

RUBY IS FROM MARS, Functional Languages Are from Venus

Integrating Ruby with Erlang, Scala & F#

ANDREA O. K. WRIGHT
Chariot Solutions
aok@chariotsolutions.com

The source for all of the examples can be found at: <https://github.com/A-OK/RubyIsFromMars>

Conventions Used on These Slides

Console sessions are generally represented with a black background.

Console prompts and output are colored yellow.

Text entered into the console is colored white, red or green.

Red and green are used to highlight portions of entered code.

Here is a sample irb session:

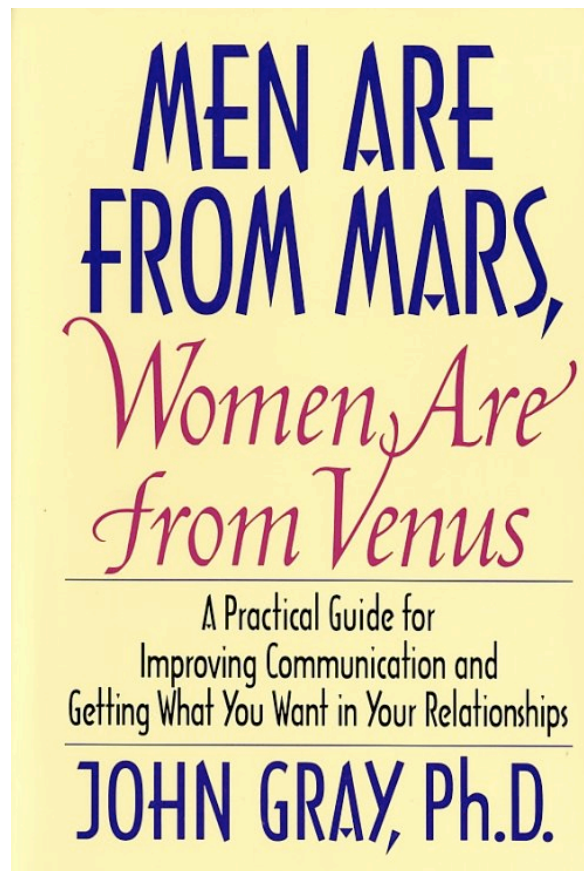
```
> phrases = ["Ruby is from Mars", "Functional Languages are from Venus"]  
=> ["Ruby is from Mars", "Functional Languages are from Venus"]  
> phrases.join(", ")  
=> "Ruby is from Mars, Functional Languages are from Venus"
```

About the Sample Code on the Slides...

Code examples on the slides don't always show all the `require` or `include` statements needed in order for the code to run.

For the complete code samples used in this presentation, as well as complete details about how they were run and which versions of projects were used:

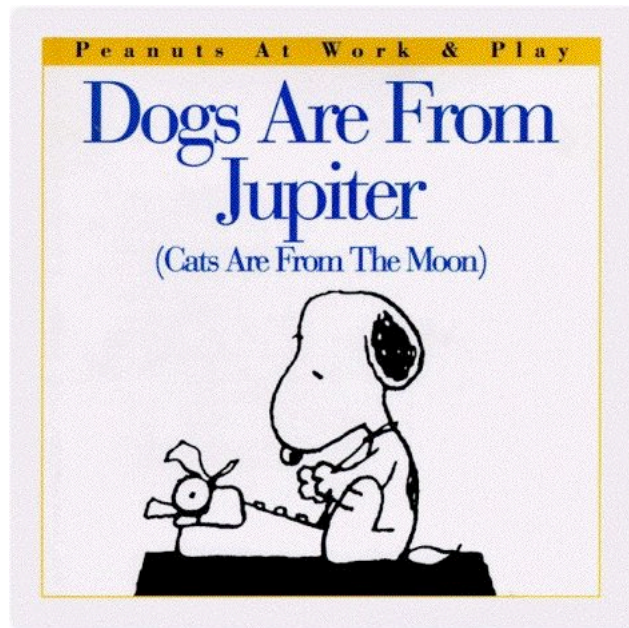
<https://github.com/A-OK/RubyIsFromMars>



The title of this talk is a reference to the classic book that asserts that men and women handle problems so differently, it's as if they come from different planets where different languages are spoken.

But I'm not going to be talking about programming languages in terms of gender. I chose to link Ruby with Mars because Mars is the red planet.

The book title has taken on a life of its own...



...and is now used to refer to all manner of polar opposites.

**DEVELOPERS ARE FROM MARS,
Ops People are from Venus**

**ENGINEERS ARE FROM MARS,
Marketers are from Venus**

**CIOs ARE FROM MARS
CEOs are from Venus**

These are titles of blog posts about how people with vastly different approaches to solving problems could potentially work together well if they could just learn to communicate with each other.

FUNCTIONAL

IMPERATIVE

Functional and imperative programming are two vastly different approaches to solving problems.

Functional programming is programming without side-effects.

It's programming with functions that you can count on to return the same thing whenever you pass in the same arguments because they don't depend on anything other than the arguments they are passed.

Side-effect free functions don't need to access anything that is subject to change, like the filesystem, the GUI, mutable data structures or mutable variables -- and they likewise don't modify anything that exists outside of their function scopes.

For this talk, I'm going to go ahead and define a functional language as a language that's partial to functional programming.



From Programming Erlang by Joe Armstrong

Let's see what can happen when you're allowed to change a variable.

Let's define a variable X as follows:

```
1> x = 23.  
23
```

Erlang meets that criteria. Here's a passage about Erlang's view of mutable variables from Erlang creator Joe Armstrong's [Programming Erlang](#) book.

He writes: "Let's see what can happen when you're allowed to change a variable," -- and he shows a console session where x is bound to 23.



From Programming Erlang by Joe Armstrong

Let's see what can happen when you're allowed to change a variable.

Let's define a variable X as follows:

```
1> X = 23.  
23
```

Now we can use X in computations:

```
2> Y = 4 * X + 3.  
95
```

He uses x in an equation...



From Programming Erlang by Joe Armstrong

Let's see what can happen when you're allowed to change a variable.

Let's define a variable X as follows:

```
1> x = 23.  
23
```

Now we can use X in computations:

```
2> y = 4 * x + 3.  
95
```

Now suppose we could change the value of X :

```
3> x = 19.
```

(HORRORS)

... and then writes: "Now suppose we could change the value of x (horrors)"

The emphasis is mine, but he does actually use the word "horrors" in his text where he shows a console session with an attempt to set x to 19.



From Programming Erlang by Joe Armstrong

Let's see what can happen when you're allowed to change a variable.

Let's define a variable X as follows:

```
1> x = 23.  
23
```

Now we can use X in computations:

```
2> y = 4 * x + 3.  
95
```

Now suppose we could change the value of X:

(HORRORS)

```
3> x = 19.  
in process <0.31.0> with exit  
value:  
{badmatch,19},[{erl_eval,expr,  
3}]]
```

And then he shows the error that appears in the console when you try to change the value of a bound variable.



From Programming Erlang by Joe Armstrong

Let's see what can happen when you're allowed to change a variable.

