ÉCOLE
POLYTECHNIQUE
DE LOUVAIN

# LSINF2335

# Programming Paradigms: Theory, Practice and Applications

Sebastián González

15 March 2016

# Lecture 6

# Introduction to Ruby

Partly inspired on :

- the pickaxe book “Programming Ruby” by Dave Thomas
- slides by Prof. Tom Mens (UMons)
- slides by Prof. Wolfgang De Meuter (VUB)
- an update to the slides made by Benoît Daloze (UCL)

# 6.1 History of Ruby

# What is Ruby?



Originally conceived by Yukihiro Matsumoto

started in 1993; first public release in 1995

February 2013: 2.0 release

Motivation:

"wanted a **scripting** language [...] more powerful than Perl, and more **object-oriented** than Python"

wanted a **natural** and very **expressive** language in which programmers can **easily** express themselves

Language named after a precious gemstone

à la "Perl"

Open source project

# Ruby is …



- Interpreted
- Object-oriented
  - ✓ everything is an object
  - ✓ every operation is a message sent to some object
- Dynamically typed
- Strongly reflective
  - ✓ add and modify code at runtime (metaprogramming)
  - ✓ ask objects about themselves (reflection)

# Ruby has …



- Compact syntax inspired by Python and Perl
- Semantics akin to dynamic class-based languages like Smalltalk
- Scripting facilities akin to those of Python and Perl
  - manipulation of text files
  - file handling
  - execution of system tasks
  - support for regular expressions
  - …

# Ruby's "killer app"




Ruby on Rails (a.k.a. "Rails" or "RoR")

- Web-application framework
- Implemented in Ruby
- Allows you to quickly create powerful web applications
- Based on the model-view-controller pattern
- Web-application =
  - Ruby program + database + webserver

# 6.2 Getting started with Ruby 

# Getting Started With Ruby

```
RUBY VERSION

→ ruby -v
ruby 2.3.0p0 (2015-12-25 revision 53290) [x86_64-darwin15]

→ irb -v
irb 0.9.6(09/06/30)

SIMPLE COMMAND PROMPT EXAMPLES

→ irb
>> 3+4
=> 7
>> "George"+"Orwell"
=> "GeorgeOrwell"

PRINTING

>> puts "Hello World"
Hello World
=> nil

>> 3.times { puts "Ruby" }
Ruby
Ruby
Ruby
=> 3
```

irb = Interactive Ruby

`puts` is a standard Ruby method that writes its argument(s) to the console, adding a newline after each

