

Ruby

A Programmer's Best Friend



Why Ruby?



Ruby is designed to
make programmers
happy

Yukihiro Matsumoto (Matz), creator of Ruby



A Little History



- Work on Ruby started on February 24, 1993
- First public release in 1995
- A C based single pass interpreter till version 1.8.6
- As of version 1.9, released December 2007, it is virtual machine based
- A few other implementations sprang into existence
 - JRuby
 - IronRuby
 - Rubinius



Philosophy



Often people, especially computer engineers, focus on the machines.

They think, "By doing this, the machine will run faster. By doing this, the machine will run more effectively."

They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming.

We are the masters. They are the slaves.



Philosophy



- Focus on humans rather than machines
- Make programming fun
- Principle Of Least Surprise (POLS)
 - Languages should minimize confusion
 - Languages should minimize programmer work
 - Matz does not really believe Ruby adheres to POLS



Origins



Ruby inherited features from several languages

- Message sending -> *SmallTalk*
- Everything is an Object -> *SmallTalk*
- Uniform Access Principle -> *Eiffel*
- Dynamic, concise -> *Perl, Python*
- Higher Order Functions -> *Lisp, Perl*



Syntax Primer



Ruby has a simple concise syntax

Strings are defined in quotes or double quotes

```
"String 1", 'String 2'
```

There are literals for arrays, hashes and ranges

```
[1,2,3,4,"duh!"] # array
```

```
{'a'=>'thing','b'=>'another'} # hash
```

```
1..9 # range
```

Blocks are surrounded by do-end pairs (sometimes { })

```
if $x > 1 do # do..end block  
  puts $x  
end
```

```
if @x > 1 { puts @x+@@y } # {...} block
```



Syntax Primer



Ruby has a simple concise syntax (cont'd)

Functions are invoked using the dot notation

```
method.call(argument)
```

Function parentheses are optional!

```
method.call argument #parentheses are optional
```

Ruby has concise conditionals

```
if not x == y
  puts x
end
```

#equivalent to:

```
puts x if not x == y
```

```
puts x unless x = y
```



Syntax Primer



Ruby has symbols

Symbols are variables whose names are their values!

Only one copy exists of a certain symbol value

Very efficient memory wise, useful in indexing hashes

```
:symbol
```

```
students = [  
  { :name=>'anwar' }, { :name=>'ashraf' },  
  { :name=>'wael' }, { :name=>'akram' },  
  { :name=>'ayman' }  
]
```

Symbols are are immutable



Object Oriented



Every Thing Is An Object



Every Thing Is An Object



In Ruby, everything is an object

- This statement is almost true
- Some (very few) elements are not objects

```
"This is a string".length # 16
"another one".index('t')  # 3
-12.abs                   # 12
3.times { print "hi" }    # hihihi
"Happy Day".downcase.split(//).uniq.sort.join
nil.nil?                  # true
true.false?               # false
fals.false?               # true
```



Every Thing Is An Object



What elements are not objects?

- Code blocks
 - They can be turned to objects though (more on that later)



Defining Classes



Classes are defined using the keyword class

```
class Animal
  #class body goes here
end
```

Methods are defined using the def keyword

```
class Animal
  def eat
    # method body goes here
  end
end
```



Defining Classes



You can use an optional initialization method. That method will be called when a new object of the class' type is created

```
class Animal
  def initialize
    # this method will be called
    # when a new object of type Animal
    # is created
  end
end
```



Defining Classes



Classes are always open. Means you can reopen them and edit them all the time

```
class Animal
  def eat
    #...
  end
end
# Animal objects now have method eat

class Animal #reopen Animal
  def sleep
    #...
  end
end
# Animal objects now have methods eat and sleep
```



Defining Classes



Defined classes can instantiate objects by calling the method “new” on the class object

```
class Animal
  def eat
    #...
  end
end

dog = Animal.new
#or
dog = Animal.new()
```

Remember, classes are objects!



Defining Classes



We can pass parameters to the “new” method and those will be passed to the “initialize” method if one is found

```
class Animal
  def initialize(name)
    #...
  end
end

cat = Animal.new "meshmesh"
#or
cat = Animal.new("meshmesh")
```



Defining Classes



Methods added to classes affect all already instantiated objects and any new ones

```
class Animal
  def eat
  end
end

cat = Animal.new #cat can eat

class Animal
  def sleep
  end
end

#cat can now sleep as well as eat
dog = Animal.new #dog sleeps and eats too
```



Defining Classes



Methods can be added to individual objects after instantiation

```
class Animal
  def eat
    # ..
  end
end

cat = Animal.new #cat can eat

def cat.sleep
  # ..
end
#cat can now sleep as well as eat

dog = Animal.new #dog eats but cannot sleep
```



Access Levels



Ruby classes have different access levels

For methods:

- Public (by default)
- Private
- Protected

For instance variables:

- All instance variables are private, you cannot access them directly from client code.