Let's define a variable X as follows:

```
1> x = 23.  
23
```

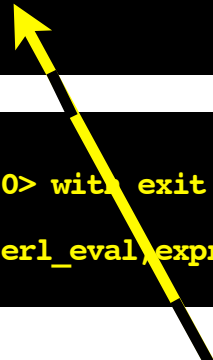
Now we can use X in computations:

```
2> y = 4 * x + 3.  
95
```

Now suppose we could change the value of X :

(HORRORS)

```
3> x = 19.  
in process <0.31.0> with exit  
value:  
{badmatch, 19}, [{erl_eval, expr,  
3}]}
```



But just suppose we could do this; then the value of Y would be wrong in the sense that we can no longer interpret statement **2** as an equation.

He collects himself and goes on to point out that the statement in line 2 could no longer be interpreted as an equation if the value of x was allowed to change.



From Programming Erlang by Joe Armstrong

Let's see what can happen when you're allowed to change a variable.

Let's define a variable X as follows:

```
1> X = 23.  
23
```

Now we can use X in computations:

```
2> Y = 4 * X + 3.  
95
```

Now suppose we could change the value of X:

(HORRORS)

```
3> X = 19.  
in process <0.31.0> with exit  
value:  
{badmatch, 19},  
[{erl_eval, expr, 3}]}
```

But just suppose we could do this; then the value of Y would be wrong in the sense that we can no longer interpret statement **2** as an equation.

Notice that the error message says “bad match,” and not something like “Attempt to modify a non-modifiable values.”

This is because in Erlang, the equals sign (=) is the pattern matching operator, not the assignment operator.

The error is pointing out that 19 does not match 23.

Joe Armstrong describes the variable definition statement in line 1 as pattern matching with a simple pattern, and not as variable assignment. He explains that if there are unbound variables on the left-hand side of an equation, the pattern matching operator will take care of the variable binding to make the pattern on its left-hand side match the pattern on its right-hand side.



Equations and “=” in Erlang

```
1> 2 + 3 = 4 + 1  
5
```

The concept of using the equals sign the way it would be used in a mathematical equation is very important in Erlang: in math the equals sign signifies that the left-hand side matches the right-hand side.

Here's an example that shows the extent to which Ruby and Erlang are not on the same page with respect to equals sign. Ruby would choke on a statement like the one in this slide, but it's perfectly valid in Erlang.

Incidentally there *are* ways to mutate state in Erlang. For example, each process has a mutable process dictionary. But I think Erlang's functional bias is clear.



Immutable vals

```
scala> val x = 23
x: Int = 23
scala> x = 19
<console>:5: error: reassignment to val
      x=23
```


Mutable vars

```
scala> var y = 23
y: Int = 23
scala> y = 19
y: Int = 19
```

Now let's look at why think Scala is partial to functional programming.


This slide shows two ways to define variables in Scala. The `val` keyword is used to define immutable variables and the `var` keyword is for mutable variables. As you can see, changing the `val`, `x`, to 19 after initially setting it to 23, causes an error -- while there is no problem changing the value of the `var`, `y`.

The `vals` are considered superior in the realm of Scala.

In  **Scala**, we should prefer using `val` over `var` as much as possible since that promotes immutability and functional style.

— Venkat Subramaniam,
Programming Scala

In the Programming Scala, Venkat Subramaniam writes: “In Scala, we should prefer using `val` over `var` as much as possible since that promotes immutability and functional style.”

 **Scala** enables you to program imperatively, but as you get to know Scala better, you'll likely often find yourself programming in a more functional style.

— Bill Venners, Lex Spoon, Martin Odersky,
“First Steps to Scala”

This sentiment is echoed in an article on the web called “First Steps to Scala” that was co-authored by Scala creator Martin Odersky: “Scala enables you to program imperatively, but as you get to know Scala better, you'll likely often find yourself programming in a more functional style.”

F#

Immutable Variables

```
> let x = 23;;  
val it : int = 23  
> x = 19;;  
val it : bool = false  
> x;;  
val it : int = 23
```

The mutable Keyword

```
> let mutable y = 23;;  
val mutable y : int = 23  
> y <- 19;;  
val it : unit = ()  
> y;;  
val it : int = 19
```

In F#, variables are immutable by default. Although the F# console doesn't complain about the attempt to assign 19 to `x` after it was bound to 23, you can see that when `x` is evaluated in the console on the next line, it is still 23.

To create mutable variables, you need to use the `mutable` keyword, and you have to use the special left arrow operator (`<-`) to modify their values.

Incidentally in the F# console two semicolons (`;;`) are used to indicate that an expression is ready to be evaluated.

Many F# programmers use functional programming techniques first before turning to their imperative alternatives, and we encourage you to do the same...

-- Don Syme, Adam Granicz, Atoninio Cisternino,
Expert F#

This is from the book Expert F#, which was co-authored by F# creator Don Syme: “Many F# programmers use functional programming techniques first before turning to their imperative alternatives, and we encourage you to do the same...”

I have once dreamed of a such
language [immutable Ruby], and
had a conclusion that was not
Ruby...

— Yukihiro “Matz” Matsumoto



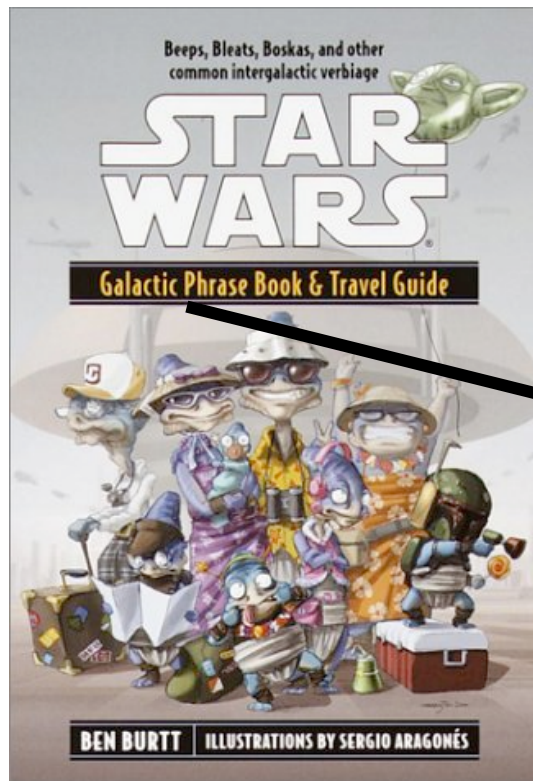
I do not believe Ruby can be called a functional language based on the my criteria.

Even though Matz was influenced by some functional concepts from the beginning, and even though many techniques associated with functional programming are built into Ruby, I don't think Ruby is a functional language at heart.

Here's Matz on the subject of immutable Ruby from a mailing list discussion about Reia, a language with Ruby-like syntax that runs on the Erlang VM. He posted: "I have once dreamed of such a language, and had a conclusion that was not Ruby..."

Fortunately, even if you have a use case that requires immutability, you don't necessarily need to use a functional language in lieu of Ruby. I'm going to talk about projects that enable you to develop polyglot systems with Ruby and functional languages.

(Hat tip to Tony Arcieri for highlighting this Matz quote in his presentation "Building Languages on Erlang (and an introduction to Reia)" which is downloadable from here: <http://www.erlang-factory.com/conference/SFBayAreaErlangFactory2009/speakers/TonyArcieri>).



