployment of Rails application as WAR files
- Competitive MRI performance

First new 1.8 port to a new environment
- JVM uses native threads
- Can be used to run Rails applications in production
- For certain uses is faster than MRI

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
- In active development
- 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 script
- 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

/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

Ruby First Glance

Straight in the deep end!

Hello World

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

Hello.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 class method that is called 'say'
- string interpolation, variable substitution

Some Comparisons?

Ruby

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

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

@interface Magazine : Book <Glossy, SoftCover> {
  NSString * title;
  NSString * content;
}

@property (nonatomic, assign) NSString * title;
@property (nonatomic, assign) 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]) {
      title = aTitle;
      content = someContent;
    }
    return self;
}

+ (NSString *)description {
  return [NSString stringWithFormat:@"%s: %s", title, 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

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

JavaScript

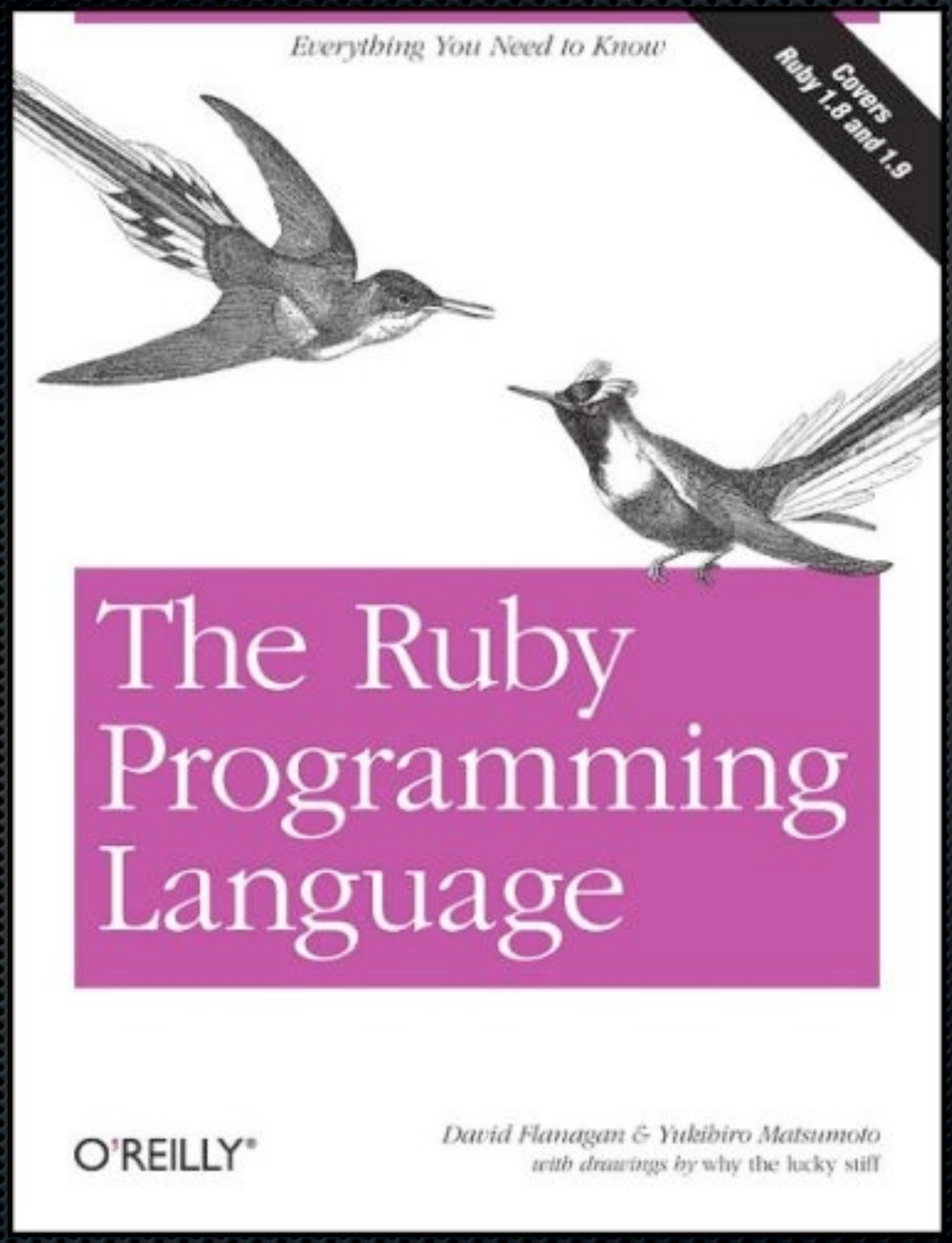
Ask Justin/Gareth

Ruby The Language

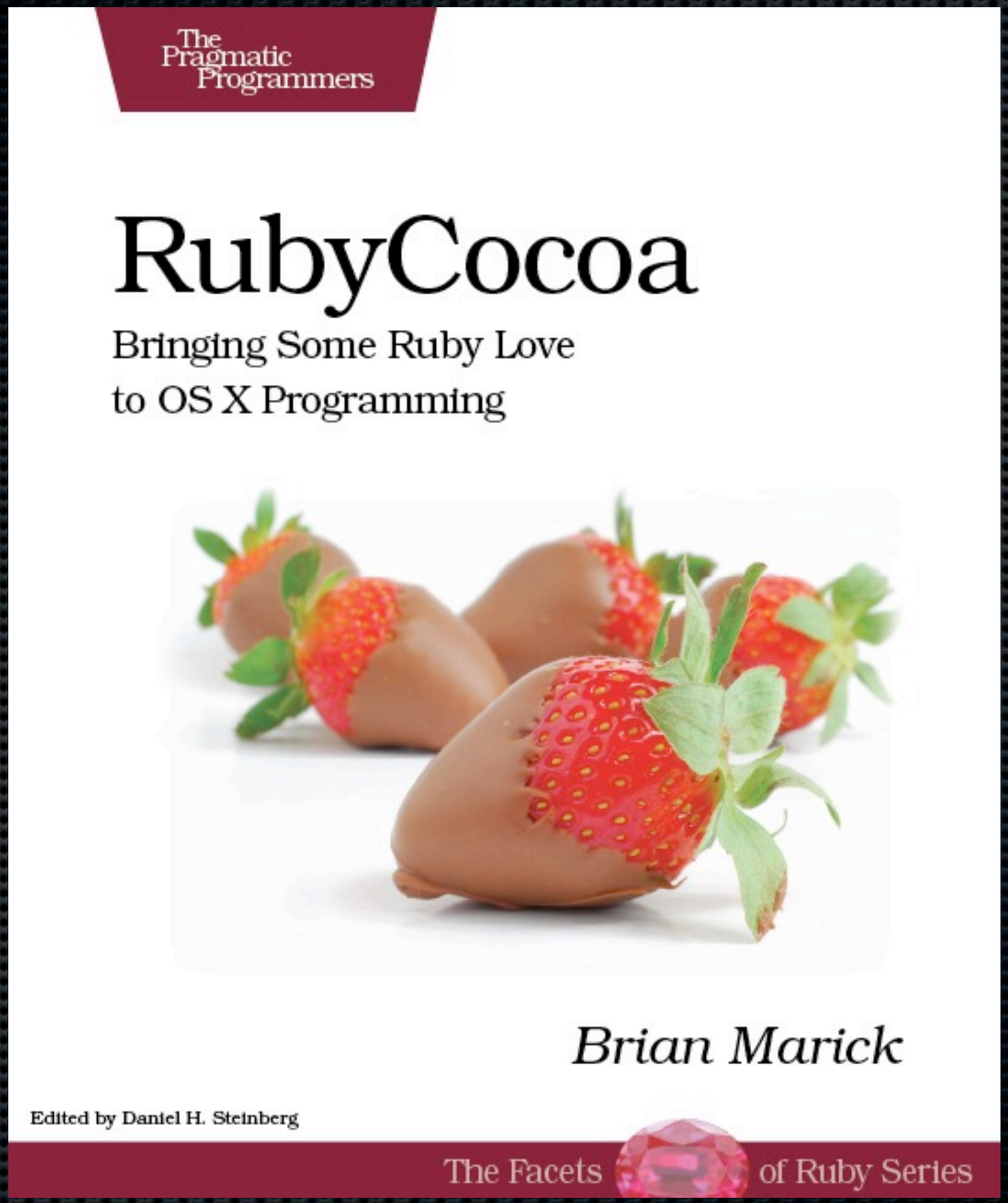
The Details!

