A world outside of Java
Before we get started
Before we get started

- A few words on testing
Before we get started

- A few words on testing
- Let’s look at some stats from project 1
Testing: coverage
Testing: passage

- 60% All Passed
- 40% Failures
Testing: automation

- Automation of Test Suite: 90%
- Singleton Test Cases: 10%
Testing: thoughts?
Testing: goals
Testing: goals

- One-click automatic testing of all major functionality
Testing: goals

- One-click automatic testing of all major functionality
- Ideally, 100% coverage of all code
Testing: goals

- One-click automatic testing of all major functionality
- Ideally, 100% coverage of all code
  - Practically, 70% coverage of statement, loop, and conditions
Testing: goals

- One-click automatic testing of all major functionality
- Ideally, 100% coverage of all code
  - Practically, 70% coverage of statement, loop, and conditions
- Meaningful tests
Testing: suggestions
Testing: suggestions

- Write tests alongside code
- Even better, have your partner write them
Testing: suggestions

- Write tests alongside code
  - Even better, have your partner write them
- Think of corner cases
  - Verifying basic functionality isn’t enough!
Testing: suggestions

- Write tests alongside code
  - Even better, have your partner write them
- Think of corner cases
  - Verifying basic functionality isn’t enough!
- Use a code coverage tool
  - Ask a grader
Testing: requirements
Testing: requirements

- All tests must pass
Testing: requirements

- All tests must pass
- All major functionality must be covered
Testing: requirements

- All tests must pass
- All major functionality must be covered
- Create a suite to run all tests
  - One class, preferably called `AllTests`
- Makes it easier for you to prevent regression
Testing: JUnit Suites

- JUnit 4
- JUnit 3
Testing: JUnit Suites

- JUnit 4

```java
import org.junit.*;
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(value=Suite.class)
@Suite.SuiteClasses(value=
    UnitTest1.class,
    UnitTest2.class,
    IntegrationTest1.class,
    ...
)
public class AllTests {
}
```

- JUnit 3
Testing: JUnit Suites

JUnit 4

```java
import org.junit.*;
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(value=Suite.class)
@Suite.SuiteClasses(value=
    UnitTest1.class,
    UnitTest2.class,
    IntegrationTest1.class,
    ...
})
public class AllTests {
}
```

JUnit 3

```java
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {
    public static Test suite() {
        TestSuite suite =
            new TestSuite("Cool test suite");
        suite.addTest(UnitTest1.suite());
        suite.addTest(UnitTest2.suite());
        ...
        return suite;
    }
}
```
Testing: JUnit Suites

- JUnit 4

```java
import org.junit.*;
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(value=Suite.class)
@Suite.SuiteClasses(value=
    UnitTest1.class,
    UnitTest2.class,
    IntegrationTest1.class,
    ...)
public class AllTests {
}
```

- JUnit 3

```java
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {
    public static Test suite() {
        TestSuite suite =
            new TestSuite("Cool test suite");
        suite.addTest(UnitTest1.suite());
        suite.addTest(UnitTest2.suite());
        ... return suite;
    }
}
```
Questions on testing?
The business
An Alternative

The world outside of Java
An Alternative

The world outside of Java
An Alternative
The world outside of Java
An alternative

LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.

I WONDER IF THE CYCLES WILL CONTINUE FOREVER.

A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.

THESE ARE YOUR FATHER’S PARENTHESES

ELEGANT WEAPONS

FOR A MORE... CIVILIZED AGE.
An alternative

You guys have heard enough about Lisp
An alternative

You guys have heard enough about Lisp
The real deal
The real deal

- I get an hour to warp your minds however I wish
The real deal

- I get an hour to warp your minds however I wish
- It’s time to spread the propaganda joy
The real deal

- I get an hour to warp your minds however I wish
- It’s time to spread the propaganda joy
- One of my guilty pleasures...
The real deal

- I get an hour to warp your minds however I wish
- It’s time to spread the propaganda joy
- One of my guilty pleasures...
What is Ruby?