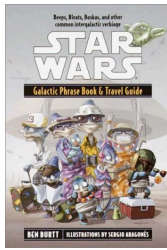
Galactic Phrase Book

For the most part, I'm going to focus on what I think of as "Phrase Book Level" interoperability.

There's a way in which communicating exclusively with phrases from a phrase book in a country where you don't speak the native language is not unlike incorporating a second programming language by making discreet library calls to a API implemented in that second language.

We'll also look at some deeper types of integration.

We won't just look at how to use the libraries I'm covering, we'll also walk through some of the lower-level code allows the inter-language communication to happen.



Phrase Book-Level Interoperability

Mars Library



Venus Libraries



Most of the examples come from my Mars and Venus micro-libraries that can calculate an age in Mars or Venus years.

The Mars library is implemented in Ruby -- and there are Erlang, Scala and F# implementations of the Venus library.



Rebar



(Ruby to Erlang Bridge and Runner)

Tom Preston-Werner

<http://github.com/mojombo/rebar>



RBridge/rulang



Toshiyuki Hirooka; Chuck Vose

<http://github.com/grockit/rulang>

We'll start by looking at Rebar and RBridge, two projects that make it easy to embed calls to Erlang functions in Ruby code.

The usage for Rebar and RBridge is almost identical. But as we will see, the implementations are different, particularly on the Erlang side.

Before I started researching this talk, I did not think of Erlang as a language that was good for constructing DSLs. But in the Rebar and RBridge source, I found two different ways to dynamically build up calls to arbitrary functions.



```
-module(venus).  
-export([venus_age/1]).  
  
venus_age(Age) ->  
    Age * (365.26/224.68).
```

Here's `venus_age`, the Erlang function I'm going to use in both my Rebar and RBridge examples. You supply an `Age`, and the function returns that `Age` in Venus years.

I've placed it in a module called `venus`.



```
-module(venus).  
-export([venus_age/1]).
```

```
venus_age(Age) ->  
    Age * (365.26/224.68).
```

Rebar 

```
1> rebar:start().
```

RBridge 

```
1> rulang:start_server(9900).
```

Both Rebar and RBridge come with Erlang-based servers that need to run in the background.

The Erlang console session on the left shows the command that starts Rebar's server, and the Erlang console session on the right shows the command that kicks off the RBridge server, which is called "rulang".



```
-module(venus).
-export([venus_age/1]).
```

```
venus_age(Age) ->
  Age * (365.26/224.68).
```

Rebar



```
1> rebar:start().
```

RBridge



```
1> rulang:start_server(9900).
```

Rebar



```
> require './client.rb'
> proxy = Rebar::Erlang.new(:venus,
                             '127.0.0.1',
                             5500)
> proxy.venus_age(17)
=> 27.6367
```

RBridge



```
> require 'rbridge'
> proxy = RBridge.new("venus",
                       "localhost",
                       9900)
> proxy.venus_age(17)
=> 27.6367277906356
```

This slide shows how easy Rebar and RBridge make it for a Ruby program to call to a function defined in an Erlang module.

The `irb` session on the right creates a proxy for the Erlang module by passing a symbol representing the name of the Erlang module, the Erlang server ip address and the Erlang server port to the `Rebar::Erlang` constructor.

Similarly, the `irb` session on the left creates a proxy for the Erlang module by passing the name of the Erlang module, the Erlang server ip address, and the Erlang server port to the constructor for the `RBridge` class.

As shown in the lines highlighted in red in both the Rebar and RBridge `irb` sessions, calling an Erlang function on the Erlang module linked to the proxy looks just like calling a Ruby method on the proxy. To call the Erlang function named `venus_age`, we just call `venus_age` on the proxy.

These sample `irb` sessions pass 17 to the `venus_age` function. Why did I chose to send 17? Because that is how old Ruby is in Earth years. These console sessions show that Ruby would be around 27 years old on Venus.

Rebar

RBridge

```
> proxy.venus_age(17)
```

method_missing

```
{"method": "venus:venus_age",  
  "params": [17],  
  "id": 0}
```

```
"venus:venus_age(17)."
```

Now I'm going to walk through some of the Ruby source for the Rebar and RBridge projects.

Both Rebar and RBridge leverage `method_missing` to transform the method call on the proxy into data that can be sent to their Erlang servers.

In `method_missing`, Rebar uses JSON to label a “method” (a string comprised of the module name, a colon and the function name) and “params” (in this case, an array containing the one parameter, 17).

In `method_missing`, RBridge creates a string using the correct Erlang syntax for calling a function (ie, the module name, a semi-colon, the function name, and the arguments enclosed in parentheses). Unless the function is local to a module or unless it is in the `erlang` module, it should be prefixed with the module name and a colon.

Rebar 

RBridge 

> proxy.venus_age(17)

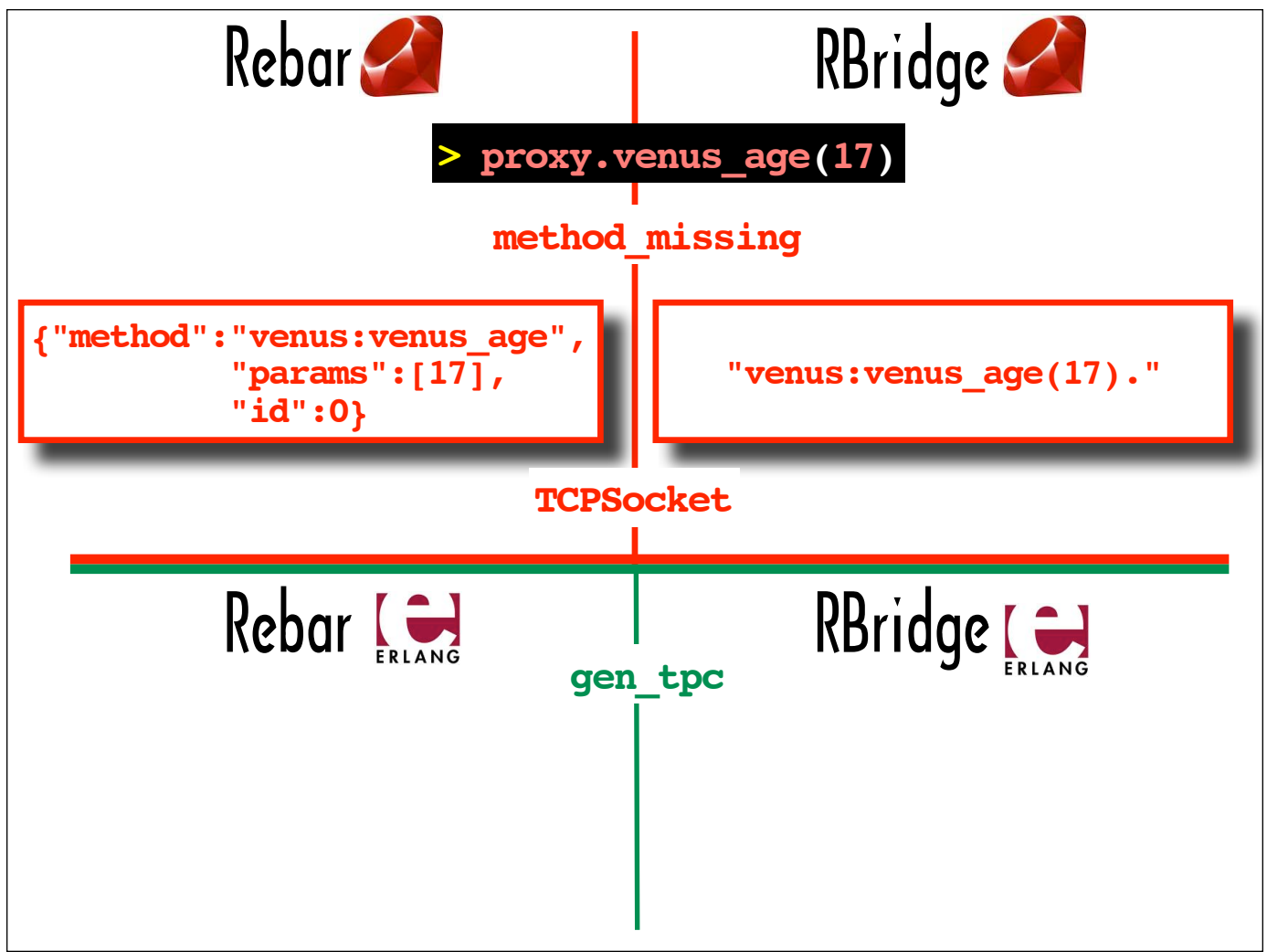
method_missing

```
{"method": "venus:venus_age",  
  "params": [17],  
  "id": 0}
```

```
"venus:venus_age(17)."
```