Reference Material

We've been using the following books for reference material

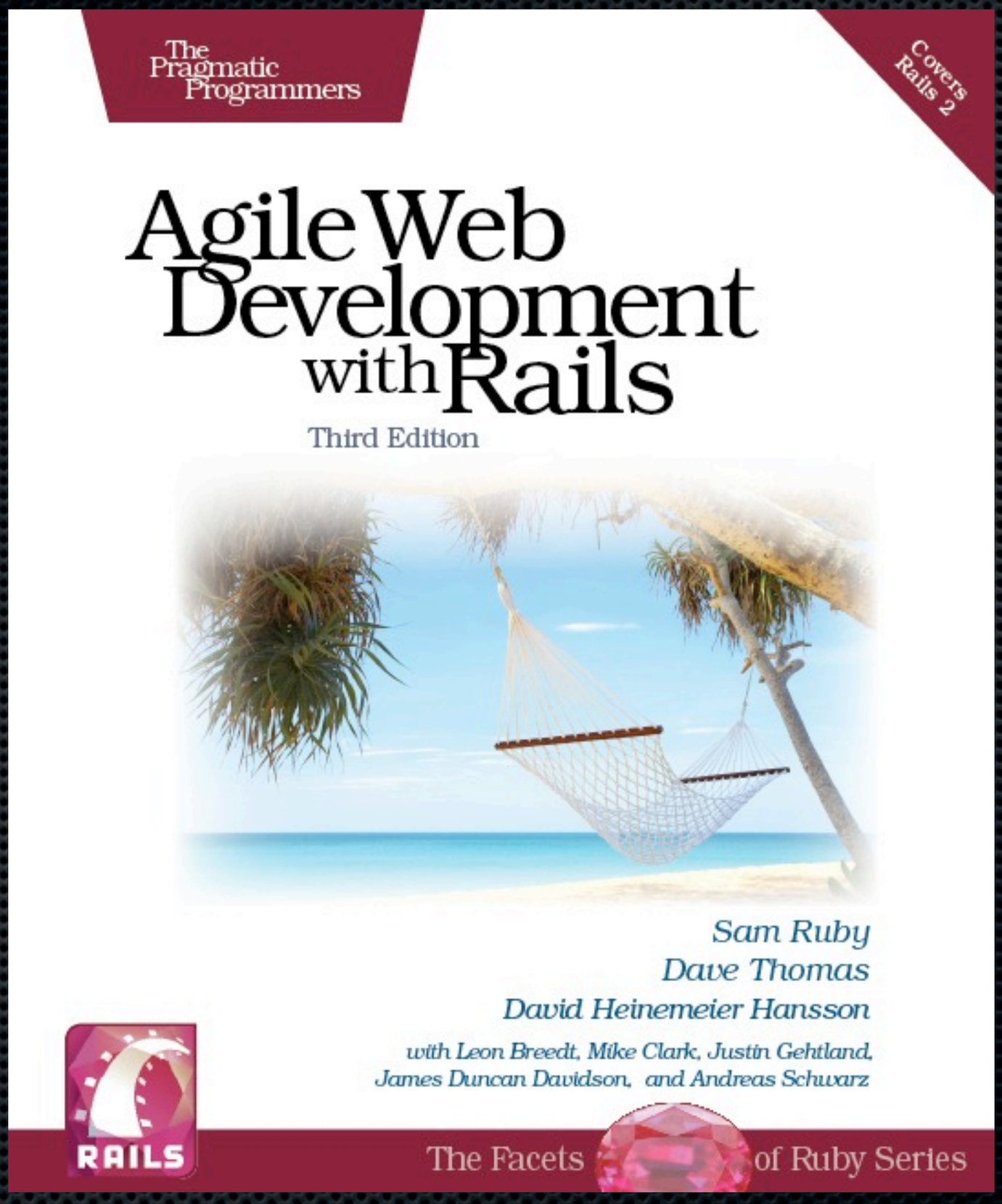


This is the book we're using for the course
- written by the creator of Ruby, Matz and David Flanagan
- the most definitive resource for the ruby language
- up to date, covers Ruby 1.8 and 1.9, the language, environment and standard library
- does not cover applications, other libraries like RubyCocoa, MacRuby, etc
- this one has been bought for everyone, will be available for the course or posted afterwards



Monday, 9 March 2009

Pragmatic programmer RubyCocoa book
- Still a beta book, available as a PDF, but in print soon
- Gareth will cover more



Monday, 9 March 2009

Rails development

- the definitive source of how to use and develop Rails
- MVC, web, databases, javascript, etc

Data Types

Numbers, Text, Literals, Collections, etc

Numbers

- Represented by Objects

```
10.times do |i|
  print "#{i}"
end # => 0123456789

5.to_s # => String “5”

3.send :+, 5 # => 3 + 5 = 8
```

- Numeric, base class for all numbers

Monday, 9 March 2009

Everything is an object in Ruby (repeat again and again :))

- Even numbers are objects
- They represent their numeric values
    - They have methods (can even use .send)
    - can be opened up and extended (do a live example)
- Numeric
    - base class for all numbers

Integers

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

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

- Bignum, if larger

```
1234567890.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

Monday, 9 March 2009

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

Text

- Represented by the String class
- Mutable sequences of characters