Access Levels

Example

```
class Animal
  public
  def eat
  end

  private
  def drink
  end
end

cat = Animal.new

cat.eat      #works

cat.drink    #error!
```



Access Levels



What about instance variables?

```
class Animal
  def fight(animal)
    if animal == @enemy # @enemy is an ivar
      keep_fighting
    end
  end
end
```

We can use normal setters and getters

```
class Animal
  def enemy=(enemy)
    @enemy = enemy
  end
  def enemy
    return @enemy
  end
end
```



Access Levels



Usage

```
Cat = Animal.new
```

```
cat.enemy=(dog) #or  
cat.enemy = dog
```

```
cat.enemy # => dog
```

But there is a shortcut for the definition

```
class Animal  
  #same as defining basic setter and getter  
  attribute_accessor :enemy  
end
```



Access Levels



Shortcuts for all access types

```
class Animal
  #set and get
  attribute_accessor :enemy

  #get only
  attribute_reader :eye_color

  #set only
  attribute_writer :see
end
```



Uniform Access Principle



There is only one way to access instance variables, which is via accessor methods

- If you just need set/get then use `attribute_accessor`
- If you need more logic create your own methods
- Your interface remains the same. You never need to refactor client code to reflect changing from



Inheritance



Classes can inherit from other classes

- Single inheritance, you cannot have multiple ancestors for the same class
- By default all classes inherit from Object (if no parent is defined)

```
class Animal # Animal inherits from Object
  # ...
end
Animal.superclass # => Object

class Cat < Animal # Cat inherits from Animal
  # ...
end
Cat.superclass # => Animal
```



Polymorphism



- Classes can redefine methods from super classes
- Classes refer to methods in parent classes via super

```
class Animal
  def initialize(name)
    print name
  end
end

class Cat < Animal
  def initialize(name)
    print "cat's name is "
    super
  end
end
```

```
Cat.new('Tom') #=> prints: cat's name is Tom
```



Mixins



Allow us to functionality to classes

They are not interfaces, rather actual implementations

```
module Behavior
  def bite
  end
end
class Animal
  include Behavior
end

cat = Animal.new (cat can bite now)
```

- You can add as many modules to your class as you like.
- Modules can refer to methods of the class they are included in.



The Enumerable module

- One of the mostly used modules
- Adds enumeration methods like each, detect and select to classes with traversable elements
- A class needs only to implement an “each” method

```
class Array
  include Enumerable
  def each
  end
end
class Hash
  include Enumerable
  def each
  end
end
```



Methods



Methods



In Ruby there are several ways to call methods

- The dot notation
- The “send” method
- Calling “call” on a method object

```
class Animal
  def eat(food)
  end
end

cat = Animal.new

cat.eat(mouse)

cat.send('eat', mouse)

eating = cat.method('eat')

eating.call(mouse)
```



Methods

Methods can accept a variable length of arguments

```
Class Animal
  def eat(*food)
    # when variables are recieved,
    # they are stored in an array named food
  end
end

cat = Animal.new

cat.eat(mouse, fish, shrimp) # or
cat.send("eat", mouse, fish, shrimp)
```



Expanding arrays in method calls

Arrays can be expanded to match expected arguments

```
def four(a,b,c,d)
  puts "#{a}, #{b}, #{c}, #{d}"
end

four(1,2,3,4) # => 1,2,3,4
four(1,2,['a','b']) # => 1,2,a,b
four(*(5..8).to_a) #=> 5,6,7,8
```



Methods can have default argument values

```
def tool_for(lang="ruby", tool="irb")
  puts "#{tool} is a tool for #{lang}"
end

tool_for # irb is a tool for ruby
tool_for 'jruby' # irb is a tool for jruby
tool_for 'jruby', 'jconsole'
#irb is a tool for jconsole
```



Collecting hash arguments

- A simple way to achieve named arguments
- Simply pass a hash (without the braces)

```
def to_xml(object, options)
end

to_xml(chart, { :name=>'one', :color=>'blue' })

# or

to_xml(chart, :name=>'one', :color=>'blue')

# or

to_xml chart, { :name=>'one', :color=>'blue' }
```



Return values

- Methods return the last executed expression value
- Unless return is provided explicitly

```
def add(a, b)
  a + b # this is equal to "return a + b"
end

def max(a,b)
  return a if a > b # explicit return
  b
end
```



Return values

- Methods can return multiple values
- Ruby has mass assignments

```
def min_max(a, b)
  return a,b if a < b
  b, a
end
```

```
min, max = min_max(7,2)
# min now equals 2
# max now equals 7
```

```
def swap(a, b)
  b, a # similar to saying: a, b = b, a
end
```



method_missing

- This method acts as a trap for undefined methods
- Used to create methods on the fly!

```
class Recorder
  @@calls = []
  def method_missing(name, *args)
    @@calls << [name, args]
  end
  def play_back(obj)
    @@calls.each do |call|
      obj.send(call[0], *call[1])
    end
  end
end
```



method_missing (contd.)

```
Recorder = Recorder.new

# send messages to the recorder, nothing seems
# to happen

recorder.downcase!

recorder.capitalize!

recorder.playback('camelCase')
# returns Camelcase

# stored calls were replayed on the string
```



Blocks



What are blocks any way?