TCPSocket

Both Rebar and RBridge use TCP to send the formatted version of the method call on the proxy to their corresponding Erlang servers. Both use Ruby's `TCPsocket` class.



On the Erlang side, both Rebar's Erlang server and RBridge's Erlang server use Erlang's `gen_tpc` module to receive the formatted method call.

In the next couple of slides, I'll show how the Rebar and RBridge Erlang server implementations differ, and how the two different implementations made me see how much potential Erlang has for being a DSL host language and for supporting projects that require a degree of dynamism.

We'll look at Rebar's Erlang server implementation first.

Erlang Run-Time System Application (ERTS) Reference Manual

```
apply(Module, Function, Args) -> term() | empty()
```

Types:

```
Module = Function = atom()
```

```
Args = [term()]
```

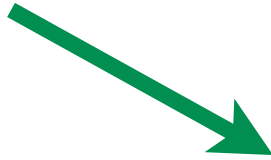
Returns the result of applying `Function` in `Module` to `Args`. The applied function must be exported from `Module`. The arity of the function is the length of `Args`.

Rebar uses the `apply` function, one of the Erlang BIFs (built-in-functions that can be found in the `erlang` module). This slide shows a section of the Erlang reference manual that covers the `apply` function.

As the “Types” sub-section indicates, you can pass apply a module name and a function name as atoms (which are comparable to symbols in Ruby) and the arguments as a list of terms (terms are Erlang expressions, and list syntax in Erlang is comparable to array syntax in Ruby).

The `apply` function will return the result of invoking the specified function on the specified module with the specified arguments.

```
apply(list_to_atom(Module),  
      list_to_atom(Function),  
      tuple_to_list(Params))
```



```
apply(venus,  
      venus_age,  
      [17])
```

The Rebar Erlang server processes the data it receives using the Erlang `json` module's `decode_string` function and populates the `Module`, `Function` and `Params` variables, based on the values it received.

The top code fragment shows the call to `apply` from the Rebar source code. It's using `list_to_atom` to convert the `Module` and `Function` values to atoms. You might expect it to use a function called something like "string_to_atom" instead of "list_to_atom", but there is no `String` datatype in Erlang. A group of characters enclosed by double quotes represents a list -- a list of characters.

The green-bordered box shows the `apply` function call with the values of the parameters substituted for the expressions that yield those values.

The Rebar Erlang server uses `gen_tcp:send` to send the return value of the `apply` function (ie the invocation of `venus:venus_age(17)` via the `apply` function) to Rebar's Ruby proxy.

Erlang Run-Time System Application (ERTS) Reference Manual

MODULE

`erl_eval`

MODULE SUMMARY

The Erlang Meta Interpreter

DESCRIPTION

This module provides an interpreter for Erlang expressions. The expressions are in the abstract syntax as returned by `erl_parse`, the Erlang parser, or a call to `io:parse_erl_exprs/2`.

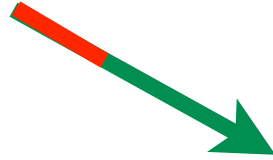
`exprs(Expressions, Bindings) -> {value, Value, NewBindings}`

Where Rebar uses `apply`, RBridge uses the `erl_eval` module's `exprs` function. The `exprs` function is an Erlang interpreter. It evaluates the expression it is passed.

This slide shows some details about `exprs` from the Erlang reference manual.

RBridge uses the `erl_scan` tokenizer to process the data it receives from TCP, and then uses the `erl_parse` function specified by the reference manual text in this slide to generate abstract syntax expressions to pass to the `erl_eval` module's `exprs` function.


```
"venus:venus_age(17)."
```



```
[{call,0,
 {remote,0,
  {atom,0,venus},
  {atom,0,venus_age}},
 [{integer,0,17}]]]
```

The top box shows the data passed to the RBridge Erlang server (ie the module name, a colon, the function name, the argument enclosed in parentheses, and a period -- all enclosed in quotes), and the bottom box shows the corresponding abstract syntax expression passed to `erl_eval:exprs` after it has been tokenized and parsed.

RBridge sends the value returned by `erl_eval:exprs` (ie the value of the call to `venus:venus_age(17)`) back to the Ruby proxy via TCP.