  - Ruby 1.9 adds comprehensive multi-byte support
- Quoting
  - Single, double, %q/Q and HERE documents

Single quoted Strings

'the quick brown fox jumps over the lazy dog'

- Not escaped
- No variable interpolation
- No special characters such as \

Double quoted Strings

"the #{speed} brown #{animal} jumps over the #{mood} dog\n"

- Escape sequences
- Variable interpolation

Variable interpolation is really useful
- other languages like java only support + which means the interpreter has to join all of the strings together, before the method call is invoked, which can be expensive
  - debug logging calls are an example

logger.debug("started run, name = " + name + ", and time is: " + current_time);

%q/Q quoted Strings

%q(the ‘quick’ brown fix jumps over the ‘lazy’ dog)

%Q(the “quick” #{colour} fix jumps over the ‘lazy’ dog)

- %q equivalent to a single quoted string, including ‘
- %Q equivalent to a double quoted string, including “

%q/Q are equivalent to single/double quoted strings but let you use the ‘ or “ character inside them without needing to have back tick \ characters or tooth pick syndrome - when I was coding in Java or C this was a big problem, especially with regular expressions

HERE documents

```
document = << EOF
  some example text for the Ruby course.
  Several lines of “text” and #{variables}
EOF
```

- Multiline strings with ease
- Includes variable interpolation and ‘/” characters

%q/Q are equivalent to single/double quoted strings but let you use the ‘ or “ character inside them without needing to have back tick \ characters or tooth pick syndrome

Usual Suspects

- Common string operators and methods are supported
- Examples

