

# Ruby 101

Dmitry Ratnikov

October 22, 2008

## Outline

### Basics

### Object orientation

- Classes

- Modules

### The cool bits

- General principles

- Extending ruby

- Dynamic method generation

# What is ruby?

## Wikipedia says...

Ruby is a dynamic, reflective, general purpose object-oriented programming language that combines syntax inspired by Perl with Smalltalk-like features.

# What is ruby?

## Wikipedia says...

Ruby is a **dynamic**, **reflective**, general purpose **object-oriented** programming language that combines syntax inspired by Perl with Smalltalk-like features.

# What is ruby?

## Wikipedia says...

Ruby is a **dynamic**, reflective, general purpose object-oriented programming language that combines syntax inspired by Perl with Smalltalk-like features.

# What is ruby?

## Wikipedia says...

Ruby is a dynamic, **reflective**, general purpose object-oriented programming language that combines syntax inspired by Perl with Smalltalk-like features.

# What is ruby?

## Wikipedia says...

Ruby is a dynamic, reflective, general purpose **object-oriented** programming language that combines syntax inspired by Perl with Smalltalk-like features.

# What is ruby?

## Wikipedia says...

Ruby is a dynamic, reflective, general purpose object-oriented programming language that combines syntax inspired by **Perl** with **Smalltalk**-like features.



# Variables/Arrays/Hashes

## ► Variables

```
pickaxe_book = "Programming Ruby"  
cs_bible = "Art of Computer Programming"  
js_book = "Javascript: The Good Parts"
```

## ► Arrays

```
available_books = [ pickaxe_book, cs_bible ]
```

## ► Hashes

```
library = {  
  :available => available_books,  
  :checked_out => [ js_book ]  
}
```

# Variables/Arrays/Hashes

## ▶ Variables

```
pickaxe_book = "Programming Ruby"  
cs_bible = "Art of Computer Programming"  
js_book = "Javascript: The Good Parts"
```

## ▶ Arrays

```
available_books = [ pickaxe_book, cs_bible ]
```

## ▶ Hashes

```
library = {  
  :available => available_books,  
  :checked_out => [ js_book ]  
}
```

# Variables/Arrays/Hashes

## ► Variables

```
pickaxe_book = "Programming Ruby"  
cs_bible = "Art of Computer Programming"  
js_book = "Javascript: The Good Parts"
```

## ► Arrays

```
available_books = [ pickaxe_book, cs_bible ]
```

## ► Hashes

```
library = {  
  :available => available_books,  
  :checked_out => [ js_book ]  
}
```

# Defining Methods

## Making life easier

```
def available?(library, book_name)
  library[:available].include?(book_name)
end
```

- ▶ Method body is enclosed by `def` and `end`.  
Parameters are a list of variable names.
- ▶ Method name must be lower case letters and `_`.  
It may be suffixed by `?`, `=` or `!`.
- ▶ Last line is returned by the method.

# Defining Methods

## Making life easier

```
def available?(library, book_name)
  library[:available].include?(book_name)
end
```

- ▶ Method body is enclosed by `def` and `end`.  
Parameters are a list of variable names.
- ▶ Method name must be lower case letters and `_`.  
It may be suffixed by `?`, `=` or `!`.
- ▶ Last line is returned by the method.

# Defining Methods

## Making life easier

```
def available?(library, book_name)
  library[:available].include?(book_name)
end
```

- ▶ Method body is enclosed by `def` and `end`.  
Parameters are a list of variable names.
- ▶ Method name must be lower case letters and `_`.  
It may be suffixed by `?`, `=` or `!`.
- ▶ Last line is returned by the method.

# Defining Methods

## Making life easier

```
def available?(library, book_name)
  library[:available].include?(book_name)
end
```

- ▶ Method body is enclosed by `def` and `end`.  
Parameters are a list of variable names.
- ▶ Method name must be lower case letters and `_`.  
It may be suffixed by `?`, `=` or `!`.
- ▶ Last line is returned by the method.