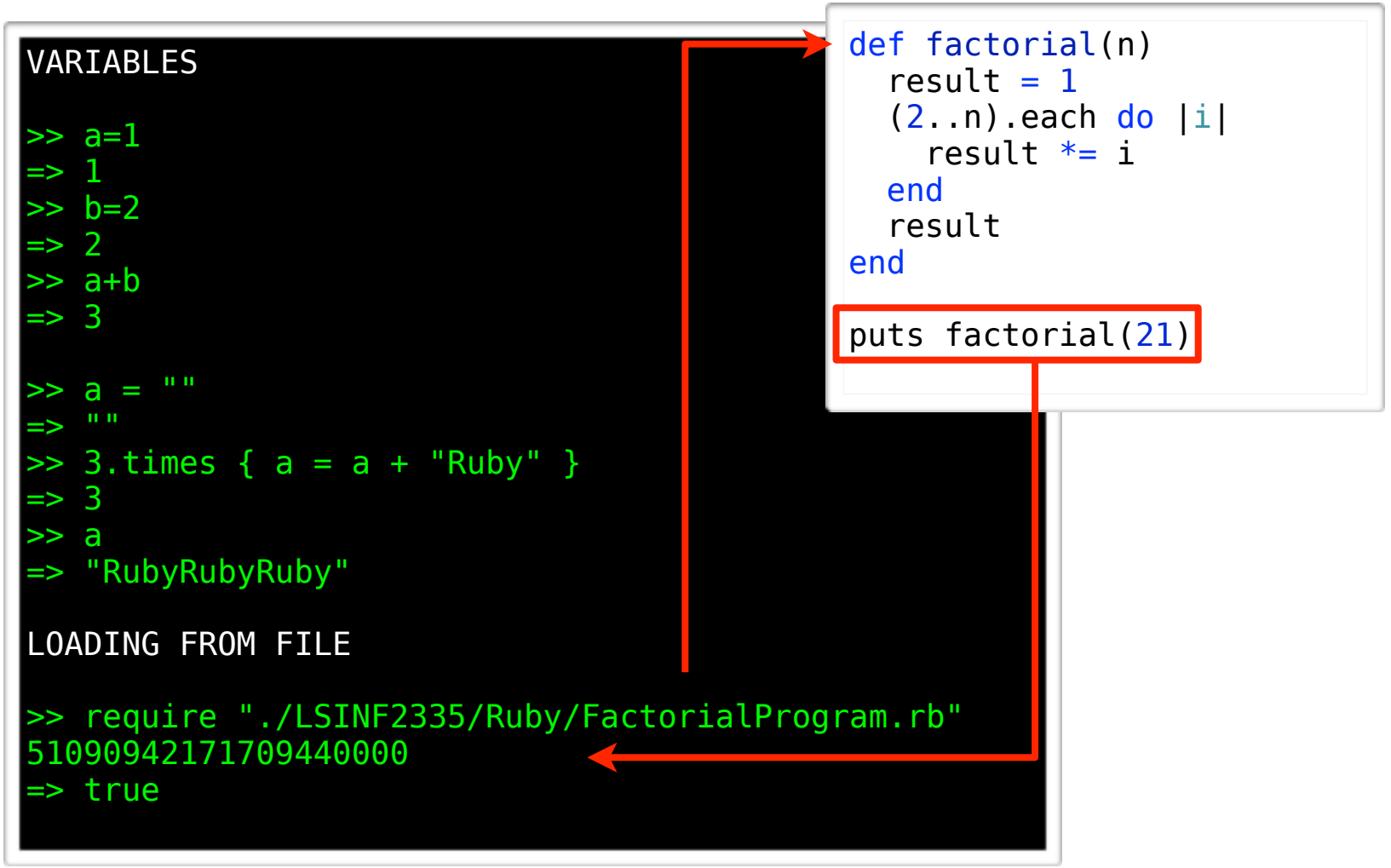
```
METHOD DEFINITIONS

>> def product(n1,n2)
>>   n1 * n2
>> end
=> nil

>> product(3,4)
=> 12

>> def fib(n)
>>   if n <= 1 then n
>>   else fib(n-1) + fib(n-2)
>>   end
>> end
=> nil

>> fib(1)
=> 1
>> fib(3)
=> 2
>> fib(5)
=> 5
>> fib(10)
=> 55
```


VARIABLES
>> a=1
=> 1
>> b=2
=> 2
>> a+b
=> 3
>> a = ""
=> ""
>> 3.times { a = a + "Ruby" }
=> 3
>> a
=> "RubyRubyRuby"
LOADING FROM FILE
>> require "./LSINF2335/Ruby/FactorialProgram.rb"
51090942171709440000
=> true
def factorial(n)
result = 1
(2..n).each do |i|
result *= i
end
result
end
puts factorial(21)

```
SCRIPTS

# Create .rb file
→ edit first_ruby_program.rb

# Make executable
→ chmod +x first_ruby_program.rb

# Run it!
→ ./first_ruby_program.rb

Hello World
It is now 2016-03-14 9:01:44 +0100
```

HelloWorld.rb

```
1  #!/usr/bin/ruby -w
2  puts "Hello World"
3  puts "It is now #{Time.now}"
4
```

shebang

# 6.3 The Ruby Programming Language

# Ruby's Language Features



Purely object-oriented (everything is an object)

Dynamically typed

Message passing is only control structure

Single, class-based inheritance

Automatic garbage collection

Exception handling

First-class closures (blocks)

Advanced reflective facilities

Like Smalltalk

Features for program evolution (e.g., mixins)

Embedded scripting facilities (e.g. regular expressions)

Large standard library

# Numbers, Booleans



```
# Numbers:
    1   77.0   -12   12.5e23

# Booleans:
    true   false

# Comparison:
    ==    !=   < >   =~   !~
```

# Strings

String literals:

```
“Hello\nWorld”      # equivalently: %Q{Hello\nWorld}
'Hello World’       # equivalently: %q{Hello World}
```

Double quoted strings

special characters:

```
"Ruby\nRuby\nRuby\n"
```

expression interpolation:

```
"Good night, #{name}"
"Good night, #{name.capitalize}"
"#{$greeting}, #{@name}"
```

Difference between `'...'` and `"..."` is the amount of processing Ruby does on the string.