```
'123' * 3     # => 123123123
'MiXed'.upcase! # => MIXED
'MiXed'[0]    # => M
'a ' << 'test' # => a test
```

- Ruby API has all the details

Check Ruby docs for more details

- string manipulation
- concatentating, splitting, regular expressions
- modification, inline and returning a changed value
- accessing characters

- often open up the String class to do your own methods rather than write a Utils class as in Java
    - implement an example, pluralize? humanize, titleize? from active support?

Arrays

- Sequential position indexed data structure
- []

```
array = [1, 2, 3, 4]
```

- Array.new

```
array = Array.new
4.times { |i| array << i + 1 }
```

- %w()

```
array = %w( this is a new array )
# => [ ‘this’, ‘is’, ‘a’, ‘new’, ‘array’ ]
```

Monday, 9 March 2009

- Literal [] creates an empty array
- Array.new creates a new one using the class name
    - our example here we’re using a block to inject 4 numbers into the array using the << operator
- Last method is really useful if you want to have an array of strings
    - dont have to worry about quoting and all of the ‘, ‘ etc..

Usual Suspects

- Common array operations supported

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

a[-1]   # => 5
a[2, 3] # => [2, 3]

a.each_with_index do |o, i|
  puts "object at position #{i} is #{o}"
end
```


Check Ruby docs for more details

- enumeration, usually via blocks, we'll get to that construct soon in Enumerable
- accessing items
- pushing, popping items in out of the array
- nil (ie. a non value) is allowed inside an array
-

Usual Suspects

- Set operations

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

a | b # => [ 1, 2, 3, 4, 5 ] # unique values
a & b # => [ 2, 3, 4 ] # intersection
```

Monday, 9 March 2009

Some more interesting operations include set operations

- unique values

- intersection

- Nothing existed like this when for Java, had to use Apache Commons Collections

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 |k, v|
  puts "My #{k} name is #{v}"
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.

Symbols

- Common object used in Ruby
- References a constant string inside the Ruby symbol table
- Reduces complex String into a cheap integer operation
  - Fast lookup and comparison

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

Monday, 9 March 2009

Common object
- interpreters often use them to reference method names, variables, etc as they're indexed inside a symbol table
- ruby allows you to define any number of symbols, and their access/comparison is a constant time integer lookup rather than complex string manipulation
- similar to an immutable string object at the class level, Symbol defines many common string operations such as =~, [], etc..

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 'false'
  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 :)

Monday, 9 March 2009

For a light break :)

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

Objects

- Everything we've covered so far has been an object
- No fundamental types like in C, C++, Java, etc
- All work with objects are done via references
- Arguments as always passed by reference, not value
  - Methods can modify objects outside their scope

References

```ruby
# creates a new string called Ruby, referenced by s
s = "Ruby"

# creates a new reference to s called t, same object
t = s

# modifies the object, affects both t and s
s << ' is cool'

# t now points to a new object
t = 'Rails'

# => Ruby is cool on Rails
puts "#{s} on #{t}"
```

Types and Reflection

- Each object has a type
  - Object is the root of all class hierarchies
  - class, superclass
- No casting required for references, dynamic typing
- Object: instance_of? is_a?, kind_of? and ===
- Class: ancestors

Monday, 9 March 2009

We'll get into more details tomorrow when we get into meta programming
- each object has a class type, in the hierarchy
- remember ruby is dynamically typed so there's no casting required for references
- you can query an object for its class, its superclass, and its ancestors
- ancestors (modules and classes included into the class)
    - demo with String "string".class.ancestors

Examples

```
dog = Dog.new('Ben')

dog.class           # => Dog
dog.class.superclass # => Object
dog.class.ancestors  # => [Dog, Object, Kernel]

Dog === dog          # => true
dog.is_a? Dog        # => true

dog.kind_of? Cat     # => false
```

Assignment

```
# Single

a = "cassie"

# Parallel

dog, cat = "Cassie", "Alf"
dog, cat = cat, dog

# Abbreviated

pets ||= %w( Dog Cat Rabbit )
```

Splat (*) operator

```
# Splat (RHS)

names = %w( Alf Banjo )
dog, cat, rabbit = 'Ben', *names
  # => dog = 'Ben'; cat = 'Alf'; rabbit = 'Banjo'