- An object-oriented scripting language
- A more different Perl
  - Just as powerful
  - Simpler, better OO
- Principle of “least surprise” -- What you expect is what you get
Typical uses of Ruby

- System scripts (superb regex support)
- Web programming (CGI, Rails)
  - Also agents and crawlers
- DB programming (DBI, ActiveRecord)
- GUI (Tk, RubyMagick)
- Scientific computing (GSL, RSRuby)
Ruby philosophy according to Matz

- **Lightweight Languages** are not lightweight in the sense of ease of implementation, but they are called lightweight because of their intention to lighten the workload of the programmer.
Ruby philosophy according to Matz

- **Brevity** is one element that makes code beautiful. As Paul Graham says, “Succinctness is power.” In the vocabulary of programming, brevity is a virtue.

- Brevity can also mean the elimination of redundancy.

- In order to eliminate redundancy, we follow the **DRY principle**: Don’t Repeat Yourself. The concept of DRY is the antithesis of copy-and-paste coding.
Ruby philosophy according to Matz

- **Simplicity** is the next element of beautiful code. We often feel beauty in simple code.
- When simpler tools are used to solve a complex problem, complexity is merely shifted to the programmer.
Ruby philosophy according to Matz

- **Balance** is the final element of beautiful code. I advocate brevity, conservatism, simplicity, and flexibility. No element by itself will ensure a beautiful program.
Ruby philosophy

- Enable programmers to write beautiful code
- Make their lives easier
- You should *enjoy* coding, not waste time hacking
General Features

- High level language
- True OO -- EVERYTHING is an object
- Interpreted (for the most part)
- Easy to learn
Ruby as a high level language
Ruby the high level language
Ruby the high level language

- Interpreted with garbage collected runtime
Ruby the high level language

- Interpreted with garbage collected runtime
- Extensive libraries
Ruby the high level language

- Interpreted with garbage collected runtime
- Extensive libraries
- Very easy to accomplish common tasks and computations
Ruby the high level language

- Interpreted with garbage collected runtime
- Extensive libraries
- Very easy to accomplish common tasks and computations
- Great regexp support
Ruby the high level language

- Interpreted with garbage collected runtime
- Extensive libraries
- Very easy to accomplish common tasks and computations
- Great regexp support
- Easy to pick up from other languages
Ruby the high level language

- Simple, straightforward syntax
Ruby the high level language

- Simple, straightforward syntax
Ruby the high level language

- Simple, straightforward syntax
Ruby the high level language

- Simple, straightforward syntax

```perl
perl -le 'chomp(@words = sort {length($b) <=> length($a)}
grep {/^[a-z-]\{2,\}\s*$/} `cat wordlist`);
push @words, "a", "i";$source=shift @ARGV;
$let{lc $1}++ while $source =~ /([a-z])/ig;
@owords{map {lc $_} ($source=~/([a-z]+)/ig)}=();
$p=join ",", map {"$ *"} sort keys $let;@words = grep {!exists $owords{$_}}
grep {join("",sort split //, $_) =~ /$p/} @words;
sub next_word {
    my $wi=shift;my $l = shift @_; W:for(my $i=$wi;$i<@words;$i++){
        my $word=$words[$i];my %t = %$l;
        foreach my $ll (grep {/[a-z]/} split //, $word) { next W if --$t{$ll} < 0 }
        if (grep {$_ > 0} values %t){next_word($i,%t,@_,$word)}
        else{print join ",", @_, $word}};next_word(0,%let)' "phrase"
```
Ruby the high level language

- Simple, straightforward syntax
Ruby the high level language

- Simple, straightforward syntax

```ruby
def anagrams(chars, result='', &b)
  if chars.empty? then
    yield result
  else
    chars.length.times do |i|
      c = (x = chars.clone).slice!(i)
      anagrams(x, result + c, &b)
    end
  end
end

words = []
anagrams("phrase"){|anagram| words << anagram}
```
Ruby the high level language

- Iteration of enumerable objects is really easy
Ruby the high level language

- Iteration of enumerable objects is really easy

```ruby
# Declare an array with some members
xs = [1, 2, 'a', 5, 17, :mother, nil]

# print a verbose representation to stdout
xs.each{|x| puts x.inspect}

# same thing in block form
xs.each do |x|
  puts x
end
```
Ruby the high level language