# Conditionals

## if/else syntax

```
if available?(library, "Art of War")  
  puts "Sun Tzu's Art of War is available."  
else  
  puts "Art of War is not available."  
  puts "Try later..."  
end
```

## Can be inlined:

```
puts "yay" unless boo?
```



# Loops

## For/While loops:

```
file = File.open("checked_out_backup.txt")  
while (book = file.gets)  
  library[:checked_out] << book  
end
```

```
str = ""  
for i in 0..(library[:checked_out].size) do  
  str += "#{library[:checked_out][i]} "  
end  
puts "Checked out books: #{str}"
```

# Blocks/iterators

Same as for, but easier

```
str = ""  
library[:checked_out].each do |book|  
  str += "#{book} "  
end  
puts "Checked out books: #{str}"
```

- ▶ Invoke method `each` on the checked out books array.
- ▶ Declare a block that takes one parameter as `book`.
- ▶ Specify the body of the block to append the book to the `str`.

# Blocks/iterators

Same as for, but easier

```
str = ""  
library[:checked_out].each do |book|  
  str += "#{book} "  
end  
puts "Checked out books: #{str}"
```

- ▶ Invoke method `each` on the checked out books array.
- ▶ Declare a block that takes one parameter as `book`.
- ▶ Specify the body of the block to append the book to the `str`.

# Blocks/iterators

Same as for, but easier

```
str = ""  
library[:checked_out].each do |book|  
  str += "#{book} "  
end  
puts "Checked out books: #{str}"
```

- ▶ Invoke method `each` on the checked out books array.
- ▶ Declare a block that takes one parameter as `book`.
- ▶ Specify the body of the block to append the book to the `str`.

# Blocks/iterators

Same as for, but easier

```
str = ""  
library[:checked_out].each do |book|  
  str += "#{book} "  
end  
puts "Checked out books: #{str}"
```

- ▶ Invoke method `each` on the checked out books array.
- ▶ Declare a block that takes one parameter as `book`.
- ▶ Specify the body of the block to append the book to the `str`.

# Accumulator Style

## Even easier

```
arr = library[:checked_out]
str = arr.inject("") do |acc, item|
  "#{acc} #{item}"
end
puts "Checked out books: #{str}"
```

- ▶ `inject`'s block takes accumulator and book parameters.
- ▶ Return of the block is passed in the next `acc`
- ▶ Last `acc` is returned by the `inject` which is assigned to `str`.

# Accumulator Style

Even easier

```
arr = library[:checked_out]
str = arr.inject("") do |acc, item|
  "#{acc} #{item}"
end
puts "Checked out books: #{str}"
```

- ▶ inject's block takes accumulator and book parameters.
- ▶ Return of the block is passed in the next acc
- ▶ Last acc is returned by the inject which is assigned to str.

# Accumulator Style

Even easier

```
arr = library[:checked_out]
str = arr.inject("") do |acc, item|
  "#{acc} #{item}"
end
puts "Checked out books: #{str}"
```

- ▶ `inject`'s block takes accumulator and book parameters.
- ▶ Return of the block is passed in the next acc
- ▶ Last acc is returned by the `inject` which is assigned to `str`.



# Accumulator Style

Even easier

```
arr = library[:checked_out]
str = arr.inject("") do |acc, item|
  "#{acc} #{item}"
end
puts "Checked out books: #{str}"
```

- ▶ `inject`'s block takes accumulator and book parameters.
- ▶ Return of the block is passed in the next `acc`
- ▶ Last `acc` is returned by the `inject` which is assigned to `str`.

# Declaring a Method With Block

## Sample implementation of inject

```
def available_inject(library, init, &block)
  raise "Block missing" unless block_given?
  arr = library[:checked_out]
  acc = init
  arr.each { |item| acc = yield(acc, item) }
  acc
end
```

# Declaring a Method With Block

## Sample implementation of inject

```
def available_inject(library, init, &block)
  raise "Block missing" unless block_given?
  arr = library[:checked_out]
  acc = init
  arr.each { |item| acc = yield(acc, item) }
  acc
end
```

- ▶ Block is passed using &
- ▶ `block_given?` returns whether method was provided a block.

# Declaring a Method With Block

## Sample implementation of inject

```
def available_inject(library, init, &block)
  raise "Block missing" unless block_given?
  arr = library[:checked_out]
  acc = init
  arr.each { |item| acc = yield(acc, item) }
  acc
end
```

- ▶ `yield` yields control to the provided block with specified parameters.