# Splat (LHS)

cat, *dogs = 'Alf', 'Ben', 'Cassie'
  # => cat = 'Alf', dogs = [ 'Ben', 'Cassie' ]
```


Splat operator

- Collect and spread values into/from arrays
- Useful in methods for variable length parameter definitions

Operator & Precedence

- Similar to other C/Java/etc langauges
- Page 102 in The Ruby Language

Wont spend much time here
- operator precedence is often what you expect
- similar to C, Java, Objective-C, etc.
- its in page 102 in the book if you have it/when you get it

Statements

if, unless, while, until, for, loop

the if statement

```
if expression
  code
end

if expression
  code
elsif expression # note the missing ‘e’
  code
else
  code
end

code if expression

# Example

puts "We have some Ruby programmers!" if rubyists.size > 0
```


Parenthesis are optional, if the expression can be evaluated without it
- single line version makes things very readable
- elsif has a missing ‘e’, no one really knows why :)

the unless statement

```
unless condition
  code
end

unless condition
  code
else    # note no elsif
  code
end

code unless condition

# Example

puts "We have some non-Ruby programmers!" unless rubyists.size > 0
```


Unless, the opposite of ‘if’
- again more readability, don’t need to use the ! operator
- note there’s no elsif with an unless, only ‘unless’ and ‘else’

the case statement

```
case object
when conditions
when conditions
when conditions
else
end
```

```
case language
when 'Ruby'
when 'Perl'
when 'Objective-C'
when 'Smalltalk'
else
end
```

Monday, 9 March 2009

Case statement,

- allows you to bunch up a set of conditional testing
- Case statements don't fall through, i.e. no need for a break
- Can put multiple conditions on one line

the case statement

```
case object
when String
when Numeric
when TrueClass, FalseClass
else "other"
end
```

Both case and if return a value to its caller

Even more powerful than C/Java based switch statements
- tests with === (the case equality) operator
  - fixnum, its the same as ==
  - Class, defines it so that the RHS is the same class as the object in question
- both case and if return a value to the caller for assignment
  - x = if blah; self; else; nil;

the while statement

```
while condition_is_true
  code
end

# Example
while summertime?
  party!
end

puts x = x + 1 while x < 10
```


while statement, similar to C/Java, etc.

- can be used as a single line modifier
- notice boolean methods by convention have a '?' on the end of them
- enhances readability

the until statement

```
until condition_is_true do
  code
end

# Example

until iphone_stock.empty? do
  iphone_stock.sell!
end

puts stack.pop until stack.empty?
```


unless statement, opposite of ‘while’
- can be used as a single line modifier
- notice boolean methods by convention have a ‘?’ on the end of them
- enhances readability

the for loop

```
for reference in collection do
  code
end

# Example

array = %w( an array of strings )
for word in array do
  puts word
end

# Also equivalent

array.each do |word|
  puts word
end
```


The for loop
- references a member of a collection, evaluates the block for each value
- not quite used as much as you'd expect due to block support in ruby
- array.each is seen more due to Enumerable, which we'll get to, adds support for many styles of enumeration, iteration over Ruby collections

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
- Identified by a do...end or { .. } at the end of a method
- Can accept variables, and return a value
- Accessible by the developer using yield or block.call

Examples

```ruby
1.upto(10) do |i|
  puts i
end

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

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

Monday, 9 March 2009

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.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.

