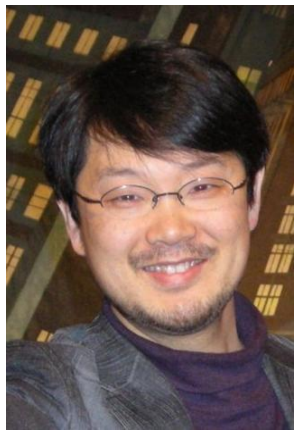
The Ruby programming language

Laura Farinetti
Dipartimento di Automatica e Informatica
Politecnico di Torino
laura.farinetti@polito.it

Ruby



- Powerful and flexible programming language that can be used alone or as part of the Ruby on Rails web framework
- Released in 1995 by Yukihiro Matsumoto with the goal to design a language that emphasize human needs over those of the computer
  - Named after his birthstone

Ruby
A Programmer's Best Friend

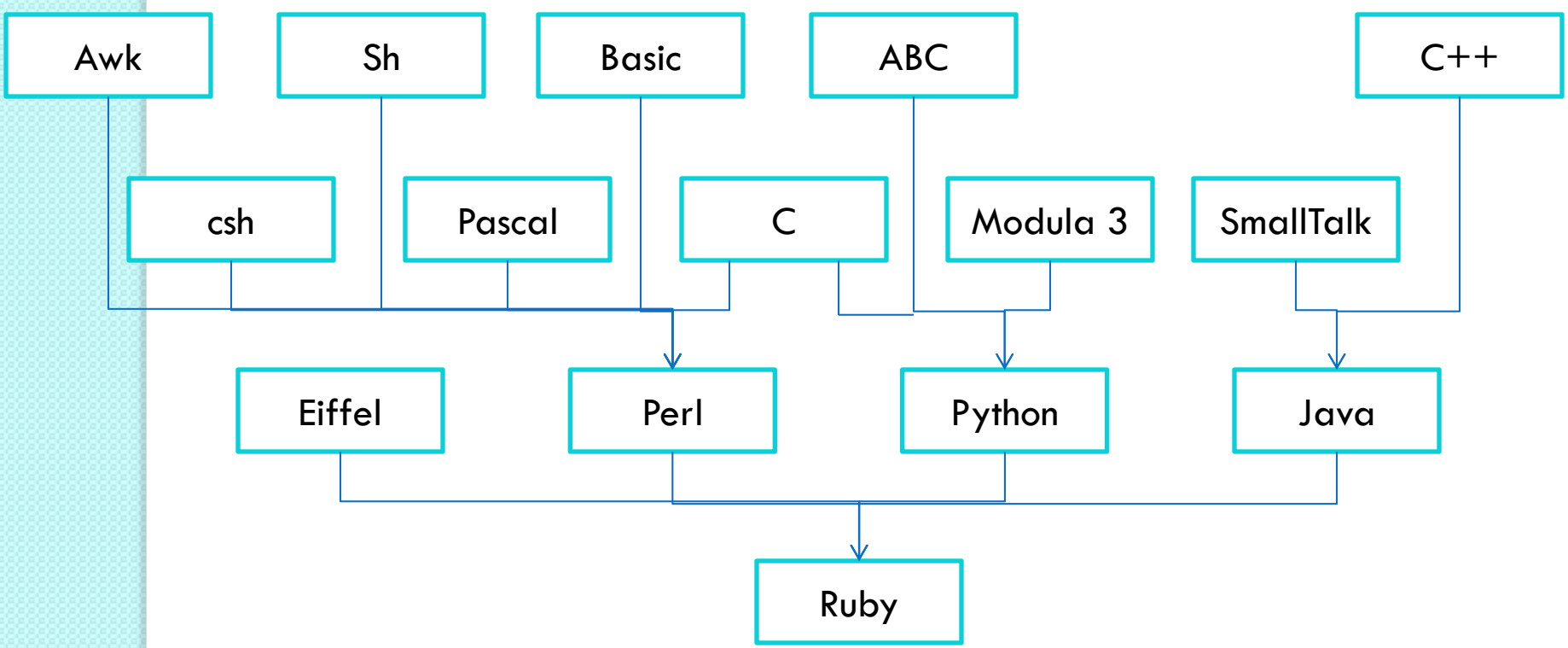
Ruby

- "Designed to maximize programmer happiness" or in other words for "programmer productivity and fun"
  - Emphasis towards the convenience of the programmer rather than on optimization of computer performance
- Follows the principle of least surprise (POLS), meaning that the language behaves in such a way as to minimize confusion for experienced users
- In Japan is more popular than Python

Ruby

- Freely available and open-source
- Highly portable, works on Linux, UNIX, DOS, Windows 95/98/NT/2K, Mac, …
- Main characteristics:
  - High-level: easy to read and write
  - Interpreted: no need for a compiler
  - Object-oriented: it allows users to manipulate data structures called objects in order to build and execute programs
- Pure object-oriented language (even the number 1 is an instance of a class)

Ruby and its ancestors



Rails (Ruby on Rails…)

- A framework for building web applications in Ruby based on the MVC pattern
- A set of strong ideological opinions about how web applications should be structured
- A particularly good tool for building prototypes rapidly
- A thriving, productive, fractious community of free software developers
- A constantly growing and changing software ecosystem with libraries to do almost anything

Object-oriented language

- Ruby is a real object-oriented language
  - Everything you manipulate is an object
  - The results of manipulations are objects
- Classes: categories of things that need to be represented in code
  - E.g. the class “song”
- A class is a combination of state (e.g. the name of the song) and methods that use that state (e.g. a method to play that song)
- Instances of a class: specific “individuals”

Example of class in Ruby

```
# class that models a plain text document

Class Document
  attr_accessor :title, :author, :content

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

  def words
    @content.split
  end

  def word_count
    words.size
  end
end
```

Object-oriented language

- Instances of classes

```
song1 = Song.new("Yesterday")
empty = Array.new
```

- Everything is an object
  - Methods can be applied to data directly, not just on variables holding data

```
puts "Yesterday".length       # prints 9
puts "Rick".index("c")        # prints 2
puts 42.even?                 # prints true
5.to_s                        # returns "5"
```