# Declaring a class

## Declaration syntax:

- ▶ Classes are declared by keyword `class`
- ▶ Instance variables are specified by prepending '@' to a variable name (e.g. `@foo`)
- ▶ Class variables are specified by prepending '@@' (e.g. `@@bar`)

# Sample declaration

```
class Book
  def name; @title end
  def name=(new_title)
    @title = new_title; @title
  end
  attr_accessor :author, :isbn
end
```

- ▶ Class names must be capitalized.
- ▶ Creates author and isbn accessors.

# Sample declaration

```
class Book
  def name; @title end
  def name=(new_title)
    @title = new_title; @title
  end
  attr_accessor :author, :isbn
end
```

- ▶ Class names must be capitalized.
- ▶ Creates `author` and `isbn` accessors.

# Sample declaration

```
class Book
  def name; @title end
  def name=(new_title)
    @title = new_title; @title
  end
  attr_accessor :author, :isbn
end
```

- ▶ Class names must be capitalized.
- ▶ Creates author and isbn accessors.



# Inheritance

## Example (Inheriting classes)

```
class Game < Book
  attr_accessor :platform
end
```

- ▶ `Game` now inherits all instance methods from `Book` class.
- ▶ And has an additional `platform` accessor.
- ▶ But... inherits `isbn` which games do not really have.

# Inheritance

## Example (Inheriting classes)

```
class Game < Book
  attr_accessor :platform
end
```

- ▶ Game now inherits all instance methods from Book class.
- ▶ And has an additional platform accessor.
- ▶ But... inherits isbn which games do not really have.

# Inheritance

## Example (Inheriting classes)

```
class Game < Book
  attr_accessor :platform
end
```

- ▶ Game now inherits all instance methods from Book class.
- ▶ And has an additional platform accessor.
- ▶ But... inherits isbn which games do not really have.

# Inheritance

## Example (Inheriting classes)

```
class Game < Book
  attr_accessor :platform
end
```

- ▶ Game now inherits all instance methods from Book class.
- ▶ And has an additional platform accessor.
- ▶ But... inherits isbn which games do not really have.

# How do we fix it?

What we really want is:

Encapsulate the 'has name' functionality that allows classes to have a name and author and then include it into Book and Game classes.

# How do we fix it?

What we really want is:

Encapsulate the 'has name' functionality that allows classes to have a name and author and then include it into Book and Game classes.

Solution:  
Modules

# What's a module

## Definition (Module)

A Module is a collection of methods and constants.

Game plan:

1. Create `HasName` module that gives name functionality.
2. Weave it into `Book` and `Game` classes.

# What's a module

## Definition (Module)

A Module is a collection of methods and constants.

## Game plan:

1. Create `HasName` module that gives `name` functionality.
2. Weave it into `Book` and `Game` classes.



# What's a module

## Definition (Module)

A Module is a collection of methods and constants.

## Game plan:

1. Create HasName module that gives name functionality.
2. Weave it into Book and Game classes.

# Declaring a module

```
module HasName  
  attr_accessor :name, :author  
end
```

- ▶ Same rules as for classes: must be capitalized.

# Declaring a module

```
module HasName  
  attr_accessor :name, :author  
end
```

- ▶ Same rules as for classes: must be capitalized.

# Using a module

```
class Book
  include HasName
  attr_accessor :isbn
end
class Game
  include HasName
  attr_accessor :platform
end
```

- ▶ Book and Game are now independent and shared functionality is abstracted neatly in the HasName module.

# Using a module

```
class Book
  include HasName
  attr_accessor :isbn
end
class Game
  include HasName
  attr_accessor :platform
end
```

- ▶ Book and Game are now independent and shared functionality is abstracted neatly in the HasName module.

# Principle #1 (DRY)

“Don’t Repeat Yourself” principle:

If you have to do something more than once, abstract it away.

# Principle #1 (DRY)

“Don’t Repeat Yourself” principle:

If you have to do something more than once, abstract it away.

Why?

# Principle #1 (DRY)

“Don’t Repeat Yourself” principle:

If you have to do something more than once, abstract it away.

Why?

- ▶ Code duplication means you wrote it at least twice.



# Principle #1 (DRY)

“Don’t Repeat Yourself” principle:

If you have to do something more than once, abstract it away.

Why?

- ▶ Code duplication means you wrote it at least twice.
- ▶ Code duplication reduces clarity.

# Principle #1 (DRY)

“Don’t Repeat Yourself” principle:

If you have to do something more than once, abstract it away.

Why?