Closures

- Blocks are also closures
- Can reference variables outside the scope of the block, even after the those variables original scope has disappeared

Example

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

parts = SpareParts.new

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

http://flickr.com/photos/slieschke/48590998/

Monday, 9 March 2009

Seeing we’re talking about Ferraris so much

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

- 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

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

pets   = [ cassie, alf, ben ]

pets.reject { |pet| pet.is_a? Cat }

# => [ cassie, ben ]
```


Here we're rejecting all types that are a Cat

Collect

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

pets   = [ cassie, alf, ben ]

pets.collect { |pet| pet.breed }

# => [‘cocker spaniel’, ‘house cat’, ‘spaniel’]
```


Here we’re collecting the results of calling the method ‘breed’ on all pets in the array, returning a new array with the results

Monday, 9 March 2009

Inject is very powerful and magical :)

Inject

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

pets   = [ cassie, alf, ben ]
```

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

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


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.

Equivalent Iterator Style

```
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;

Map results = new HashMap();

for (Iterator i = pets.iterator(); i.hasNext; ) {
  Pet p = (Pet) i.next();
  results.put(p.breed(), p);
}
```


Inject, collect, etc are all very powerful.

Here's a example of typical Java code that achieves the same results as the 1 line previous.

Far less code, easier on the developer and more readable.

Exceptions

Raise & Rescue

Exception Handling

- Similar to other modern languages
  - raise (throw)
  - rescue (catch)

Exception handler, similar to other languages
- raise is equivalent to throw in Java, C++, etc
- rescue is equivalent to catch in Java, C++, etc

You can catch specific Exceptions, etc. Show an example if requested

Methods

Definitions, Visibility, Constructors, etc

Methods

- Defined with the keyword def, marked by the word end
- Methods return the value of the last statement
  - Unless terminated elsewhere via keyword return or an exception via raise
    - May return multiple values for parallel assignment
    - Reduces number of temporary variables
- self is the default receiver for methods
- Object.send is the same as a method invocation

Number of temporaries are reduced to hold transient values
‘return’ is required to return multiple values

self, is the current object, and its the default receiver, even when you’re writing scripts, self is the main object in the script and the default receiver, there are no global methods.

Example

```
def purchase(amount, method)
  raise 'No amount specified' unless amount or amount < 0

  case method
  when CreditCard
    online_purchase(amount)
  when Cash
    cash_purchase(amount)
  else
    raise 'Unknown payment method'
  end
end
```


Example, accepts 2 parameters
- note, no return type specified, dynamic type
- case statement is the last method
  - doesn't look like it but it is
  - its return value is the return value of the method
- 2 raise statements terminate (ie. return from) the method prematurely

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

Arguments

- Parenthesis are optional for declaration and invocation
- Arguments may have default values
- Methods may have variable length arguments
- Hashes provide named arguments
- Methods may accept one block via do..end or { }

- In general in Ruby, parenthesis are only required to clear ambiguity
- Default values allow you to consolidate methods (eg. constructors)
- variable length arguments via splat operator
- Hashes approximate named args
- Block syntax as discussed

Example

```
def purchase(amount, method)
  # ...
end

car.purchase 100_000, amex_card

def purchase(amount, method = Cash)
  # ...
end

car.purchase 100_000

def purchase(amount = 0, method = Cash)
  # ...
end

car.purchase
```


purchase method, shows default values of methods

1. amount and method must be specified
   - notice 100_000, Ruby allows _ to break up large numbers
   - notice parenthesis not required - perl programmers will feel at home
2. default method of cash
   - thats a nice big cash payment
3. default amount (free) and method (cash)
   - no parameters nor parenthesis required at all

```
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

Monday, 9 March 2009

Default arguments help a lot with constructors
- here some example Java code i used to have to write all the time
- 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

Variable Example

```
def method_missing(sym, *args, &block)
  super unless @proxy.respond_to? sym
  @proxy.send sym, *args, &block
end
```

- Special method, more about method_missing soon
- Variable arguments via the splat operator
  - All arguments are wrapped in an array args
  - Arguments splatted back into @proxy
- .send provides dynamic method invocation

Named arguments

```
def purchase(options = {})
  amount = options[:amount]
  method = options[:method]
  # ...
end

car.purchase :amount => 100_000, :method => Card
```