A simple program

- File extension: .rb
- Rubymine environment, empty project

```
puts "Hello world!"
puts "It is now #{Time.now}"
# prints Hello world!
#        It is now 2014-03-06 09:28:40 +0100

def say_goodnight(name)
  result = "Good night, #{name}"
  return result
end
puts say_goodnight('Pa')      # Good night, Pa

def say_goodnight(name)
  result = "Good night,\n#{name.capitalize}"
  return result
end
puts say_goodnight('mary')    # Good night,
                              # Mary
```

Syntax basics

- Ruby is case sensitive
- Ruby indentation convention: 2 spaces per indent, never use tabs
- Anything following a # is a comment
- Expressions are delimited by newlines (or ; if in the same line)
- Parentheses are optional
- Convention for names

Example of parentheses

```
def find_document (title, author)
  # body omitted
end
...
find_document ('Frankenstein','Shelley')
...
def words()
  @content.split()
end
...
if (word.size < 100)
  puts 'The document is
end
```

```
def find_document title, author
  # body omitted
end
...
find_document 'Frankenstein','Shelley'
...
def words
  @content.split
end
...
if word.size < 100
  puts 'The document is not very long.'
end
```

Variables

- Ruby is weakly typed: variables receive their types during assignment
- Four types of variable
  - Global variables (visible throughout a whole Ruby program) start with '$'
  - Local variables (defined only within the current method or block) start with a lowercase letter or '_'
  - Instance variables start with '@'
  - Class variables start with '@@'

Ruby naming conventions

- Initial characters
  - Local variables, method parameters, and method names: lowercase letter or '_'
  - Global variable: '$'
  - Instance variable: '@'
  - Class variable: '@@'
  - Class names, module names, constants: uppercase letter
- Multi-word names
  - Instance variables: words separated by underscores
  - Class names: use MixedCase (or "CamelCase")

Ruby naming conventions

- End characters
  - ? Indicates method that returns true or false to a query
  - ! Indicates method that modifies the object rather than returning a copy

```
puts 42.even?          # prints true

a = [1,2,3]
a.reverse
print a                # prints [1,2,3]

a.reverse!
print a                # prints [3,2,1]
```

Syntactic structure

- The basic unit of syntax in Ruby is the expression
- The Ruby interpreter evaluates expressions, producing values
- The simplest expressions are primary expressions, which represent values directly
  - e.g. number and string literals, keywods as true, false, nil, self
- More complex values can be written as compound expressions

```
[1,2,3]                    # an array literal
{1=>"one", 2=>"two"}       # a hash literal
1..3                       # a range literal
```

Syntactic structure

- Operators perform computations on values
- Compound expressions are built by combining simpler sub-expressions with operators

```
1                   # a primary expression
x = 1               # an assignment expression
x = x + 1           # an expression with two operators
```

- Expressions can be combined with Ruby keywords to create statements

```
if x < 10 then          # if this expression is true
  x = x + 1             # then execute this statement
end                     # marks the end of a conditional
```

Assignment

- Traditional assignment
- Abbreviated assignment (combined with binary operators)
- Parallel assignment

```
x = 1              # set the value x to the value 1
x += 1             # set the value x to the value x+1
x,y,z = 1,2,3      # set x to 1, y to 2, z to 3

x,y = y,x          # parallel: swap the value of x and y
x = y; y = x       # sequential: x and y have the same
                   # value

x = 1,2,3          # x = [1,2,3]
x, = 1,2,3         # x = 1, the rest is discarded
x,y,z = 1,2        # set x to 1, y to 2, z to nil
```

True, false and nil

- Are keywords in Ruby
- 'true' and 'false' are the two boolean values
  - 'true' is not 1, and 'false' is not 0
- 'nil' is a special value reserved to indicate the absence of value
- When Ruby requires a boolean value, 'nil' behaves like 'false', and any other values behaves like 'true'

```
o == nil        # is the object o nil?
o.nil?          # the same
```

Method invocation

- A method invocation expression is composed of four parts (only the second is required)
  - An arbitrary expression whose value is the object on which the method is invoked followed by a '.' or '::' (if omitted, the method is invoked on 'self')
  - The name of the method
  - The argument values being passed to the method (parentheses are optional)
  - An optional block of code delimited by curly braces or a 'do..end' pair

Method invocation

- Example

```
puts "Hello world"     # puts invoked on self, with
                       # one string argument
Math.sqrt(2)           # sqrt invoked on object Math
                       # with one argument
message.lenght         # length invoked on object message
                       # with no arguments
a.each {|x| puts x}    # each invoked on object a
                       # with an associated block
```

Control structures

- Familiar set of control structures (most…)
- Include
  - Conditionals: if, unless, case
  - Loops: while, until, for
  - Iterators: times, each, map, upto
  - Flow-altering statements like return and break
  - Exceptions
  - The special-case BEGIN and END statements
  - Threads and other "obscure" control structures: fibers and continuations

If and unless statements

- The usual: if, else, elsif

```
if (score > 10)
  puts "You have done very good!"
elsif (score > 5)
  puts "You have passed."
else
  puts "You have failed :-("
end
```

- Something new: unless
  - With unless the body of a statement is executed only if the condition is false
  - Less mental energy to read and understand

Example

- Concept of read-only document

```
Class Document
  attr_accessor :writable
  attr_reader :title, :author, :content
  ...
  def title= (new_title)
    if @writable
      @title = new_title
    end
end
```

```
def title= (new_title)
  if not @read_only
    @title = new_title
  end
end
```

```
def title= (new_title)
  unless @read_only
    @title = new_title
  end
end
```

While and until statements

- The usual: while
  - Loops while condition is true

```
while ! document.is_printed?
  document.print_next_page
end
```

- Something new: until
  - Loops until condition becomes true

```
until document.is_printed?
  document.print_next_page
end
```

Modifier form

- If, unless, while and until can be considered operators in which the value of the right-hand expression affects the execution of the left-hand one
  - Advantage: collapse in a single sentence for readability

```
unless @read_only
  @title = new_title
end
```

```
@title = new_title unless @read_only
```

```
@title = new_title if @writable
```

```
document.print_next_page while document.pages_available?
```

```
document.print_next_page until document.printed?
```

For and each statements

- The for loop is very familiar

```
fonts = [ 'courier', 'times roman', 'helvetica' ]
...
for font in fonts
  puts font
end
```

- However, each is more frequently used

```
fonts = [ 'courier', 'times roman', 'helvetica' ]
...
fonts.each do |font|
 puts font
end
```