- ▶ Code duplication means you wrote it at least twice.
- ▶ Code duplication reduces clarity.
- ▶ Code duplication is much harder to keep in sync.

# Principle #1 (DRY)

“Don’t Repeat Yourself” principle:

If you have to do something more than once, abstract it away.

Why?

- ▶ Code duplication means you wrote it at least twice.
- ▶ Code duplication reduces clarity.
- ▶ Code duplication is much harder to keep in sync.

Exceptions:

# Principle #1 (DRY)

“Don’t Repeat Yourself” principle:

If you have to do something more than once, abstract it away.

Why?

- ▶ Code duplication means you wrote it at least twice.
- ▶ Code duplication reduces clarity.
- ▶ Code duplication is much harder to keep in sync.

Exceptions:

- ▶ Code duplication is sometimes okay in tests.

# Principle #1 (DRY)

“Don’t Repeat Yourself” principle:

If you have to do something more than once, abstract it away.

Why?

- ▶ Code duplication means you wrote it at least twice.
- ▶ Code duplication reduces clarity.
- ▶ Code duplication is much harder to keep in sync.

Exceptions:

- ▶ Code duplication is sometimes okay in tests.
- ▶ Code duplication is sometimes okay in documentation.

## Principle #2 (YAGNI)

“You Ain’t Gonna Need It” principle:

Always implement things when you actually need them,  
never when you just foresee that you will need them.

## Principle #2 (YAGNI)

“You Ain’t Gonna Need It” principle:

Always implement things when you actually need them,  
never when you just foresee that you will need them.

Why?

## Principle #2 (YAGNI)

“You Ain’t Gonna Need It” principle:

Always implement things when you actually need them,  
never when you just foresee that you will need them.

Why?

- ▶ Time is better spent on something you actually need



## Principle #2 (YAGNI)

“You Ain’t Gonna Need It” principle:

Always implement things when you actually need them,  
never when you just foresee that you will need them.

Why?

- ▶ Time is better spent on something you actually need
- ▶ What you predict will happen usually is not what really happens.

## Principle #2 (YAGNI)

“You Ain’t Gonna Need It” principle:

Always implement things when you actually need them,  
never when you just foresee that you will need them.

Why?

- ▶ Time is better spent on something you actually need
- ▶ What you predict will happen usually is not what really happens.
- ▶ By the time you will need it, you will know the problem better.

## Principle #2 (YAGNI)

“You Ain’t Gonna Need It” principle:

Always implement things when you actually need them,  
never when you just foresee that you will need them.

Why?

- ▶ Time is better spent on something you actually need
- ▶ What you predict will happen usually is not what really happens.
- ▶ By the time you will need it, you will know the problem better.

But what about DRY?

## Principle #2 (YAGNI)

“You Ain’t Gonna Need It” principle:

Always implement things when you actually need them,  
never when you just foresee that you will need them.

Why?

- ▶ Time is better spent on something you actually need
- ▶ What you predict will happen usually is not what really happens.
- ▶ By the time you will need it, you will know the problem better.

But what about DRY?

- ▶ DRY things up when you actually repeat yourself,  
not when you think you may repeat yourself

## Principle #3 (duck typing)

Duck typing principle:

If it walks like a duck and quacks like a duck, it is a duck.

# Principle #3 (duck typing)

Duck typing principle:

If it walks like a duck and quacks like a duck, it is a duck.

In practice that means

- ▶ What's important is what an object does, not what it is.

# Principle #3 (duck typing)

## Duck typing principle:

If it walks like a duck and quacks like a duck, it is a duck.

## In practice that means

- ▶ What's important is what an object does, not what it is.
- ▶ In duck-typed languages, interfaces are implicitly specified by defined methods.

## Principle #3 (duck typing, cont.)

```
class Library
  attr_accessor :books
  def catalog
    books.map { |b| b.name }.join ", "
  end
end
```

Only things Library cares about:

- ▶ books responds to map.
- ▶ Each element of books responds to name.
- ▶ Whatever b.name returns must be concatenatable by join.



## Principle #3 (duck typing, cont.)

```
class Library
  attr_accessor :books
  def catalog
    books.map { |b| b.name }.join ", "
  end
end
```

Only things Library cares about:

- ▶ books responds to map.
- ▶ Each element of books responds to name.
- ▶ Whatever b.name returns must be concatenatable by join.