Substitutes expression `#{expr}` in the string by its value converted by `to_s`

# String Manipulation



Find-and-replace operator:

```
x = "Ruby\nRuby\nRuby\n"
x["Ruby"] = "Dear"
x  => "Dear\nRuby\nRuby\n"
```

Break up a single string into an array

```
“Ruby\nRuby\nRuby\n".split("\n")      # equivalently: x.lines.to_a
```

Join an array of strings into a single string

```
["Ruby", "Ruby", “Ruby"].join("\n")   # equivalently: x.join
```

Remove extra spaces before and at the end of a string

```
"   Hello    there    ".strip # => "Hello    there"
```

Arithmetic operators on strings

```
"Ruby" * 3 # => "RubyRubyRuby"
```

And many, many, many more:

```
upcase, capitalize, center, chars, chomp, end_with?, (g)sub, …
```

# Regular Expressions

try rubular.com for your regex needs!

```
# Notation:
  /this|that/      # matches either 'this' or ‘that’
  /this|that/i     # same, but case insensitive
  /th(is|at)/      # use of parens in patterns
  /ab+c/           # matches 'a', then  one or more 'b', then ‘c'
  /ab*c/           # matches 'a', then zero or more 'b', then ‘c'
  /\s\d\w./        # whitespace, digit, char in word, any char
  %r{this|that}i   # alternative notation with %r

# Matching:
   “charles@uclouvain.BE” =~ /(.*)@(.*)\.be$/i
   => false if no match
   => if match, $1…$n capture parenthesised groups:
      $1 == ‘charles’, $2 == ‘uclouvain’

# Substitution:
line.sub(/this|that/,  'foo') # replaces first occurrence
line.gsub(/this|that/, 'foo') # replaces every occurrence
```

# Symbols



A symbol is a literal that evaluates to itself

Symbols are used wherever there is a limited number of possibilities, whereas strings are for arbitrary data

```
# Symbols:
  :a    :red

  framework = :rails
  :rails.to_s() == “rails”
  “rails”.to_sym() == :rails
  :rails == “rails”    # => false
```

Symbols serve as tokens that denote identity

Symbols connote specialness

# Arrays & Hashes



```
# Arrays:
    []  [1,2,3]  [1,true,"Hello"]
    x = [1,true,”Hello”]
    x[1] == true; x.length == 3

# Hashes (a.k.a. dictionaries):
    { :PGD => 18, :GD => 16, :D => 14, :S => 12 }
    w = { ‘a’ => 1, :b => [2, 3] }
    w[:b][0] == 2
    w.keys == [‘a’, :b]
```

# Everything Is An Object



```
42.class              => Fixnum
:foo.class            => Symbol
"Ruby".class          => String
/ab+c/.class          => Regexp
true.class            => TrueClass
[1,2,3].class         => Array
{ :a => 1 }.class     => Hash
nil.class             => NilClass
Fixnum.class          => Class
```

Similar to Smalltalk

No primitive types as in Java

```
# nil is an object
# a class is an object
```

```
50.methods  # => [:to_s, :inspect, :-@, :+, :-, :*, :/, …]
nil.respond_to?(:to_s)  # => true
my_str.length          my_str.send(:length)
1 + 2                  1.send(:+, 2)
my_array[4]            my_array.send(:[], 4)
my_array[3] = 'foo'    my_array.send(:[]=, 3, 'foo')
if (x == 3) …          if (x.send(:==, 3)) …
my_func(z)             self.send(:my_func, z)
```

# Message Passing



```
# Parameterless messages
  "Ruby".reverse()
# Kernel messages with parameters
  Kernel.print("Ruby")
  Kernel.print("Ruby","Ruby","Ruby")
# Message chaining
  "Ruby".reverse().upcase().length()
# Undefined method error
  "Ruby".foo()
# Arithmetic operators
  -12.abs()
  4.+(5)
  "Ruby".*(3)
```

receiver.message($arg_1$,$arg_2$,...,$arg_n$)

receiver can be dropped for Kernel methods

Every message send produces a result (or an error)

parentheses are optional

# Message Passing



```
# Every operation is initiated through message passing
# a.b means: send message b to receiver a
# does not mean: b is an instance variable of a


# Consider these 3 different uses of '+':
y = 3 + 5                    => 8
y = [1,2] + ['foo',:bar]     => [1,2,'foo',:bar]
y = 'hello' + 'world'        => 'hello world'
# These are all message sends, not language operators.
# They result in the invocation of Numeric#+, Array#+, String#+
```