Case statement

- Many variants
  - Note: everything in Ruby returns a value

```
case title
when 'War and peace'
  puts 'Tolstoy'
when 'Romeo and Juliet'
  puts 'Shakespeare'
else
  puts "Don't know"
end
```

```
author = case title
         when 'War and peace'
           'Tolstoy'
         when 'Romeo and Juliet'
           'Shakespeare'
         else
           "Don't know"
end
```

```
author = case title
         when 'War and peace' then 'Tolstoy'
         when 'Romeo and Juliet' then 'Shakespeare'
         else "Don't know"
end
```

Iterators

- Although while, until and for are a core part of the Ruby language, it is more common to write loops using special methods known as iterators
- Iterators interact with the block of code that follow them
- Numeric iterators
- Iterators on enumerable objects

Block structure

- Ruby programs have a block structure
  - Blocks of nested code
- Blocks are delimited by keywords or punctuation and by convention are indented two spaces relative to the delimiters
- Two kinds of blocks in Ruby
  - “True blocks”: chunks of code associated with or passed to iterator methods
  - “Body”: simply a list of statement that represent the body of class definition, a method definition, a while loop or whatever

Block structure

- Example of “true blocks”
  - Curly braces, if single line, or do .. end keywords

```
3.times { print "Ruby! " }

1.upto(10) do |x|
  print x
end
```

- Example of “bodies”
  - No curly braces, but keyword .. end

```
if x < 10 then
  x = x + 1
end
```

  - Methods begin with the keyword, ‘def’, and are terminated with an ‘end’

Numeric iterators

- upto: invokes the associated block once for each integer between the one on which it is invoked and the argument
- downto: the same but from a large integer to a smaller one
- times: when invoked on the integer n, it invokes the associated block n times, passing values 0 through n-1
- step: numeric iteration with floating-point numbers

```
4.upto(6) {|x| print x}        # prints 456
3.times {|x| print x}          # prints 012

# start at 0 and iterates in step of 0.1 until it
# reaches Math::PI
0.step(Math::PI, 0.1) {|x| print x}
```

each iterator

- Defined on a number of classes that are collections of enumerable objects: Array, Hash, Range, …
- each passes each element of the collection to its associated block
- Defined also for the Input/Output object

```
[1,2,3].each {|x| print x}    # prints 123
(1..3).each {|x| print x}     # prints 123

['cat', 'dog', 'horse'].each {|name| print name, " "}
# cat dog horse

File.open(filename) do |f|    # open named file, pass as f
  f.each {|line| print line} # print each line in f
end                           # end block
```

Other enumerable iterators

- 'collect' or 'map': executes its associated block for each element of the enumerable object, and collects the return values into an array
- 'select:' executes its associated block for each element and returns an array of the elements for which the block returns a value other than false or nil
- 'reject': the opposite of select (false or nil)

```
squares = [1,2,3].collect {|x| x*x}          # [1,4,9]
evens = (1..10).select {|x| x%2 == 0}        # [2,4,6,8,10]
odds = (1..10).reject {|x| x%2 == 0}         # [1,3,5,7,9]
```

Statements that alter the control flow

- 'return': causes a method to exit and return a value to its caller
- 'break': causes a loop (or iterator) to exit
- 'next': causes a loop (or iterator) to skip the rest of the current iteration and move to the next one
- 'redo': restarts a loop (or iterator) from the beginning
- 'retry': restarts an iterator, reevaluating the entire expression; used in exception handling
- 'throw/catch': exception propagation and handling mechanism

BEGIN and END

- BEGIN and END are reserved words in Ruby that declare code to be executed at the very beginning and very end of a program

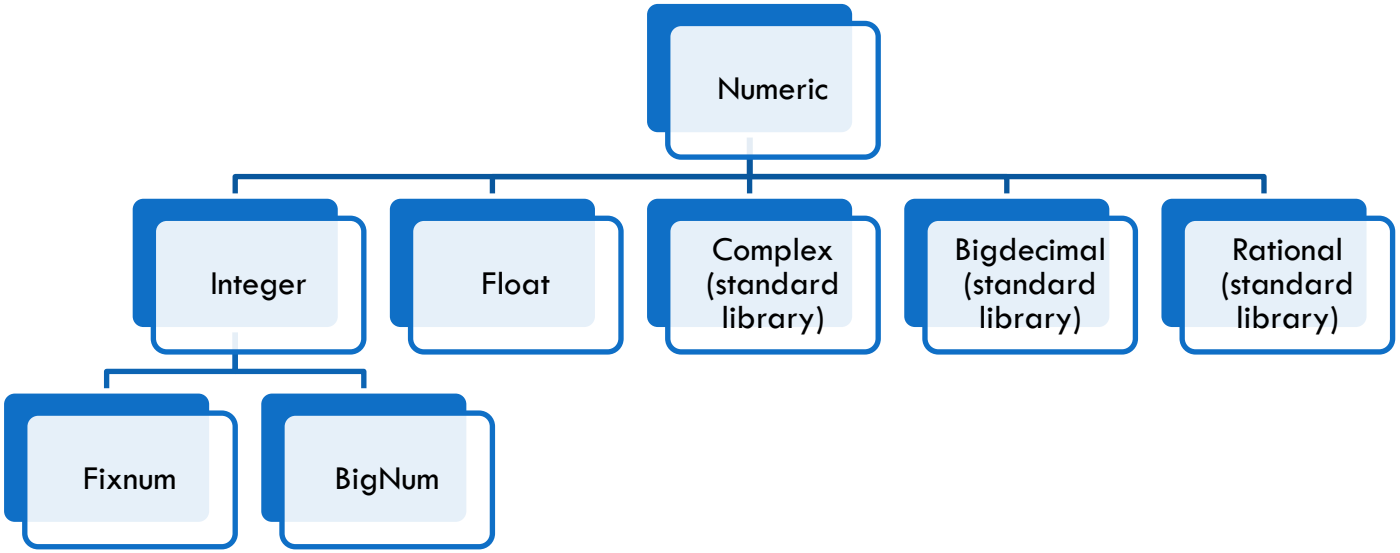
```
BEGIN {
  # Global initialization code goes here
}

END {
  # Global shutdown code goes here
}
```

Basic data types

- Numeric classes
- Strings
- Ranges
- Arrays
- Hashes
- Symbols
- Regular expressions

Numeric classes hierarchy

- Five built-in classes
- Three more in the standard library
- All numbers in Ruby are instances of Numeric