## Principle #3 (duck typing, cont.)

```
class Library
  attr_accessor :books
  def catalog
    books.map { |b| b.name }.join ", "
  end
end
```

Only things `Library` cares about:

- ▶ `books` responds to `map`.
- ▶ Each element of `books` responds to `name`.
- ▶ Whatever `b.name` returns must be concatenatable by `join`.

## Principle #3 (duck typing, cont.)

```
class Library
  attr_accessor :books
  def catalog
    books.map { |b| b.name }.join ", "
  end
end
```

Only things `Library` cares about:

- ▶ `books` responds to `map`.
- ▶ Each element of `books` responds to `name`.
- ▶ Whatever `b.name` returns must be concatenatable by `join`.

# Adding Fixnum#inject

Suppose we want to be able to do:

```
sorted_profiles = 50.inject([]) do |acc|
  acc + [Profile.random!]
end.sort_by { |p| p.name }
```

But...

- ▶ But ruby doesn't have Fixnum#inject

# Adding Fixnum#inject

Suppose we want to be able to do:

```
sorted_profiles = 50.inject([]) do |acc|
  acc + [Profile.random!]
end.sort_by { |p| p.name }
```

But...

- ▶ But ruby doesn't have Fixnum#inject

## Adding Fixnum#inject (cont.)

No problem:

```
class Fixnum
  def inject(init = nil, &block)
    raise "Block missing" unless block_given?
    acc = init
    for i in 0..(self-1) do
      init = yield(init, i)
    end
  end
end
```

# DRYing things up

```
module JavascriptHelper
  def author_js author
    "var author = "+
    "constructAuthor({ name: #{author.name}})"
  end
  def book_js book
    "var book = constructBook({ name: #{book.name}})"
  end
  def author_js_tag author
    script_tag author_js(author)
  end
  def book_js_tag book; script_tag book_js(book) end
end
```

# DRYing things up (cont.)

Before:

```
def author_js_tag author
  script_tag author_js(author)
end
def book_js_tag book; script_tag book_js(book) end
```

- ▶ Both methods have a very similar structure.
- ▶ Both methods do the same things with their arguments.



# DRYing things up (cont.)

Before:

```
def author_js_tag author
  script_tag author_js(author)
end
def book_js_tag book; script_tag book_js(book) end
```

- ▶ Both methods have a very similar structure.
- ▶ Both methods do the same things with their arguments.

# DRYing things up (cont.)

Before:

```
def author_js_tag author
  script_tag author_js(author)
end
def book_js_tag book; script_tag book_js(book) end
```

- ▶ Both methods have a very similar structure.
- ▶ Both methods do the same things with their arguments.

# DRYing things up (cont.)

After refactoring:

```
%w(book_js author_js).each do |js_method|
  define_method "#{js_method}_tag" do |item|
    script_tag send(js_method, item)
  end
end
```

- What about make all methods ending in \_js have a \_tag counterpart?

# DRYing things up (cont.)

After refactoring:

```
%w(book_js author_js).each do |js_method|
  define_method "#{js_method}_tag" do |item|
    script_tag send(js_method, item)
  end
end
```

- What about make all methods ending in \_js have a \_tag counterpart?

# DRYing things up (cont.)

After second refactoring:

```
instance_methods.select do |m|
  m =~ /^_js$/
end.each do |js_method|
  define_method "#{js_method}_tag" do |*args|
    script_tag send(js_method, *args)
  end
end
```

# Interfaces

## Wikipedia says:

Interface generally refers to an abstraction that an entity provides of itself to the outside.

- ▶ In java, interface type defines how components may interact.
- ▶ In ruby, how components interact defines what interface they have.

(That's called duck typing)

# Interfaces

Wikipedia says:

Interface generally refers to an abstraction that an entity provides of itself to the outside.

- ▶ In java, interface type defines how components may interact.
- ▶ In ruby, how components interact defines what interface they have.

(That's called duck typing)

# Interfaces

Wikipedia says:

Interface generally refers to an abstraction that an entity provides of itself to the outside.

- ▶ In java, interface type defines how components may interact.
- ▶ In ruby, how components interact defines what interface they have.

(That's called duck typing)



# Interfaces

## Wikipedia says:

Interface generally refers to an abstraction that an entity provides of itself to the outside.

- ▶ In java, interface type defines how components may interact.
- ▶ In ruby, how components interact defines what interface they have.

(That's called duck typing)