- Mutation of enumerable objects is just as simple
Ruby the high level language

- Mutation of enumerable objects is just as simple

```ruby
# Declare an array with some numbers
xs = [1, 2, 94, 4, 17, 23, 5]

ys = xs.sort.map{|x| x + 1}
# ys --> [2, 3, 5, 6, 18, 24, 95]
```
Object orientation in Ruby
Ruby the OO language
Ruby the OO language

- Starts with the same general features of Java
Ruby the OO language

- Starts with the same general features of Java
  - Base class Object
Ruby the OO language

- Starts with the same general features of Java
  - Base class Object
  - Public, private, and protected visibility
Ruby the OO language

- Starts with the same general features of Java
  - Base class Object
  - Public, private, and protected visibility
  - No multiple inheritance
Ruby the OO language

- Starts with the same general features of Java
  - Base class Object
  - Public, private, and protected visibility
  - No multiple inheritance
  - Package namespaces
Ruby the OO language

- Starts with the same general features of Java
  - Base class Object
  - Public, private, and protected visibility
  - No multiple inheritance
  - Package namespaces
- Similarity ends there
Ruby the OO language
Ruby the OO language

- No [anonymous] inner classes
- “Poor man’s closures”
Ruby the OO language

- No [anonymous] inner classes
- “Poor man’s closures”
- Ruby has real closures
Ruby the OO language

- No [anonymous] inner classes
- “Poor man’s closures”
- Ruby has real closures

```ruby
class Adder
  attr_reader :add_n
  attr_accessor :n

  def initialize(n)
    @n = n
    @add_n = lambda{|x| x + n}
  end
end

adder = Adder.new(5)
add_closure = adder.add_n
add_closure.call(3) # returns 3 + 5
adder.n = 8
add_closure.call(3) # returns 3 + 8
```
Ruby the OO language
Ruby the OO language

- Duck typing, not identity by inheritance
Ruby the OO language

- Duck typing, not identity by inheritance
- “If it walks like a duck and quacks like a duck, I would call it a duck.”
Ruby the OO language

- Duck typing, not identity by inheritance
- “If it walks like a duck and quacks like a duck, I would call it a duck.”
- An object’s current set of methods and properties determines the valid semantics
Ruby the OO language

- Duck typing, not identity by inheritance
- “If it walks like a duck and quacks like a duck, I would call it a duck.”
- An object’s current set of methods and properties determines the valid semantics
- Similar to OCaml structural typing, but type compatibility is determined only by the part of a type’s structure that is accessed
Ruby: duck typing

- Manifestation of Duck Typing
  - No abstract classes
    - (Though modules can define classless groups of methods)
  - No interfaces
    - `instanceof` is no longer an appropriate means of introspection
Ruby: duck typing

- We care about object behavior more than inheritance
Ruby: duck typing

- We care about object behavior more than inheritance

```ruby
class Fee
  attr_reader :date

  def date=(val)
    if val.class != Date then
      raise ArgumentError.new("Not a date")
    end
  end
end
```
Ruby: duck typing

- We care about object behavior more than inheritance

```ruby
class Fee
  attr_reader :date

  def date=(val)
    if val.class != Date then
      raise ArgumentError.new("Not a date")
    end
  end
end
```
Ruby: duck typing

- We care about object behavior more than inheritance

```ruby
def date=(val)
  case val
  when Date
    @date = val
  when Time
    @date = Date.new(val.year, val.month, val.day)
  when String
    if val =~ /\d{4}\s\[\-\//\]\s\d{1,2}\s\[\-\//\]\s\d{1,2}/
      @date = Date.new($1.to_i,$2.to_i,$3.to_i)
    else
      raise ArgumentError, "Unable to parse #{val} as date"
    end
  when Array
    if val.length == 3
      @date = Date.new(val[0], val[1], val[2])
    else
      raise ArgumentError, "Unable to parse #{val} as date"
    end
  end
end
```
Ruby: duck typing