- If an integer value fits within 31 bits it is an instance Fixnum, otherwise it is a Bignum

Examples of literals

```
#Integer literals
0
123
1234567891234567890
1_000_000_000          # One billion
0377                   # Octal representation of 255
0b1111_1111            # Binary representation of 255
0xFF                   # Hexadecimal representation of 255

#Floating-point literals
0.0
-3.14
6.02e23                # This means 6.02 x 10^23
1_000_000.01           # One million and a little bit more
```

Operators

- Arithmetic
  - +, -, *, /, %, ** (exponentiation)
- Comparison
  - ==, <=> (returns -1, 0 or 1), <, <=, >=, >, =~ (matching), eql? (test of equality of type and values)
- Logical
  - and, or, not

Numeric methods

- Also number literals are objects!
- Numeric and its subclasses define many useful methods for determining the class or testing the value of a number

```
# General predicates
0.zero?                         # => true (is this number zero?)
1.0.zero?                       # => false
1.nonzero?                      # => true
1.integer?                      # => true
1.0.integer?                    # => false
1.scalar?                       # => false: not a complex number
1.0.scalar?                     # => false: not a complex number
Complex(1,2).scalar?            # => true
```

Numeric methods

```
# Integer predicates
0.even?                          # => true
1.even?                          # => false

# Float predicates
ZERO, INF, NAN = 0.0, 1.0/0.0, 0.0/0.0       # constants

ZERO.finite             # => true (is this number finite?)
INF.finite              # => false
NAN.finite              # => false

ZERO.infinite           # => nil (infinite positive or negative?)
INF.infinite            # => 1
-INF.infinite           # => -1
NAN.infinite            # => nil

ZERO.nan                # => false (is this number not-a-number?)
INF.nan                 # => false
NAN.nan                 # => true
```

Numeric methods

```
# Rounding methods
1.1.ceil                # => 2: smallest integer >= argument
-1.1.ceil               # => -1: smallest integer >= argument
1.9.floor               # => 1: largest integer <= argument
-1.9.floor              # => -2: largest integer <= argument
1.1.round               # => 1: round to nearest integer
0.5.round               # => 1: round toward positive infinity
-0.5.round              # => -1: round toward negative infinity
1.1.truncate            # => 1: chop off fractional part
-1.1.to_i               # => -1: same as truncate

# Absolute value and sign
-2.0.abs                # => 2.0: absolute value
-2.0<=>0.0              # => -1: sign

# Constants
Float::MAX              # may be platform dependent
Float::MIN              # may be platform dependent
Float::EPSILON          # difference between adjacent floats
```

The Math module

```
# Constants
Math::PI                # => 3.14159265358979
Math::E                 # => 2.71828182845905

# Roots
Math.sqrt(25.0)         # => 5.0: square root
27.0**(1.0/3.0)         # => 3.0: cube root with ** operator

# Logaritms
Math.log10(100.0)       # => 2.0: base-10 logarithm
Math.log(Math::E**3)    # => 3.0: natural logarithm
Math.log2(8)            # => 3.0: base-2 logarithm
...

#Trigonometry
...
```

Text

- In Ruby text is represented by objects of the String class
- Textual patterns are represented as Regexp objects
  - Syntax for including regular expressions

Strings

- Examples of string literals

```
'This is a simple Ruby string literal'
'Won\'t you read O\'Reilly\'s book?'
"\t\"This quote begins with a tab and ends with a newline\"\n"

# Double quoted string literals may include arbitrary Ruby
# expressions
"360 degrees=#{2*Math::PI} radians"
                        #360 degrees=6.28318530717959 radians

# When the expression is simply a global, instance or class
# variable, curly braces can be omitted
$salutation = 'hello'          # Define a global variable
"#$salutation world"           # Use it

%q(Don't worry about escaping ' characters!)
```

Strings

- Multilines strings

```
a_multiline_string = "this is a multiline
string"
another_one = %q{Another multiline
string}
```

- Here documents
  - Begin with << or <<- followed by an identifier or string that specifies the ending delimiter
  - Useful for very long multiline strings

```
document = <<'THIS IS THE END, MY FRIEND, THE END'
  lots and lots of text
  here, with no escaping characters
  ...
THIS IS THE END, MY FRIEND, THE END
```

String interpolation

- With double-quoted strings
- The sequence #{expression} is replaced by the value of the expression
- Arbitrary complex expression are allowed in the #{} construct
- Double-quoted strings can include newlines (\n)

```
planet = "Earth"
"Hello planet #{planet}"      # String interpolation
```

String operators

- Operator + concatenates two strings
- Operator << appends its second operand to its first
- Operand * repeats a text a specified number of times

```
planet = "Earth"
"Hello" + " " + planet                  # Produces "Hello Earth"
"Hello planet ##{planet_number}"        # String interpolation

greeting = "Hello"
greeting << " " << "world"
puts greeting

ellipsis = '.' * 3    # Evaluates to '...'

a = 0
"#{a=a+1} " * 3       # Returns "1 1 1 " and not "1 2 3 "
```

String operators

- Operators == and != compare strings for equality and inequality
- Operators <, <=, >, >= compare the relative order of strings
  - Based on caracters' code
  - String comparison is case sensitive

Characters and substrings

- Accessing characters and substrings

```
s = 'hello'
s[0]                  # the first character
s[s.length-1]         # the last character
s[-1]                 # the last character
s[-2]                 # the second-to-last character
s[-s.length]          # the first character
s[s.length]           # nil: there is no character at that index

s = 'hello'
s[0,2]                # "he"
s[-1,1]               # "o"
s[0,0]                # ""
s[0,10]               # "hello"
s[s.length,1]         # ""
s[s.length+1,1]       # nil
s[0,-1]               # nil (negative length)
```

Characters and substrings

- Modifying characters and substrings

```
s = 'hello'
s[-1] = ""              # deletes the last character
s[-1] = "p!"            # the string is now "help!"

s = 'hello'
s[0,1] = "H"                    # replaces first letter with H
s[s.length,0] = " world"        # appends a new string
s[5,0] = ","                    # inserts a comma without deleting
s[5,6] = ""                     # deletes with no insertion
                                # the string is now "Hellod"
```