# Shorthand Notation



```
# Parameterless messages
  "Ruby".reverse
# Message chaining
  "Ruby".reverse.upcase.length
# Messages with parameters
  print "Ruby"
  print "Ruby", "Ruby", "Ruby"
# Arithmetic operators
  -12.abs
  4 + 5
  "Ruby" * 3
```

# Poetry Mode



```
# Make code readable almost as plain English:

redirect_to login_page and return unless logged_in?

a.should be >= 7

link_to :controller => :users, :action => :show


# Equivalent parenthesised expressions:

redirect_to(login_page) and return() unless logged_in?

a.should(be() >= 7)

link_to({:controller => :users, :action => :show})
```

# Method Naming Conventions



```
# Methods use snake_case

# Predicate methods end by “?”
    0.zero?
    [1,2,3].include? 2

# Destructive methods end by “!”
    a = "Ruby"
    a.reverse!
    # usually non-destructive variant available:
    a.reverse

# Conversion methods start by to_
    40.to_s.reverse
    "40".to_i
    (1..3).to_a
```

# Method Definition



```
# Arguments are passed by reference

def foo(x, y)
  return [x,y+1]
end

def foo(x, y=0)   # y is optional, 0 if omitted
   [x,y+1]        # last expression returned as result
end

# Call with   a, b = foo(x, y)
# or          a, b = foo(x)
```

# Statements

```
# Statements end with ‘;’ or newline

def foo(x, y)
   [x,y+1]
end

def foo(x, y); [x,y+1]; end
```

# Control Structures

## classic notation:

```
if <cond>
  <body>
elsif <cond>
  <body>
else
  <body>
end
```

```
while <cond>
  <body>
end
```

```
until <cond>
  <body>
end
```

## postfix notation:

```
<expression> if <cond>
```

```
<expression> unless <cond>
```

In Ruby, only `nil` and `false` represent falseness in conditions

```
<expression> while <cond>
```

```
<expression> until <cond>
```

# Object Creation



Ex-nihilo

- object literals
- (without an explicit constructor message)

Instance creation method

- To create objects from classes
  - `Array.new`, `Hash.new`, `ComplexPolar.new(r,teta)`

Initialize method

- initializes instance variables upon object creation
- is called by `new`, passing all parameters that were passed to `new`

# Class Definition



```
class ComplexCartesian

  # constructor method
  def initialize(x,y)
    @x = x    # instance variable
    @y = y    # instance variable
  end

  # method definitions
  def r
    Math.sqrt(@x**2 + @y**2)
  end

  def theta
    Math.atan2(@y, @x)
  end

end

# instance creation
c = ComplexCartesian.new(1,2)
# using the instance
puts c.r
```

Attention:

@x is an instance variable

('@' is part of its name)

x is a normal variable

No separators for statements.

Each statement on a separate line.

Indentation is not significant.

No need to explicitly declare instance variables

# Scoping and Naming of Variables



Syntax:

Global variables (globally visible)

```
$pi
```

Class variables (same for all instances)

```
@@epsilon
```

Instance variables (unique for each instance)

```
@x, @y
```

Local (local to a method)

```
temp = y/x
```

Block (parameters of a block)

```
{ |entry| entry * entry }
```

Global and class variables are seldom used

Variables are dynamically typed

Variables don't contain values but pointers to objects (like in Java or Smalltalk)

Rules:

Variables_use_snake_case

Constants start with uppercase and so most class and module names

# Accessor Methods



```
class ComplexCartesian

  ...

  # Getter methods
  def x
    @x
  end

  def y
    @y
  end

  # Setter methods
  def x=(new_x)
    @x = new_x
  end

  def y=(new_y)
    @y = new_y
  end

end
```

```
# Constructor method
def initialize(x,y)
  @x = x
  @y = y
end

# Method definitions
def r
  Math.sqrt(x**2 + y**2)
end

def theta
  Math.atan2(y,x)
end
```