Erlectricity

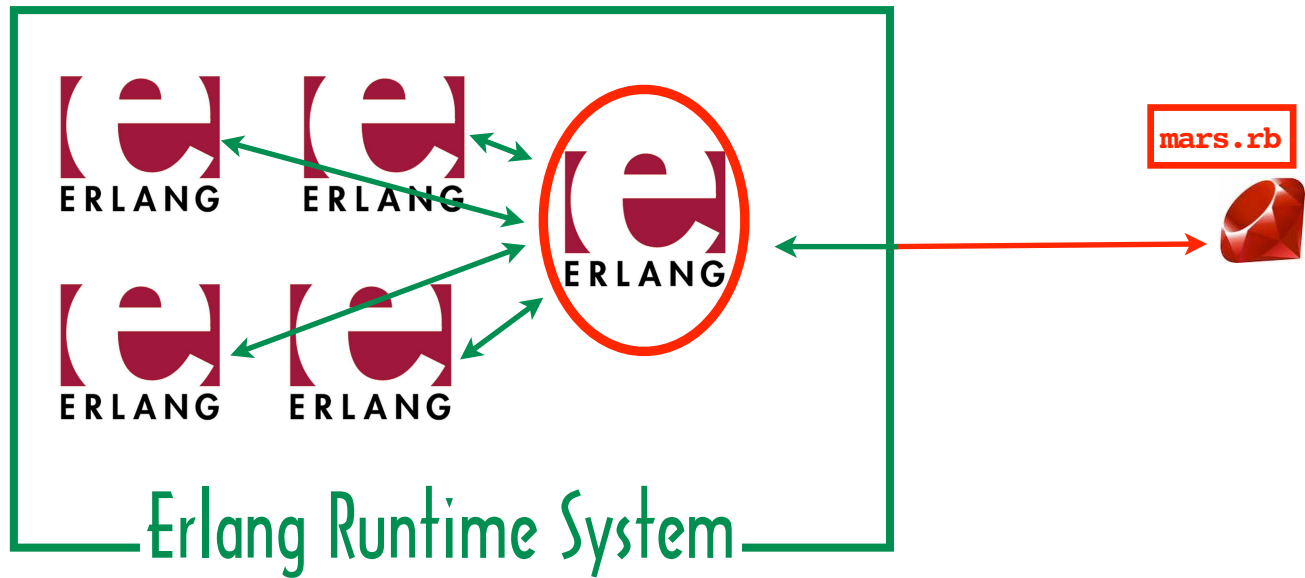


Scott Fleckenstein; Tom Preston-Werner

<http://github.com/mojombo/erlectricity>

The next Erlang/Ruby interop library we're going to look at is Erlectricity. Unlike Rebar and RBridge, which make Erlang functions accessible from the Ruby side, Erlectricity enables Erlang programs to initiate contact with Ruby routines. Once contact is established, the Erlang caller and Ruby service can exchange multiple messages.

Erlang's Ports System



Electricity leverages Erlang's Ports System, which is comprised of a set of BIFs ('Built-in Functions' that can be found in the `erlang` module) that enable an Erlang process to connect to an external process.

In this slide, the green box represents an Erlang runtime system with multiple active processes, and the Ruby represents a Ruby program. The sample Ruby program we're going to look at is called "mars.rb". The Erlang process circled in red represents the what is known as the "connected process" -- the one that opens a port to the external process.

Only the connected process communicates directly with the external process to ensure that the entire Erlang runtime system will not go down if the external process crashes.

Other Erlang processes that need the functionality provided by the external process go through the connected process because the external process is not able to send messages to any other process. The connected process typically exposes a function that wraps the code that sends messages to and receives messages from the external process.

Erlectricity



```
require 'erlectricity'

receive do |receiver|
  receiver.when([:mars_age, Fixnum]) do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
end
```

Here is the entire contents of mars.rb. The part highlighted in red is the business logic the program provides: the calculation that returns an age in Mars years.

The rest of the code is Erlectricity boilerplate code needed to communicate with an Erlang process. After the next few slides, we'll go over each line of this code in detail so you can see how Erlectricity works.

Before we take a behind-the-scenes look at Erlectricity, I want to show how easy it is to use.

Erlectricity



```
require 'erlectricity'

receive do |receiver|
  receiver.when([:mars_age, Fixnum]) do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
end
```



```
-module(mars).
-export([mars_age/1]).

mars_age(Age) ->
  Port = open_port({spawn, "ruby mars.rb"}, [{packet, 4},
                                              {nouse_stdio, exit_status, binary}],
                  port_command(Port, term_to_binary({mars_age, Age})),
  receive
    {Port, {data, Data}} ->
      {result, Mars_Years} = binary_to_term(Data),
      io:format("~p~n", [Mars_Years])
  end.
```

Here's all the code required to enable mars.rb to communicate with an Erlang process.

On the Ruby side, there's mars.rb.

On the Erlang side, there's a module called `mars` in a file called "mars.erl", with a function called `mars_age` that accepts a parameter called `Age`. Pass an `Age` to `mars:mars_age`, and it will print out that age in Mars years, courtesy of the calculation in mars.rb.

Erlectricity



```
require 'erlectricity'

receive do |receiver|
  receiver.when([:mars_age, Fixnum]) do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
end
```



```
-module(mars).
-export([mars_age/1]).
```

```
mars_age(Age) ->
  Port = open_port({spawn,"ruby mars.rb"},[{packet, 4},
                                             {nouse_stdio, exit_status, binary}],
  port_command(Port, term_to_binary({mars_age, Age})),
  receive
    {Port, {data, Data}} ->
      {result, Mars_Years} = binary_to_term(Data),
      io:format("~p~n", [Mars_Years])
  end.
```

```
Eshell V5.6.5 (abort with ^G)
1> mars:mars_age(17).
9.038720195638883
ok
```

Here's an Erlang console session in which `mars:mars_age` is invoked with an argument of 17. As you can see, Ruby is about 9 years old in Mars years.

Now we'll take a closer look at what happens behind the scenes, first on the Erlang side, and then on the Ruby side.

Erlectricity



ruby mars.rb

```
require 'erlectricity'

receive do |receiver|
  receiver.when([:mars_age, Fixnum]) do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
end
```



```
-module(mars).
-export([mars_age/1]).

mars_age(Age) ->
  Port = open_port({spawn,"ruby mars.rb"},[{packet, 4},
                                             nouse_stdio, exit_status, binary]),
  port_command(Port, term_to_binary({mars_age, Age})),
  receive
    {Port, {data, Data}} ->
      {result, Mars_Years} = binary_to_term(Data),
      io:format("~p~n", [Mars_Years])
  end.
```

The function `mars:mars_age` uses the `open_port` BIF (Erlang built-in function found in the `erlang` module) to run and establish contact with `mars.rb`. The `open_port` BIF takes two parameters: 1) a command (which in this case is invoking “`ruby mars.rb`”) and 2) a list of port settings (an Erlang list is enclosed in square brackets) The port settings that should be used with Erlectricity are:

1. the `packet` length. This is expressed as a tuple -- a collection of values enclosed in curly braces in Erlang -- where the first value is the atom `packet` (an atom is comparable to a symbol in Ruby, but is not prefixed with a colon) and the second value is the number 4. The `open_port` BIF will accept 4, 2, or 1, but 4 should be used with Erlectricity.
2. an atom which indicates whether STDIN and STDOUT should be used for input and output by the port. When the atom `nouse_stdio` is included in the port settings list, Erlang will use file descriptors 3 and 4 for input and output streams respectively. If the atom `use_stdio` is in the settings list, STDIN and STDOUT will be used (ie file descriptors 1 and 2). Erlectricity uses descriptors 3 and 4 by default.
3. the atom `exit_status`, which indicates that if the external process exits, a tuple including the atom `exit_status` and the exit status of the external process should be sent to the connecting process. Handling the exit status is beyond the scope of this talk.
4. the atom `binary`, which indicates that data exchanged should be in binary format.

Erlectricity



```
require 'erlectricity'

receive do |receiver|
  receiver.when([:mars_age, Fixnum]) do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
end
```



```
-module(mars).
-export([mars_age/1]).

mars_age(Age) ->
  Port = open_port({spawn,"ruby mars.rb"},[{packet, 4},
                                             {nouse_stdio, exit_status, binary}],
                  port_command(Port, term_to_binary({mars_age, Age})),
  receive
    {Port, {data, Data}} ->
      {result, Mars_Years} = binary_to_term(Data),
      io:format("~p~n", [Mars_Years])
  end.
```

Next, the `mars_age` function uses the `port_command` BIF, which writes to the output stream for the spawned command (ie "ruby mars.rb") to request the Mars years calculation.