Characters and substrings

- Indexing a string with a Range object
- Splitting a string into substrings based on a delimiter

```
s = 'hello'
s[2..3]                 # "ll": characters 2 and 3
s[-3..-1]               # "llo": negative indexes work too
s[0..0]                 # "h": one characters
s[0...0]                # "": this Range is empty
s[2..1]                 # "": this Range is empty
s[7..10]                # nil: this Range is outside the string

s = 'hello'
s[-2..-1] = "p!"        # replacement: s becomes "help!"
s[0...0] = "Please "    # insertion: s becomes "Please help!"
s[6..10] = ""           # deletion: s becomes "Please!"

"this is it".split      # ["This", "is", "it"]
"hello".split('l')      # ["he", "", "o"]
```

Ranges

- A Range object represents the values between a start value and an end value
- Range literals are written placing two or three dots between the start value and the end value
  - Two dots: range is inclusive (end value is part of the range)
  - Three dots: range is exclusive (end value is not part of the range)

```
1..10          # the integers 1 through 10, including 10
1.0...10.0     # the numbers 1.0 through 10.0,
               # excluding 10.0
```

Ranges

- The include? method check if a value is included in a range
- Ordering is implicit in the definition of a range
- Comparison operator '<=>' which compares two operands and evaluates to -1, 0 or 1
  - A value can be used in a range only if it responds to this operator
- Purposes of range: comparison and iteration

```
cold_war = 1945..1989
cold_war.include? birthdate.year

r = 'a'..'c'
r.each {|l| print "[#{l}]"}       # prints "[a][b][c]"
r.step(2) {|l| print "[#{l}]"}    # prints "[a][c]"
r.to_a                            # ['a','b','c']
```

Arrays

- Set of values that can be accessed by position, or index
- Indexed with integers starting at 0
- Methods 'size' and 'length' return the number of elements
- Negative index counts from the end of the array
  - E.g. size – 2 is the second-to-last element
- If you try to read an element beyond the end or before the beginning Ruby returns 'nil' and do not throw an exception
- Ruby's arrays are untyped and mutable: the elements need not be of the same class, and they can be changed at any time

Arrays

- Arrays are dinamically resizeable
- Arrays are objects: must be instantiated with 'new'
- Examples:

```
[1, 2, 3]                 # array of three Fixnum objects
[0..10, 10.. 0]           # array of two ranges
[[1,2],[3,4],[5]]         # array of nested arrays
[x+y, x-y, x*y]           # elements can be arbitrary expressions
[]                        # empty array has size 0

words = %w{this is a test}    # same as words =
                              # ['this', 'is', 'a', 'test']

empty = Array.new             # []: empty array
nils = Array.new(3)           # [nil, nil, nil]
nils = Array.new(4,0)         # [0, 0, 0, 0]
copy = Array.new(nils)        # copy of an existing array
count = Array.new(3) {|i| i+1}    # [1,2,3]: three elements
                                  # computed from index
```

Arrays

- Examples:

```
a = [0, 1, 4, 9, 16]
a[0]                    # first element is 0
a[-1]                   # last element is 16
a[-2]                   # second-to-last element is 9
a[a.size]               # last element
a[-a.size]              # first element
a[8]                    # beyond the end: nil
a[-8]                   # before the beginning: nil

a[0] = "zero"           # a is ["zero", 1, 4, 9, 16]
a[-1] = 1..16           # a is ["zero", 1, 4, 9, 1..16]
a[8] = 64  # a is ["zero", 1, 4, 9, 1..16, nil, nil, nil, 64]
a[-10] = 100            # error: can't assign before beginning
```

Arrays

- Like strings, array can be indexed also
  - by two integers: first element and number of elements
  - by Range objects
- Works also for insertion and deletion

```
a = ('a'..'e').to_a   # range converted to ['a','b','c','d','e']

a[0,0]                # sub-array: []
a[1,1]                # sub-array: ['b']
a[-2,2]               # sub-array: ['d','e']
a[0..2]               # sub-array: ['a','b','c']
a[-2..-1]             # sub-array: ['d','e']
a[0...-1]             # sub-array: ['a','b','c','d']
                      # all but the last element

a[0,0] = [1,2,3]      # insert elements at the beginning of a
a[0,2] = []           # delete those elements
a[-1,1] = ['z']       # replace last elements
a[-2,2] = nil         # replace last two elements with nil
```

Array operators

- The operator '+' concatenates two arrays
- The operator '<<' appends an element to an existing array
- The operator '-' subtracts one array to another
- The operator '*' is used for repetition

```
a = [1,2,3] + [4,5]    # [1,2,3,4,5]
a = a + [[6,7,8]]      # [1,2,3,4,5, [6,7,8]]
a = a + 9              # error: righthand side must be an array

a = []                 # start with an empty array
a << 1                 # a is [1]
a << 2 << 3            # a is [1,2,3]
a << [4,5,6]           # a is [1,2,3, [4,5,6]]
a.concat [7,8]         # a is [1,2,3, [4,5,6], 7,8]

['a','b','c','d','a'] - ['b','c','d']  # ['a','a']

a[0] * 8               # [0,0,0,0,0,0,0,0]
```

Array operators

- Boolean operators '|' and '&' are used for union and intersection
  - These operators are not transitive
- Many useful methods, e.g. 'each'

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

a | b           # [1,2,3,4,5]: duplicates are removed
b | a           # [5,4,3,2,1]: order is different
a & b           # [2,3,4]
b & a           # [4,3,2]

a = ('A'..'Z').to_a   # begin with an array of letters
a.each {|x| print x}  # print the alphabet, one letter
                      # at the time
```

Hashes

- Data structures that maintain a set of objects known as 'keys' and associate a 'value' to each key
- Called also maps or associative arrays
  - The array index is the key insted of an integer
- Keys are almost always symbols

```
numbers = Hash.new      # create a new empty hash object
numbers["one"] = 1      # map the String "one" to the Fixnum 1
numbers["two"] = 2      # map the String "one" to the Fixnum 1
numbers["three"] = 3    # map the String "one" to the Fixnum 1

sum = numbers["one"] + numbers["two"]  # retrieve values
```

Hash literals

- Written as a comma-separated list of key/value pairs, enclosed in curly braces
- Symbol objects work more efficiently than strings