# Accessor Methods Shorthand

```
class ComplexCartesian

   …

   attr_reader :x, :y

   attr_writer :x, :y

   …

end
```

or simply:

```
attr_accessor :x, :y
```

# Access Control



Ruby provides 3 levels of access control for methods

- checked at run-time, not statically

1. `public`
   - can be called by anyone
   - default when nothing is specified
2. `protected`
   - can be called by any object of the defining class and its subclasses, very rarely used
3. `private`
   - only on implicit receiver ( `meth(arg)`, not `self.meth(arg)` )

# Accessor Methods



```
class ComplexCartesian

   # public by default
   # Getter methods
   def x
     @x
   end

   def y
     @y
   end

   private
   # Setter methods
   def x=(new_x)
     @x = new_x
   end

   def y=(new_y)
     @y = new_y
   end

   #everything remains private here until declared otherwise
end
```

# Arrays, Variables & Side-Effects



```
[12,46,35].max          # => 46
[12,46,35].reverse      # => [35, 46, 12]

list = [7,4,1]          # => [7, 4, 1]
list.sort               # => [1, 4, 7]
list                    # => [7, 4, 1]
list.sort!              # => [1, 4, 7]
# has destructively modified the list:
list                    # => [1, 4, 7]
list.reverse!           # => [7, 4, 1]
# has destructively modified the list

[ 1, 2 ] << "c" << "d" << [ 3, 4 ]
  # => [1, 2, "c", "d", [3, 4]]
```

# Blocks

Similar in usage to Smalltalk blocks

(though not exactly the same)

Blocks are “chunks of code”

that can be passed to a method

not really first-class functions

but Ruby also has a notion of Procs which are first class objects (see later)

Examples:

```ruby
3.times { print "Ruby" }                    # prints 'Ruby' 3 times
3.times do print "Ruby" end                 # alternative syntax

['I', 'like', 'Ruby'].each do |entry|       # using block to iterate
  print entry, ' '                          # over a collection
end
[1,2,3,4,5].map { |entry| entry * entry }
  # => [1, 4, 9, 16, 25]

fact = 1
1.upto(5) { |i| fact *= i }
fact
```

# Yielding Blocks



Blocks are chunks of code that can be passed to methods

for example to implement iterators

Those methods can execute that block when needed

by using the `yield` keyword

```ruby
def say_something
  yield("I", "like")
  yield("We all", "love")
end

say_something do |person, verb|
  puts "#{person} #{verb} Ruby"
end
```

Block parameters are local to the block

```
a = [1, 2, 3]
a.each { |i| puts i }
i # => NameError: undefined local variable or method `i'
```

```
a = [1, 2, 3]
i = 42
a.each { |i| puts i }
i # => 42
```

# Functional Side: Simplify The Loops!



```ruby
def factorial(n)

  # traditional loop
  fact = 1
  for i in (1..n)
    fact *= i
  end
  return fact

  # becomes
  (1..n).inject(1) { |fact, i| fact * i }

  # or even
  (1..n).inject(1, :*)

end

factorial(5) # => 120
```

# Scripting: Set mtime From exiftime

A simple script to set the modification time of a JPG picture from its “time of capture”

```ruby
#!/usr/bin/env ruby

Dir["**/*.{jpg,JPG}"].each do |picture|
  times = `exiftime #{picture}`
        .scan(/(?:\d+:\d+:\d+ ?){2}/)
        .map { |time| Time.new(*time.split(/\W/) }

  unless times.all? { |time| time == times.first }
    raise "Not same times: #{times}"
  end
  File.utime(File.atime(picture), times.first, picture)
end
```

# 6.4 Worked Out Example

# Example: Complex Numbers



```ruby
class ComplexCartesian                  # Class definition

   EPSILON = 0.00000001                 # Constant

   def initialize(x,y)                  # Initialize instance vars
     @x, @y = x, y                      # Parallel declaration
   end

   private                              # Private methods from here
   attr_writer :x, :y                   # Setter methods

   public                               # Public methods from here
   attr_reader :x, :y                   # Getter method

   def r                                # Getter method with lazy init.
     @r ||= Math.hypot(x, y)            # Use of Math module
   end

   …
```