- Blocks are code snippets associated with method calls
- Blocks are closures (hence they are cool)
- Blocks are some sort of lambdas (only partially true)
- Ruby does have real lambdas anyway
- But what are lambdas?
 - Easy, lambdas are “anonymous functions”
 - Satisfied?



Blocks



Syntax

Blocks appear after method call, either surrounded by `do..end` or `{..}`

```
17.times do
  puts "i will not be naughty"
end
```

or

```
17.times { puts "i will not be naughty" }
```

They also accept arguments

```
open(path) do |file| # file is an argument
  # do something with file
end
```



Blocks



How are they executed?

The associated method should use the “yield” keyword

```
def iterate(collection)
  for item in collection
    yield item
  end
end

# the caller can associate a block
iterate(books) { |book| book.read }

# a more safe yield can look like this:
  yield item if block_given?
# only attempt to yield if there is a block
```



Blocks and Iterators



Enumerable (again)

Most methods in enumerable yield control to associated blocks (visitor pattern)

Classes representing collections should only provide an “each” method that knows how to iterate over the collection items

Code manipulating the items does not care about how to iterate. The collection hides this from it



Blocks and Iterators



Examples

Print array elements

```
[1,2,3,4].each{|element| puts element}  
# prints 1234
```

Select a group of elements

```
[1,2,3,4].select{|element| element > 2}  
# returns [3,4]
```

Or you can change the data structure

```
[s1,s2,s3].collect{|student| student.name}  
# returns ["anwar", "ashraf", "wael"]
```



Common idioms

Auto close for file operations

```
File.open(path) do |file|  
  file.each do |line|  
    # do something with line  
  end  
end
```

```
# The file will be closed after the block is  
# run. Less maintenance on your side
```

Similar approaches are done with sockets and other streams. Same with transactional operations.



Common idioms

Auto commit for transactions

```
# common Rails code
user.transaction do
  # some transactional operations
  # that we need to make sure all happen
end

# code running in the transaction body will be
# automatically committed upon success and
# rolled back in case of failure
```



Blocks



Blocks can be passed around

They need to be converted to Proc objects first

```
putter = Proc.new{|x|puts x} # returns a proc  
putter.call(3) # prints 3
```

```
# or we can use the shorter  
putter = proc{|x|puts x}
```

```
# or the nicer looking  
putter = lambda{|x|puts x}
```



Blocks



Blocks are closures

They retain their lexical scope

```
def add(n)
  lambda{|x|x + n}
end

add3 = add(3)

add3.call(5) # 8

add7 = add(7)
add7.call(2) # 9
```



Reflection



Reflection



Definition

It means the ability to lookup the structure of the program in run time

In Ruby, it comes in three flavors

- Object reflection
- Class reflection
- System reflection



Reflection



Object reflection

You can check the methods of an object

```
[1,2,3].methods.select{|m|m=="send"}  
# returns ["send"]
```

Or its class (or ancestor class)

```
32.class # Fixnum  
32.kind_of? Numeric # true  
32.instance_of? Fixnum # true
```

Or you can check if it has a certain method

```
32.respond_to? '+' # true  
12.respond_to? 'length' # false
```



Reflection



Class reflection

You can check a class' superclass

```
Cat.superclass # Animal  
Animal.superclass # Object
```

Or you can get the whole family history

```
Cat.ancestors  
# returns parent classes and included modules  
# [Animal, Behavior, Object, Kernel]
```

You can ask the class for methods, constants, etc

```
Cat.public_instance_methods  
Cat.singleton_methods # class (static) methods  
Cat.constants
```



Reflection



System reflection

You can ask the Ruby environment what objects are present in the system!.

```
ObjectSpace.each_object do |obj|  
  # iterate over all the objects in the system  
end
```

One caveat though, Fixnum, true, false and nil objects are not traversed via `ObjectSpace::each_object`



System Hooks



Ruby provides various callbacks for certain events

- We have met method_missing
- There are others:

```
method_added
# called whenever a method is added to a class
# or a module

inherited
# called on a class whenever it is subclassed

included
# called on a module whenever it is included
```



System Hooks



You can create your own

- Via `alias_method` (code copied from the pick axe book)

```
$object_ids = []
class Class # re-open the class Class
  alias_method :old_new, :new
  #old_new is now a 'copy' of :new
  def new(*args) # redefine new
    result = old_new(*args)
    $object_ids << result.object_id
    result
  end
end

# now we will have a collection of all object
# ids that get created in the system
```



MetaProgramming



Definition

Metaprogramming is the ability to program your program!

- In Ruby class definition bodies are actually executable code!
- We have seen them already, but tricked because of Ruby syntax
- Remember `attribute_accessor`?



What about `attribute_accessor`?

- It is actually a method call
- Executed right in the class definition body
- It alters the class definition as it runs
- It is called a class instance function (huh?)
- Here's a more familiar rendering

```
class Predator
  attribute_accessor(:prey)
  # this method call will alter the structure
  # of this class instance
end
```



Enough!



Don't Forget To Check





why's (poignant) guide to Ruby



By _why the lucky stiff



Thank You

Mohammad A. Ali
CTO, eSpace