```
numbers = {"one" => 1, "two" => 2, "three" => 3}

numbers = {:one => 1, :two => 2, :three => 3}

# succinct hash literal syntax when symbols are used
numbers = {one: 1, two: 2, three: 3}

# access to hashes
grades = { "Bob" => 82,  "Jim" => 94, "Billy" => 58 }

puts grades["Jim"]              # 94

grades.each do |name,grade|   # Bob: 82
  puts "#{name}: #{grade}"    # Jim: 94
end                           # Billy: 58
```

Symbols

- The Ruby interpreter maintains a symbol table in which it stores the names of all the classes, methods and variables it knows about
  - This allows to avoid most string comparison: names are referred by their position the symbol table
- This symbols can be used also by Ruby programs
- Symbols are simply constant names that need not to be declared and that are guaranteed to be unique
- A symbol literal starts with ':'

```
:symbol                         # a symbol literal
:'another long symbol'          # symbol with spaces
s = "string"
sym = :"#{s}"                   # the symbol :string
```

Symbols

```
# with constants
NORTH = 1
EAST = 2
SOUTH = 3
WEST = 4
walk(NORTH)

# with symbols
walk(:north)
```

- Symbols are often used to refer to method names

```
# does the object o have an each method?
o.respond_to? :each
```

Symbols and hashes

- Symbols are frequently used as keys in hashes

```
inst_section = {
  :cello         => 'string',
  :clarinet      => 'woodwind',
  :drum          => 'percussion',
  :oboe          => 'woodwind',
  :trumpet       => 'brass',
  :violin        => 'string'
}
inst_section[:oboe]          # woodwind
inst_section[:cello]         # string
inst_section['cello']        # nil
```

Symbols and hashes

- Symbols are so frequently used as hash keys that Ruby 1.9 introduced a new syntax

```
inst_section = {
  :cello         'string',
  :clarinet      'woodwind',
  :drum          'percussion',
  :oboe          'woodwind',
  :trumpet       'brass',
  :violin        'string'
}
puts "An oboe is a #{inst_section[:oboe]}"
# An oboe is a woodwind
```

Data type conversion

- Explicit conversion methods
  - to_s: convert to String class
  - to_i: convert to Integer class
  - to_f: convert to Float class
  - to_a: convert to Array class

Regular expressions

- Known also as ‘regexp’ or ‘regex’
- Describe a textual pattern
- Ruby’s Regexp class implements regular expressions and define pattern matching methods and operators
- Regexp literal are delimited by ‘/’

```
/Ruby?/         # matches the text "Rub" followed by
                # an optional "y"
```

Regular expressions

```
^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$
```

- `^[a-zA-Z0-9._-]+`
  - The email address must begin with alpha-numeric characters (both lowercase and uppercase characters are allowed): it may have periods, underscores and hyphens
- `@`
  - There must be a '@' symbol after initial characters
- `[a-zA-Z0-9.-]+`
  - After the '@' sign there must be some alpha-numeric characters; it can also contain period and and hyphens
- `\.`
  - After the second group of characters there must be a period ('.'); this is to separate domain and subdomain names.
- `[a-zA-Z]{2,4}$`
  - Finally, the email address must end with two to four alphabets; {2,4} indicates the minimum and maximum number of characters

Examples of e-mail patterns

```
[a-zA-Z0-9!#$%&'*+/=?^_`{|}~-]+
(?:\.[a-zA-Z0-9!#$%&'*+/=?^_`{|}~-]+)*
@(?:[a-zA-Z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+
[a-z0-9](?:[a-z0-9-]*[a-z0-9])?
```