- Hash braces are optional { }, along with parenthesis

Blocks

```
def purchase(amount, method)
  # ...
end

car.purchase 100_000, Card do |token, receipt|
  if token.pin_challenge_required?
    # ...
  end
end
```

- Single block is optional
- Functionality dependent on method being called

Classes

Definitions, Visibility, Constructors, etc

Classes

- Defined with the keyword class, marked by end
- Single inheritance only via the < construct
  - Multiple module inclusion via include statement
  - Object is the default and root superclass
- Constructor always called initialize, 0..n arguments
- Methods public by default

We've seen a few examples along the way, lets solidify some of the rules surrounding classes

Classes & Methods

- Class methods start with self.method_name
- Operators can be overloaded, eg. +, [], /, *, etc
- Existing classes can be extended with new methods
- Dynamic method table
  - Allows catching and responding to missing methods
- Duck typing, no casting or type matching required
  - If it responds to a method, thats acceptable

Adding methods

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

"Ruby!".parenthesize # => “(Ruby!)”
```


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

Class Variables

- Instance variables start with a @variable_name
- Class variables start with @@variable_name
- Accessors/Getters can be dynamically defined
  - attr_accessor, attr_reader and attr_writer
- Instance variables are always private
- Constants are always public

Define an accessor method to make an instance variable public

Duck Typing #1

```
class Dog
  attr_accessor :name, :breed
  def initialize(name, breed)
    @name, @breed = name, breed
  end
end
```

```
class Cat
  attr_accessor :name, :breed
  def initialize(name, breed)
    @name, @breed = name, breed
  end
end
```


We'll define two classes, very similar to represent dogs and cats

Duck Typing #2

```
class Pets
  attr_accessor :pets

  def initialize(*pets)
    @pets = pets
  end

  def to_s
    pets = @pets.collect do |pet|
      "#{pet.breed}: #{pet.name}"
    end
    pets.join(', ')
  end
end
```


Define a further class representing our pets at home

Duck Typing #3

```
cassie = Dog.new('Cassie', 'Cocker Spaniel')
alf    = Cat.new('Alf', 'House Cat')
ben    = Dog.new('Ben', 'Spaniel')

pets   = Pets.new cassie, alf, ben

puts pets # => Cocker Spaniel: Cassie, House Cat: Alf,
Spaniel: Ben
```


Now we'll create a pets, and add them to the pet container class

Printing it via puts calls .to_s automatically, and you'll see that it prints the breeds and names of each pet in a string

- Dog and Cat don't share a common parent class, other than Object,
- both Dog and Cat respond to :breed and :name, so Ruby calls them, there's no need to have them inherit from a common abstract class for correct typeness - duck typing does it all
- might be good to inherit from an animal base class for code re-use - more of a design question

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

Class Reuse & Extension

- Inheritance
- Composition
- Traits, module inclusion

Inheritance, single hierarchy only

- seen examples of this already

Composition, include references or instantiate smaller components to make a larger one

Modules, include code from modules into any number of classes

```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
```


Example we've seen which includes all 3

- composition for titles, content
- inheritance for type, which is book
- modules included for glossy and soft cover print

```
module Glossy

  def print
    # ...
  end

end

module SoftCover

  def create
    # ...
  end

end
```

```
class Magazine
  include SoftCover, Glossy
end

class Book
  include HardCover, Glossy
end

class Manual
  include SoftCover
end
```


Some example modules Glossy and SoftCover
- included into concrete classes as required
- Magazine, Book, Manual

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.txt is in /Library/Ruby/Gems/1.8/gems/twitter-0.3.7 describing the API
- 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

Further enhancements

- Support for reading a particular users tweets
- Support for posting new tweets from the command line
  - Requires valid credentials/account

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::Base.new('', '').timeline(:public)
    end

end

Tweeter.new.public_tweets
```

Twitter 2.0

RubyCocoa Desktop Client

RubyCocoa Twitter

- Create a window with a NSTableView in IB
- Create an OSX::NSObject class derivative
- Implement NSTableView delegate methods