It sends a tuple comprised of the atom `mars_age` (which corresponds to the symbol `:mars_age` in the Ruby code) and the value of the variable `Age` (which is passed into `mars:mars_age`).

Erlectricity

```
module Kernel
  def receive(input = nil, output = nil, &block)
    input ||= IO.new(3)
    output ||= IO.new(4)
    port = Erlectricity::Port.new(input, output)
    receiver = Erlectricity::Receiver.new(port, &block)
    receiver.run
  end
end
```



ruby mars.rb

```
receive do |receiver|
  receiver.when([:mars_age, Fixnum]) do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
end
```

Before we look at what happens when `mars.rb` receives a request, we'll look at what it does to prepare for receiving a request.

The red-bordered box shows the entire contents of `mars.rb` (with the exception of the `require` statement for “erlectricity”), which is a single call to `receive`.

The black-bordered box on the top of the slide shows how the `Erlectricity` library defines `receive` as a `Kernel` method.

The first two parameters to `receive` are optional, and the code in `ruby.rb` does not provide values for them. The code highlighted in red shows how these first two parameters are used. They are used to create an input stream and an output stream, which are passed to the constructor for an `Erlectricity::Port`.

The `Port` instance, `port`, reads from the input stream with a routine that decodes the binary data it receives and likewise writes to the output stream with a routine that takes care of encoding the data it sends. The `Port` instance's `read` and `write` routines also handle some type conversions. For example, data received in the Erlang Binary Term Format for an atom, will generally be converted to a symbol for use by Ruby.

Erlectricity

```
module Kernel
  def receive(input = nil, output = nil, &block)
    input ||= IO.new(3)
    output ||= IO.new(4)
    port = Erlectricity::Port.new(input, output)
    receiver = Erlectricity::Receiver.new(port, &block)
    receiver.run
  end
end
```



ruby mars.rb

```
receive do |receiver|
  receiver.when([:mars_age, Fixnum]) do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
end
```

Basically, all the mars.rb code does is pass the required block parameter to `receive`.

I've highlighted the block that mars.rb is passing to `receive`.



```

module Kernel
  def receive(input = nil, output = nil, &block)
    input ||= IO.new(3)
    output ||= IO.new(4)
    port = Erlectricity::Port.new(input, output)

    receiver = Erlectricity::Receiver.new(port,
                                          &block)

    receiver.run
  end
end

```


```

receive do |receiver|
  receiver.when([:mars_age, Fixnum]) do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
end

```

ruby mars.rb

The `receive` method, in turn, passes the block - “block stock and barrel” so to speak, along with the `Port` instance, to the constructor for the `Receiver` class.

What happens in the `initialize` method for a `Receiver` is that the block’s logic is executed with the freshly-minted `Receiver` as its parameter.

Erlectricity

```
module Kernel
  def receive(input = nil, output = nil, &block)
    input ||= IO.new(3)
    output ||= IO.new(4)
    port = Erlectricity::Port.new(input,output)
    receiver = Erlectricity::Receiver.new(port, &block)
    receiver.run
  end
end
```



ruby mars.rb

```
receive do |receiver|
  receiver.when([:mars_age, Fixnum])

  do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
end

end
```

The logic in the block calls a single method on its argument, `receiver` (the newly-minted `Receiver`): `when`.

The `when` method takes 2 arguments

Erlectricity

```
module Kernel
  def receive(input = nil, output = nil, &block)
    input ||= IO.new(3)
    output ||= IO.new(4)
    port = Erlectricity::Port.new(input,output)
    receiver = Erlectricity::Receiver.new(port, &block)
    receiver.run
  end
end
```



ruby mars.rb

```
receive do |receiver|
  receiver.when([:mars_age, Fixnum])

  do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
end

end
```

... an Array ...

Erlectricity

```
module Kernel
  def receive(input = nil, output = nil, &block)
    input ||= IO.new(3)
    output ||= IO.new(4)
    port = Erlectricity::Port.new(input,output)
    receiver = Erlectricity::Receiver.new(port, &block)
    receiver.run
  end
end
```



ruby mars.rb

```
receive do |receiver|
  receiver.when([:mars_age, Fixnum])
    do |age|
      receiver.send!([:result, age/(686.98/365.26)])
      receiver.receive_loop
    end
end
```

... and a block.

Erlectricity

```
module Kernel
  def receive(input = nil, output = nil, &block)
    input ||= IO.new(3)
    output ||= IO.new(4)
    port = Erlectricity::Port.new(input,output)
    receiver = Erlectricity::Receiver.new(port, &block)
    receiver.run
  end
end
```



ruby mars.rb

```
receive do |receiver|
  receiver.when([:mars_age, Fixnum])
    do |age|
      receiver.send![:result, age/(686.98/365.26)]
      receiver.receive_loop
    end
end
```

The `Array` represents a message profile, and it should be comprised of a symbol (in this case, `:mars_age`) and a datatype (in this case, `Fixnum`).

The block represents an action to take when a message that matches the specified profile is read from the `Erlectricity::Port`'s input stream.

The `when` method sets up a construct (an instance of `Matcher`) that maps message profiles to the blocks.

Erlectricity

```
module Kernel
  def receive(input = nil, output = nil, &block)
    input ||= IO.new(3)
    output ||= IO.new(4)
    port = Erlectricity::Port.new(input, output)
    receiver = Erlectricity::Receiver.new(port, &block)
    receiver.run
  end
end
```



ruby mars.rb

```
receive do |receiver|
  receiver.when([:mars_age, Fixnum]) do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
  receiver.when([:mars_weight, Fixnum]) do |weight|
    receiver.send!([:result, weight * 0.377])
    receiver.receive_loop
  end
end
```

If I wanted to add more functionality to mars.rb, I would add additional `when` calls to the block I pass to `receive`.

For example, this is what mars.rb might look like if I wanted to support calculating weight on Mars in addition to calculating Mars years.

Erlectricity

```
module Kernel
  def receive(input = nil, output = nil, &block)
    input ||= IO.new(3)
    output ||= IO.new(4)
    port = Erlectricity::Port.new(input,output)
    receiver = Erlectricity::Receiver.new(port, &block)
    receiver.run
  end
end
```



ruby mars.rb

```
receive do |receiver|
  receiver.when([:mars_age, Fixnum]) do |age|
    receiver.send!([:result, age/(686.98/365.26)])
    receiver.receive_loop
  end
end
```

Back to the original version of mars.rb...

We left off walking through the code in the `Kernel#receive` definition with the call to the `Receiver` constructor.

The next line, which also happens to be the last line in `receive`, is highlighted in red. It's a call to the newly created `Receiver`'s `run` method.

Electricity

```
module Kernel
```

```
  class Receiver
```

```
    def run
```

```
      loop do
```

```
        msg = port.receive
```

```
        return if msg.nil?
```

```
        case result = process(msg)
```

```
          when RECEIVE_LOOP then next
```

```
          when NO_MATCH
```

```
            port.skipped << msg
```

```
            next
```

```
          else
```

```
            break result
```

```
        end
```

```
      end
```

```
    end
```

```
  end
```

```
    lock)
```

```
    out)
```

```
    port, &block)
```

```
  b
```

Here's what `run` looks like. In a loop, it 1) calls `receive` on the Receiver's Port, which wraps a `read` call to the Receiver's input stream and 2) processes the message if the Erlang side has dispatched one (via the `process(msg)` call).