```
[a-zA-Z0-9!#$%&'*+/=?^_`{|}~-]+
(?:\.[a-zA-Z0-9!#$%&'*+/=?^_`{|}~-]+)*
@(?:[a-zA-Z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+
(?:[a-zA-Z]{2}|com|org|net|edu|gov|mil|biz|
info|mobi|name|aero|asia|jobs|museum)\b
```

Regular expression basic syntax

- Characters

Character	Description	Example
Any character except `[\^$.|?*+()`	All characters except the listed special characters match a single instance of themselves. { and } are literal characters, unless they're part of a valid regular expression token (e.g. the {n} quantifier).	`a` matches `a`
\ (backslash) followed by any of `[\^$.|?*+(){}`	A backslash escapes special characters to suppress their special meaning.	`\+` matches `+`
`\Q...\E`	Matches the characters between `\Q` and `\E` literally, suppressing the meaning of special characters.	`\Q+-*/\E` matches `+-*/`
`\xFF` where FF are 2 hexadecimal digits	Matches the character with the specified ASCII/ANSI value, which depends on the code page used. Can be used in character classes.	`\xA9` matches © when using the Latin-1 code page.
`\n`, `\r` and `\t`	Match an LF character, CR character and a tab character respectively. Can be used in character classes.	`\r\n` matches a DOS/Windows CRLF line break.
`\a`, `\e`, `\f` and `\v`	Match a bell character (`\x07`), escape character (`\x1B`), form feed (`\x0C`) and vertical tab (`\x0B`) respectively. Can be used in character classes.	
`\cA` through `\cZ`	Match an ASCII character Control+A through Control+Z, equivalent to `\x01` through `\x1A`. Can be used in character classes.	`\cM\cJ` matches a DOS/Windows CRLF line break.

Regular expression basic syntax

- Character classes or character sets [abc]

Character	Description	Example
[(opening square bracket)	Starts a character class. A character class matches a single character out of all the possibilities offered by the character class. Inside a character class, different rules apply. The rules in this section are only valid inside character classes. The rules outside this section are not valid in character classes, except for a few character escapes that are indicated with "can be used inside character classes".	
Any character except ^-]\ add that character to the possible matches for the character class.	All characters except the listed special characters.	[abc] matches a, b or c
\ (backslash) followed by any of ^-]\	A backslash escapes special characters to suppress their special meaning.	[\^\]] matches ^ or]
- (hyphen) except immediately after the opening [	Specifies a range of characters. (Specifies a hyphen if placed immediately after the opening [)	[a-zA-Z0-9] matches any letter or digit
^ (caret) immediately after the opening [	Negates the character class, causing it to match a single character *not* listed in the character class. (Specifies a caret if placed anywhere except after the opening [)	[^a-d] matches x (any character except a, b, c or d)

Regular expression basic syntax

- Character classes or character sets [abc]

Character	Description	Example
`\d`, `\w` and `\s`	Shorthand character classes matching digits, word characters (letters, digits, and underscores), and whitespace (spaces, tabs, and line breaks). Can be used inside and outside character classes.	`[\d\s]` matches a character that is a digit or whitespace
`\D`, `\W` and `\S`	Negated versions of the above. Should be used only outside character classes. (Can be used inside, but that is confusing.)	`\D` matches a character that is not a digit
`[\b]`	Inside a character class, `\b` is a backspace character.	`[\b\t]` matches a backspace or tab character

- Dot

Character	Description	Example
`.` (dot)	Matches any single character except line break characters \r and \n. Most regex flavors have an option to make the dot match line break characters too.	`.` matches `x` or (almost) any other character

Regular expression basic syntax

- Anchors

Character	Description	Example
^ (caret)	Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the caret match after line breaks (i.e. at the start of a line in a file) as well.	^. matches a in abc\ndef. Also matches d in "multi-line" mode.
$ (dollar)	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the dollar match before line breaks (i.e. at the end of a line in a file) as well. Also matches before the very last line break if the string ends with a line break.	.$ matches f in abc\ndef. Also matches c in "multi-line" mode.
\A	Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Never matches after line breaks.	\A. matches a in abc
\Z	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks, except for the very last line break if the string ends with a line break.	.\Z matches f in abc\ndef
\z	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks.	.\z matches f in abc\ndef

Regular expression basic syntax

- Word boundaries

Character	Description	Example
\b	Matches at the position between a word character (anything matched by \w) and a non-word character (anything matched by [^\w] or \W) as well as at the start and/or end of the string if the first and/or last characters in the string are word characters.	.\b matches c in abc
\B	Matches at the position between two word characters (i.e the position between \w\w) as well as at the position between two non-word characters (i.e. \W\W).	\B.\B matches b in abc

- Alternation

Character	Description	Example
| (pipe)	Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of options.	abc|def|xyz matches abc, def or xyz
| (pipe)	The pipe has the lowest precedence of all operators. Use grouping to alternate only part of the regular expression.	abc(def|xyz) matches abcdef or abcxyz

Regular expression basic syntax

- Quantifiers

Character	Description	Example
? (question mark)	Makes the preceding item optional. Greedy, so the optional item is included in the match if possible.	`abc?` matches `ab` or `abc`
??	Makes the preceding item optional. Lazy, so the optional item is excluded in the match if possible. This construct is often excluded from documentation because of its limited use.	`abc??` matches `ab` or `abc`
* (star)	Repeats the previous item zero or more times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is not matched at all.	`".*"` matches `"def" "ghi"` in `abc "def" "ghi" jkl`
*? (lazy star)	Repeats the previous item zero or more times. Lazy, so the engine first attempts to skip the previous item, before trying permutations with ever increasing matches of the preceding item.	`".*?"` matches `"def"` in `abc "def" "ghi" jkl`
+ (plus)	Repeats the previous item once or more. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only once.	`".+"` matches `"def" "ghi"` in `abc "def" "ghi" jkl`

Regular expression basic syntax

- Quantifiers

Character	Description	Example
+? (lazy plus)	Repeats the previous item once or more. Lazy, so the engine first matches the previous item only once, before trying permutations with ever increasing matches of the preceding item.	".+?" matches "def" in abc "def" "ghi" jkl
{n} where n is an integer >= 1	Repeats the previous item exactly n times.	a{3} matches aaa
{n,m} where n >= 0 and m >= n	Repeats the previous item between n and m times. Greedy, so repeating m times is tried before reducing the repetition to n times.	a{2,4} matches aaaa, aaa or aa
{n,m}? where n >= 0 and m >= n	Repeats the previous item between n and m times. Lazy, so repeating n times is tried before increasing the repetition to m times.	a{2,4}? matches aa, aaa or aaaa
{n,} where n >= 0	Repeats the previous item at least n times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only n times.	a{2,} matches aaaaa in aaaaa
{n,}? where n >= 0	Repeats the previous item n or more times. Lazy, so the engine first matches the previous item n times, before trying permutations with ever increasing matches of the preceding item.	a{2,}? matches aa in aaaaa

Examples of Regex in Ruby (1/3)

```
# Literal characters
/ruby/          # match "ruby"

# Character classes
/[Rr]uby/       # match "Ruby" or "ruby"
/rub[ye]/       # match "ruby" or "rube"
/[aeiou]/       # match any lowercase vowel
/[0-9]/         # match any digit
/[a-z]/         # match any lowercase ASCII character
/[A-Z]/         # match any uppercase ASCII character
/[a-zA-Z0-9]/   # match any of the above
/[^aeiou]/      # match anything other than a lowercase vowel
/[^0-9]/        # match anything other than a digit

# Special character classes
/./             # match any character except newline
/d/             # match a digit
/D/             # match a non-digit
/s/             # match a whitespace character [\t\r\n\f]
/S/             # match a non-whitespace character
/w/             # match a single word character: [a-zA-Z0-9_]
/W/             # match a non-word character: [^a-zA-Z0-9_]
```

Examples of Regex in Ruby (2/3)

```
# Repetition
/ruby?/         # match "rub" or "ruby": "y" is optional
/ruby*/         # match "rub" plus 0 or more "y"
/ruby+/         # match "rub" plus 1 or more "y"
/\d{3}/         # match exactly 3 digits
/\d{3,}/        # match 3 or more digits
/\d{3,5}/       # match 3, 4 or 5 digits

# Nongreedy repetition: match the smallest number of
# repetitions
/<.*>/          # greedy repetition: match "<ruby>perl>"
/<.*?>/         # nongreedy repetition: match "<ruby>"

# Grouping with parentheses
/\D\d+/                 # no group: repeat \d
/(\D\d)+/               # group: repeat \d\D pair
/([Rr]uby(, )?)+/       # match "Ruby", "Ruby, ruby, ruby", ecc.

# Backreferences: matching a previously matched group again
  # \1 matches whatever the first group matched
  # \2 matches whatever the second group matched
/([Rr]uby&\1ails/       # match "ruby&rails" or "Ruby&Rails"
/(['"])[^\1]*\1/        # single or double quoted string
```

Examples of Regex in Ruby (3/3)

```
# Alternatives
/ruby|rube/    # match "ruby" or "rube"
/rub(y|le)/    # match "ruby" or "ruble"
/ruby(!+|\?)/  # match "ruby" followed by one or more ! or one ?