- We care about object behavior more than inheritance

```ruby
def date=(val)
  case val
  when Date
    @date = val
  when Time
    @date = Date.new(val.year, val.month, val.day)
  when String
    if val =~ /\d{4}\s*[-/\\]\s*\d{1,2}\s*[-/\\]\s*\d{1,2}/
      @date = Date.new($1.to_i, $2.to_i, $3.to_i)
    else
      raise ArgumentError, "Unable to parse #{val} as date"
    end
  when Array
    if val.length == 3
      @date = Date.new(val[0], val[1], val[2])
    end
    else
      raise ArgumentError, "Unable to parse #{val} as date"
  end
end
```
Ruby: duck typing

- We care about object behavior more than inheritance

```ruby
# Accepts an object which responds to the `+year+`, `+month+` and `+day+` methods.
def date=(val)
  [:year, :month, :day].each do |meth|
    raise ArgumentError unless val.responds_to?(meth)
  end
  @date = val
end
```
Ruby: duck typing

- We care about object behavior more than inheritance

```ruby
# Accepts an object which responds to the +year+, +month+ and +day+ methods.
def date=(val)
  [:year, :month, :day].each do |meth|
    raise ArgumentError unless val.responds_to?(meth)
  end
  @date = val
end
```

This is a pretty good solution
Ruby the OO language

- If this were Java, we’d have to work in many files to separate this new functionality from the original classes
  - Create `DateCastable` interface
  - Add hooks in target classes
  - Implement `DateCaster` class
- This can mean a lot of work in many files
Ruby: class extension

- The Ruby Way
  - Methods and fields can be added to classes at any time
  - Just create a new file with the required operations
  - One file to implement and another to test
Ruby: class extension

- "Visitor" example
- Just extend existing Fie, Foe and Fum classes

```ruby
class Fie # Note we don’t have to inherit from anything
def date
  ...
end
end

class Foe
def date
  ...
end
end

class Fun
def date
  ...
end
end
```
Ruby: class extension

- Other examples
  - Additions to NArray: mean, median, ranks...
  - Additions to Fixnum: factorial
  - Additions to ActiveRecord::Base::Base: find_or_create, for_name
Ruby: mixins

- Ruby also has a mechanism to implement generic code
- Separate such patterns into modules
- Import these methods into classes to create “Mixins”
Ruby: mixins

- Ruby also has a mechanism to implement generic code
- Separate such patterns into modules
- Import these methods into classes to create “Mixins”

Great way to stay DRY
Ruby: mixins
Ruby: mixins

- First, let’s look at modules
Ruby: mixins

- First, let’s look at modules
- They’re similar to Java abstract classes
  - Cannot be instantiated
  - Can fully implement methods
Ruby: mixins

- First, let’s look at modules
- They’re similar to Java abstract classes
  - Cannot be instantiated
  - Can fully implement methods
- One big difference
  - They’re included, not used as superclasses
Ruby: mixins
Ruby: mixins

# A Math module akin to Java Math class.
module Math
  def add(val_one, val_two)
    BigInteger.new(val_one + val_two)
  end
end
Ruby: mixins

# A Math module akin to Java Math class.
module Math
  def add(val_one, val_two)
    BigInteger.new(val_one + val_two)
  end
end

# Convert a integer value to English.
module Stringify
  # Requires an instance variable @value
  def stringify
    if @value == 1
      "One"
    elsif @value == 2
      "Two"
    elsif @value == 3
      "Three"
    end
  end
end
# A Math module akin to Java Math class.
module Math
  def add(val_one, val_two)
    BigInteger.new(val_one + val_two)
  end
end

# Base Number class
class Number
  def intValue
    @value
  end
end

# Convert a integer value to English.
module Stringify
  # Requires an instance variable @value
  def stringify
    if @value == 1
      "One"
    elsif @value == 2
      "Two"
    elsif @value == 3
      "Three"
    end
  end
end
# A Math module akin to Java Math class.
module Math
  def add(val_one, val_two)
    BigInteger.new(val_one + val_two)
  end
end