Electricity

```
module Kernel
```

```
  class Receiver
```

```
    def run
```

```
      loop do
```

```
        msg = port.receive
```

```
        return if msg.nil?
```

```
        case result = process(msg)
```

```
          when RECEIVE_LOOP then next
```

```
          when NO_MATCH
```

```
            port.skipped << msg
```

```
            next
```

```
          else
```

```
            break result
```

```
        end
```

```
      end
```

```
    end
```

```
  end
```

[:mars_age,
 17]

(x)

tb

The `process` method checks to see if the message matches any of the message profiles it supports.

In our example, the `Receiver` would detect that the symbol `:mars_age` and a `Fixnum` (17) match the message profile that is linked to ...

Electricity

```
module Kernel
```

```
  class Receiver
```

```
    def run
```

```
      loop do
```

```
        msg = port.receive
```

```
        return if msg.nil?
```

```
        case result = process(msg)
```

```
          when RECEIVE_LOOP then next
```

```
          when NO_MATCH
```

```
            do |age|
```

```
              receiver.send!([:result,
```

```
                             age/(686.98/365.26)])
```

```
              receiver.receive_loop
```

```
            end
```

```
          end
```

```
        end
```

```
      end
```

```
    end
```

[:mars_age,
 17]

```
do |age|  
  receiver.send!([:result,  
                  age/(686.98/365.26)])  
  receiver.receive_loop  
end
```

... the block that calculates the age in Mars years.

The `Receiver` instance's `send!` method calls the `Receiver`'s `Port`'s `write` wrapper to format the result of the calculation and send it to the Erlang caller.



JInterface

Package `com.ericsson.otp.erlang`

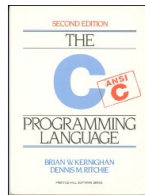
This package provides support for communication with Erlang and representation of Erlang datatypes.

See:

[Description](#)

Class Summary

AbstractConnection	Maintains a connection between a Java process and a remote Erlang, Java or C node.
AbstractNode	Represents an OTP node.
GenericQueue	This class implements a generic FIFO queue.
OtpConnection	Maintains a connection between a Java process and a remote Erlang, Java or C node.
OtpCookedConnection	Maintains a connection between a Java process and a remote Erlang, Java or C node.
OtpEpmc	Provides methods for registering, unregistering and looking up nodes with the Erlang portmapper daemon (Epmc).
OtpErlangAtom	Provides a Java representation of Erlang atoms.



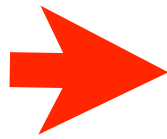
erl-interface

▼	connect	Folder
	ei_connect_int.h	C Header Source File
	ei_connect.c	C Source File
	ei_resolve.c	C Source File
	ei_resolve.h	C Header Source File
	eirecv.c	C Source File
	eirecv.h	C Header Source File
	eisend.h	C Header Source File
	send_exit.c	C Source File
	send_reg.c	C Source File
	send.c	C Source File
▼	decode	Folder
	decode_atom.c	C Source File
	decode_big.c	C Source File
	decode_bignum.c	C Source File

In addition to providing the Ports mechanism for building polyglot systems, Erlang is packaged with a couple of libraries that enable programs written in other languages to communicate with Erlang processes: JInterface and erl-interface.

JInterface targets Java and can be used in conjunction with JRuby -- while erl-interface, which is C-based, can be integrated with Ruby via Ruby's C Extension mechanisms.

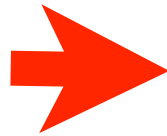
As you might infer from the JInterface Javadoc fragment on the top of the slide (with classes like `AbstractConnection`) and a screenshot of an erl-interface source directory on the bottom of the slide (with corresponding filenames like "ei_connect.c"), these two libraries offer comparable functionality.



Connecting Languages
(or polyglot programming example 1)

Ola Bini

<http://olabini.com/blog/2008/04/connecting-languages-or-polyglot-programming-example-1/>



Erlix



erl-interface

Killy Draw

<http://github.com/KDr2/erlix>

I don't know of any formal projects that provide a JRuby wrapper around JInterface, but I've seen a few blog posts with sample JRuby code that leverages JInterface to create client and server nodes that Erlang nodes communicate with much in the same way that they interact with other Erlang nodes.

We're going to look at a blog post by JRuby core team member Ola Bini called "Connecting Languages (or polyglot programming example 1)." It's about a library he wrote as part of a "15 minute experiment" to see how easy it would be to connect two languages that are "popular for solving wildly different kinds of problems." I'm going to show you how I used his library create a JRuby client that sends an Erlang server a request to calculate an age in Venus years and a JRuby server that calculates ages in Mars years for its clients.

On the C-side, there's Killy Draw's Erlix project, which provides access to erl-interface for Ruby programmers. After going over the Jinterface examples, we'll look at code using Erlix and code using Ola Bini's library side by side. Erlix is not yet Ruby 1.9.2 compatible, but 1.9.2 support is on the project roadmap.



JInterface



Connecting Languages



```
require 'java'
require '/path/to/OtpErlang.jar'

module Erlang
  import com.ericsson.otp.erlang.OtpSelf
  # additional imports

  class << self
    def tuple(*args)
      OtpErlangTuple.new(args.to_java(OtpErlangObject))
    end

    def num(value)
      OtpErlangLong.new(value)
    end

    def server(name, cookie)
      server = OtpSelf.new(name, cookie)
      server.publish_port
      while true
        yield server, server.accept
      end
    end

    def client(name, cookie)
      yield OtpSelf.new(name, cookie)
    end
    ...
  end
end
```

Here are excerpts from the Ola's experimental proof-of-concept library. Calls to the actual JInterface API are highlighted in green. The part that constitutes a JRuby wrapper is in red.

As you can see, the library is a very thin wrapper around JInterface. It's perfect for this talk because I don't have to strip away any layers to show you the language integration points.

The "Otp" that prefixes the JInterface class names stands for Open Telecom Platform and is part of the full name of the open source distribution of Erlang, which is "Erlang\OTP".



```
require 'java'
require '/path/to/OtpErlang.jar'

module Erlang
  import com.ericsson.otp.erlang.OtpSelf
  # additional imports
```

```
def server(name, cookie)
  server = OtpSelf.new(name, cookie)
  server.publish_port
  while true
    yield server, server.accept
  end
end

def client(name, cookie)
  yield OtpSelf.new(name, cookie)
end

...
end
end
```

Here's a closer look at the `server` and `client` methods, which both create new instances of the `OtpSelf`, a class that represents an Erlang node.

Both `client` and `server` expect to be passed a block, and are structured to pass the newly created client or server node to that block. The `server` method will also pass a connection to that block (the `server.accept` call returns an `OtpConnection` instance).

As the sample code in the following set of slides will show, the block passed to `client` and `server` should call the `OtpSelf` methods that enable nodes to send and receive messages from other nodes.

Client and server nodes created via JInterface can communicate with Erlang nodes and other nodes created via JInterface.