# Anchors: specifying match position
/^Ruby/        # match "Ruby" at the start of a string
/Ruby$/        # match "Ruby" at the end of a string
/\ARuby/       # match "Ruby" at the start of a string
/Ruby\Z/       # match "Ruby" at the end of a string
/\bRuby\b/     # match "Ruby" at a word boundary
/\bruby\B/     # \B is a non-word boundary
               # match "rub" in "ruby" or "rube" but not alone
/Ruby(?=!)/    # match "Ruby" if followed by an !
/Ruby(?!!)/    # match "Ruby" if not followed by an !

# Special syntax with parentheses
/R(?#comment)/         # match "R", all the rest is a comment
/R(?i)uby/             # case insensitive while matching "uby"
/R(?i:uby)/            # same thing
```

Pattern matching with regular expressions

- Ruby's basic pattern matching operator is '=~'
  - One operand is a regular expression and the other is a string
- '=~' checks its string operand to see if it, or any substring, matches the pattern specified by the regular expression
  - If a match is found, it returns the string index at which the first match begin
  - If not, it returns 'nil'

```
pattern = /Ruby?/i   # match "Rub" or "Ruby", case insensitive
pattern =~ "backrub"          # returns 4
"rub ruby" =~ pattern         # returns 0
pattern =~ "r"                # returns nil
```

Pattern matching with regular expressions

- After any succesfull (non-nil) match, the global variable '$~' holds a MatchData object which contains complete information about the match

```
"hello" =~ /e\w{2}/   # match an e followed by 2 word characters
$~.string             # "hello" : the complete string
$~.to_s               # "ell" : the portion that matched
$~.pre_match          # "h" : the portion before the match
$~.post_match         # "o" : the portion after the match
```

Classes

- Creation

```
class Point
end
```

- Instantiation

```
p = Point.new
p.class                 # Point
p.is_a? Point           # true
```

- Defining a method: initialization
  - The initialize method is special: the new methosd creates an instance object and then automatically invokes the initialize method on that instance
  - The arguments passed to new are also passed to initialize
  - @x and @y are instance variables

```
class Point
  def initialize(x,y)
    @x, @y = x, y
  end
end
p = Point.new(0,0)
```

Classes

- Accessor methods
  - To make variable values accessible

```
class Point
  def initialize(x,y)
    @x, @y = x, y
  end
  def x        # the accessor method for @x
    @x
  end
  def y        # the accessor method for @y
    @y
  end
end

p = Point.new(1,2)
q = Point.new(p.x*2, p.y*3)
```

Classes example

- Defining operators

```
class Point
  attr_reader :x :y   # define accessor methods

  def initialize(x,y)
    @x, @y = x, y
  end

  def +(other)         # define + to do vector addition
    Pont.new(@x + other.x, @y.other.y)
  end

  def -@               # define unary - to negate coordinates
    Pont.new(-@x, -@y)
  end

  def *(scalar)        # define * to do scalar multiplication
    Pont.new(@x*scalar, @y*scalar)
  end
```

Subclasses and inheritance

- Ruby has a class hierarchy in form of a tree
  - The BasicObject class in the root of the tree
- Multiple inheritance is not possible
  - However, modules allow to import methods
  - Modules are named group of methods, constants and class variables, but cannot be instantiated and do not have a hierarchy (standalone)
- Extending a class

```
# class Point3D is a subclass of class Point
class Point3D < Point
end
```

Ruby vs. Java (I)

- Like Java, in Ruby:
  - Memory is managed via garbage collector
  - Objects are strongly typed
  - There are public, private, and protected methods
  - There are embedded doc tools (RDoc)

Ruby vs. Java (II)

- Unlike Java, in Ruby:
  - You use the `end` keyword after defining things like classes, instead of braces around blocks of code
  - `require` instead of `import`
  - All member variables are private
  - `nil` instead of `null`
  - There is no casting
  - Everything is an object

Esercizio 1: stringhe

- Le stringhe si possono unire almeno in due modi: concatenandole (+ o <<) o interpolandole (#{})
- Definite due variabili 'nome' 'cognome', assegnando il vostro valore
- Provate a stampare le stringhe nome o cognome con puts e print: differenze?
- Stampate lo mi chiamo nome e cognome. usando i diversi metodi (con +, << e l'interpolazione)
- Le doppie virgolette e le virgolette singole sono hanno lo stesso effetto? Provare
- Provare ad esprimere l'espressione stampa il contenuto della variabile nome se questa non è vuota, sia nella forma 'tradizionale' che nella 'modifier form'
- Date le stringhe 'Ciao mondo' e 'Ciao_mondo' provate ad applicare il metodo 'split' ad entrambe in modo da stampare ["Ciao", "mondo"]

Esercizio 2: array e range

- Definite l'array vett = [15,30,90,22,70] e provate a stamparlo: puts o print?
- Provate ad applicare i metodi first, last, lenght
- Provate ad applicare i metodi sort, reverse, shuffle e a stampare il vettore: cosa è successo?
- Inserite un altro element in vett, usando push e << : cambia qualcosa?
- Si può aggiungere una stringa a vett? Perché?
- Provate a stampare i primi tre elementi del vettore, usando sia i range inclusivi che esclusivi

Esercizio 3: blocchi

- Prima immaginate cosa fanno le istruzioni

```
animals = %w{cat dog horse rabbit owl}
animals.each {|anim| puts anim}

(1..5).each {|i| puts 2*i}

puts ('a'..'z').to_a.shuffle[0..7].join

my_str = "blah, blah"
puts my_str.split(",")[0].split("")[2] * 3
```

- E poi provatele…

Licenza d'uso

- Queste diapositive sono distribuite con licenza Creative Commons "Attribuzione - Non commerciale - Condividi allo stesso modo 2.5 Italia (CC BY-NC-SA 2.5)"
- Sei libero:
  - di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera
  - di modificare quest'opera
- Alle seguenti condizioni:
  - **Attribuzione** — Devi attribuire la paternità dell'opera agli autori originali e in modo tale da non suggerire che essi avallino te o il modo in cui tu usi l'opera.
  - **Non commerciale** — Non puoi usare quest'opera per fini commerciali.
  - **Condividi allo stesso modo** — Se alteri o trasformi quest'opera, o se la usi per crearne un'altra, puoi distribuire l'opera risultante solo con una licenza identica o equivalente a questa.
- http://creativecommons.org/licenses/by-nc-sa/2.5/it/