# Base Number class
class Number
  def intValue
    @value
  end
end

# BigInteger extends Number
class BigInteger < Number
  include Stringify
  extend Math
  def initialize(value)
    @value = value
  end
end

# Convert a integer value to English.
module Stringify
  require an instance variable @value
  def stringify
    if @value == 1
      "One"
    elsif @value == 2
      "Two"
    elsif @value == 3
      "Three"
    end
  end
end
Ruby: mixins
Ruby: mixins

# Create a new object
bigint1 = BigInteger.new(10)
# Call a method inherited from the base class
puts bigint1.intValue     # --> 10
Ruby: mixins

# Create a new object
bigint1 = BigInteger.new(10)
# Call a method inherited from the base class
puts bigint1.intValue   # --> 10

# Call class method extended from Math
bigint2 = BigInteger.add(-2, 4)
puts bigint2.intValue   # --> 2
Ruby: mixins

# Create a new object
bigint1 = BigInteger.new(10)
# Call a method inherited from the base class
puts bigint1.intValue    # --> 10

# Call class method extended from Math
bigint2 = BigInteger.add(-2, 4)
puts bigint2.intValue    # --> 2

# Call a method included from Stringify
puts bigint2.stringify   # --> 'Two'
Ruby: mixins

- But wait, there’s more!
- You can mixin modules with *instances*
Ruby: mixins

- But wait, there’s more!
- You can mixin modules with instances

```ruby
# Add the module methods to this object instance
bigint2.extend CurrencyFormatter
puts bigint2.format  # --> '$2'

puts bigint1.format  # will generate an error
```
Ruby the interpreted language
Ruby the interpreter

- Languages gain incredible flexibility with managed runtimes
  - Garbage collection
  - Dynamic class hierarchies
- Code as Data
Ruby the interpreter

- Ruby’s runtime is similar to many others
  - Interactive interpreter available
  - Hooks for debuggers
  - Slower than compiled
Ruby the interpreter

- Its runtime is probably Ruby’s weakest link
  - Mark and sweep GC
  - Green threads
  - Fairly slow
- Each of these is addressed in the development branch
A few more cool things
No Mutation...

- ...unless you really want it
No Mutation...

- ...unless you really want it

```ruby
xs = [1, 2, 3]
xss.slice(1)    --> 2
xs              --> [1, 2, 3]
xss.slice!(1)   --> 2
xs              --> [1, 3]
```
Web development

- Ruby on Rails
  - Very nice MVC architecture
  - Great templating engine with AJAX support
  - Persistent objects
  - Countless plugins
    - Authentication, common database tasks, more AJAX stuff, generic code (e.g. pagination)
Distributed Ruby

- Just like Java’s RMI without the pain
Distributed Ruby

- Just like Java’s RMI without the pain

```ruby
require 'drb'

class TestServer
  def doit
    "Hello, Distributed World"
  end
end

aServerObject = TestServer.new
DRb.start_service('druby://localhost:9000', aServerObject)
DRb.thread.join # Don't exit just yet!
```
Distributed Ruby

- Just like Java’s RMI without the pain

```
require 'drb'

class TestServer
  def doit
    "Hello, Distributed World"
  end
end

aServerObject = TestServer.new
DRb.start_service('druby://localhost:9000', aServerObject)
DRb.thread.join # Don't exit just yet!
```

```
require 'drb'

DRb.start_service()
obj = DRbObject.new(nil, 'druby://localhost:9000')
# Now use obj
puts obj.doit
```
OO design libraries

- Visitor pattern: traverse a collection without knowing its internal organization
- Delegates: flexible and dynamic class composition
- Singleton pattern: ensures only one instance of a class exists at a time
- Observer pattern: implements protocol allowing one object to notify a set of interested objects when certain changes have occurred
Access to other languages

- JRuby
- C API
- Bridges to specialty languages
  - MATLAB, R, etc.
So, that’s about it
Ruby is neat

- Simple, clear, concise syntax
- Advanced OO features
- Extensive libraries and bridges to other languages
Flames, abuse, and questions