@r ||= expr
is shorthand for

# Example: Complex Numbers



```
class ComplexCartesian
  …

  def theta
    @theta ||= Math.atan2(y,x)
  end

  def to_s                          # Conversion method to string
    "#{x} + #{y}i"                  # expression extrapolation
  end

  def to_cartesian                  # More conversion methods
    self                            # 'self' is receiver object
  end

  def to_polar
    ComplexPolar.new(r, theta)      # Instance creation
  end

  …
```

to_s is called by Ruby automatically whenever it wants to print an object

# Example: Complex Numbers



```
class ComplexCartesian
   …

   def ==(c)                                  # Equality method
     same?(x, c.x) and same?(y, c.y)
   end

   private                                    # Auxilary method (private)
   def same?(x, y)
     (x - y).abs < EPSILON                    # Use of constant
   end

   public                                     # Public methods again
   def +(c)                                   # Arithmetic operators
     ComplexCartesian.new(self.x + c.x, self.y + c.y)
   end

   def *(c)
     ComplexPolar.new(self.r * c.r, self.theta + c.theta)
   end
   …
end
```

# Example: Complex Numbers (Polar Coords.)



```
class ComplexPolar < ComplexCartesian          # Inheritance

  def initialize(r, theta)
    @r, @theta = r, theta
  end

  def x
    @x ||= @r * Math.cos(@theta)
  end

  def y
    @y ||= @r * Math.sin(@theta)
  end

  private
  attr_writer :r, :theta

  public
  attr_reader :r, :theta

  …
```

# Example: Complex Numbers (Polar Coords.)



```
class ComplexPolar < ComplexCartesian
...

   # to_s                                    # Inherited methods
   # arithmetic operators
   # private def same?(x,y)

   def to_cartesian
     ComplexCartesian.new(x,y)
   end

   def to_polar
     self
   end

   def ==(c)                                 # Overridden method
      same?(r, c.r) and same?(theta, c.theta)
   end

end
```

# Example: Unit Testing



```
require 'test/unit'                                  # import Test::Unit module
class TC_ComplexTest < Test::Unit::TestCase  # define test class

  def setup                                          # setup infrastructure
    @cc = ComplexCartesian.new(1, 1)
    @cp = ComplexPolar.new(Math.sqrt(2), Math::PI / 4)
    ...
  end

  def test_self_comparison                           # test methods
    assert_equal @cc, @cc                            # assert equality
    assert_equal @cp, @cp
  end

  def test_comparison_polar
    assert_equal @cp, @cc.to_cartesian
    assert_equal @cp, @cc.to_polar
  end

  def test_addition
    assert_equal ComplexCartesian.new(2, 2), @cp + @cp
  end

  ...
end
```

# Example: Behavioural Testing With Spec



```
require ‘rspec'                                    # import RSpec gem

describe 'Complex numbers' do                      # define example group
  let :cartesian do                                # setup infrastructure
    ComplexCartesian.new(1, 1)
  end
  let :polar do
    ComplexPolar.new(Math.sqrt(2), Math::PI / 4)
  end

  it 'is identical to itself' do                   # example
    cartesian.should == cartesian                  # in natural order
    polar.should == polar
  end

  it 'can be converted to the other type and still be equal' do
    polar.to_cartesian.should == cartesian
    cartesian.to_polar.should == polar
  end

  it 'supports addition' do
    (cartesian + cartesian).should == ComplexCartesian.new(2, 2)
  end

  …
end
```

# 6.5 Conclusion

# Conclusion



Ruby has become a popular language

Very low-entry cost

- read-eval-print loop
- plug-and-play

Easy digestible and readable syntax

Extensive libraries: Rubygems

Very powerful (compared to other main-stream applications)

“Killer-app”: Ruby-on-Rails

Advanced reflective and meta-programming facilities

Advanced scripting facilities

# Conclusion

## Some negative points

- automatic declaration of instance variables
- often things can be done in many different ways (alternative syntax, shorthand notations)
  - can be seen as positive: use the one that works best for you
  - can be seen as negative: need to be able to read the one that doesn't work best for you as well
- a method in a subclass can override its parent method's visibility rules

```
class Base
  def a_method
    puts "Got here"
  end
  private :a_method
end
```

```
class Derived1 < Base
  public :a_method
end

class Derived2 < Base
end
```

# Next Lecture

More on Ruby's

- advanced language features
  - higher-order programs
  - singleton methods, objects and class methods
  - mixin modules
  - open classes
- advanced reflective and meta-programming facilities