Interface Server

```
require 'erlang'
```

```
Erlang::server("mars_server@localhost", "mars") do |server, connection|
  while true
    message=connection.receive
    caller_pid = message.element_at(0)
    age = message.element_at(1)
    connection.send(caller_pid,
      Erlang::double(age.double_value/(686.98/365.26)))
  end
end
```

```
module Erlang
  ...
  def server(name, cookie)
    server = OtpSelf.new(name, cookie)
    server.publish_port
    while true
      yield server, server.accept
    end
  end
  ...
end
```

The top half of this slide shows all the code needed to create a minimal JRuby-based server.

The code in the box shows the source code for the `server` method from Ola Bini's "Connecting Languages" library, for reference.

This JRuby-based server calculates the Mars years equivalent of an age.

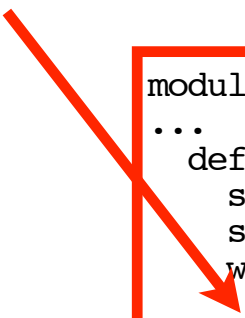
The first argument to `server`, in this case, "mars_server@localhost" represents the server name. The second argument, "mars", represents the security cookie name. The server will not accept connections from clients that do not use the specified cookie name.



Interface Server

```
require 'erlang'
```

```
Erlang::server("mars_server@localhost", "mars") do |server, connection|
  while true
    message=connection.receive
    caller_pid = message.element_at(0)
    age = message.element_at(1)
    connection.send(caller_pid,
      Erlang::double(age.double_value/(686.98/365.26)))
  end
end
```



```
module Erlang
  ...
  def server(name, cookie)
    server = OtpSelf.new(name, cookie)
    server.publish_port
    while true
      yield server, server.accept
    end
  end
  ...
end
```

This slide highlights the block passed to `server`.

As I noted a few slides back, the block takes two arguments -- the server and a connection (the `server.accept` call returns an `OtpConnection`).

When the block passed to the `server` method is invoked ...



Interface Server

```
require 'erlang'
```

```
Erlang::server("mars_server@localhost", "mars") do |server, connection|  
  while true  
    message=connection.receive ←  
    caller_pid = message.element_at(0)  
    age = message.element_at(1)  
    age = message.element_at(1)  
    connection.send(caller_pid,  
                    Erlang::double(age.double_value/(686.98/365.26)))  
  end  
end
```

```
module Erlang  
  ...  
  def server(name, cookie)  
    server = OtpSelf.new(name, cookie)  
    server.publish_port  
    while true  
      yield server, server.accept  
    end  
  end  
  ...  
end
```

.. it calls `receive` on the connection passed to it, which waits until a message is detected.



Interface Server

```
require 'erlang'
```

```
Erlang::server("mars_server@localhost", "mars") do |server, connection|  
  while true  
    message=connection.receive  
    caller_pid = message.element_at(0)  
    age = message.element_at(1)  
    connection.send(caller_pid,  
                    Erlang::double(age.double_value/(686.98/365.26)))  
  end  
end
```

```
module Erlang  
  ...  
  def server(name, cookie)  
    server = OtpSelf.new(name, cookie)  
    server.publish_port  
    while true  
      yield server, server.accept  
    end  
  end  
  ...  
end
```

A message that arrives is split into the caller process id (`caller_pid`) and the age the server will calculate in Mars years (`age`).



Interface Server

```
require 'erlang'
```

```
Erlang::server("mars_server@localhost", "mars") do |server, connection|
  while true
    message=connection.receive
    caller_pid = message.element_at(0)
    age = message.element_at(1)
    connection.send(caller_pid,
                    Erlang::double(age.double_value/(686.98/365.26)))
  end
end
```

```
module Erlang
  ...
  def server(name, cookie)
    server = OtpSelf.new(name, cookie)
    server.publish_port
    while true
      yield server, server.accept
    end
  end
  ...
end
```

The server then sends the Mars age to the consumer of its service by invoking `send` on the connection and supplying the caller's process id (`caller_pid`) as well as the result of applying the Mars years calculation to age.



JInterface Server

```
require 'erlang'

Erlang::server("mars_server@localhost", "mars") do |server, connection|
  while true
    message=connection.receive
    caller_pid = message.element_at(0)
    age = message.element_at(1)
    connection.send(caller_pid,
                    Erlang::double(age.double_value/(686.98/365.26)))
  end
end
```



Erlang Client

```
-module(mars_client).
-export([mars_age/1]).

mars_age(Age) ->
  {'mars_server', 'mars_server@localhost'} ! {self(), Age},
  receive
    Mars_Years ->
      io:format("~p~n", [Mars_Years])
  end.
```

Here's the Mars server on a split screen with a client written in Erlang.

The Erlang client, a module called `mars`, exposes a `mars_age` function that takes an `Age` in Earth years, sends it to the JRuby-based Mars server and prints out the reply it receives from the Mars server, which represents the `Age` in Mars years.

The exclamation point (!) in the line highlighted in green is Erlang's "send" operator. The send operator dispatches the expression on its right (in this case a tuple including the return value of `self()`, which is the process id of the Erlang client, and an `Age` in Earth years) to the process specified by the expression on its left (in this case a tuple including "mars_server", the registered name of the server, and its node name, "mars_server@localhost").

As we saw when we walked through the Jinterface server code, the first element of the message is assigned to `caller_pid`, which is used to address the response, and the second element is assigned to `age`, which is used in the Mars age calculation.



Interface Server

```
require 'erlang'

Erlang::server("mars_server@localhost", "mars") do |server, connection|
  while true
    message=connection.receive
    caller_pid = message.element_at(0)
    age = message.element_at(1)
    connection.send(caller_pid,
                    Erlang::double(age.double_value/(686.98/365.26)))
  end
end
```



Erlang Client

```
-module(mars_client).
-export([mars_age/1]).

mars_age(Age) ->
  {'mars_server', 'mars_server@localhost'} ! {self(), Age},
  receive
    Mars_Years ->
      io:format("~p~n", [Mars_Years])
  end.
```

The receive clause assigns the response to the variable `Mars_Years`.

The `io:format` call writes `Mars_Years` to the standard output.



Interface Server

```
require 'erlang'

Erlang::server("mars_server@localhost", "mars") do |server, connection|
  while true
    message=connection.receive
    call $ epmd
    age = message.element_at(1)
    connection.send(call_erlang(
    $ jruby mars_server.rb (age, 65.26)))
  end
end
```



Erlang Client

```
$ erl -sname mars_client@localhost -setcookie mars
-module
-export
mars_a
{ 'ma
rece
Ma
end.

Erlang (BEAM) emulator version 5.6.5 [source] [smp:2]
[async-threads:0] [kernel-poll:false]

Eshell V5.6.5 (abort with ^G)
(mars_client@localhost)1> mars:mars_age(17.0).
9.038720195638883
ok
```

To see this exchange in action, Erlang's Port Mapper Daemon (`epmd`) must be running. It starts up automatically when you use `erl` (the Erlang emulator) to create the client node -- but if you want to initiate the server/client conversation by firing up the JRuby-based server, you'll see a "java.io.IOException: Nameserver not responding" message in your console if you don't run the `epmd` command prior to starting the server.

The two console windows on the top part of the slide show the commands that primed the server to receive messages.

The console window on the bottom half shows the command that creates the client node using the `-setcookie` command line option to specify "mars", the cookie required for server access and the `-sname` option to specify a node name. The Erlang console window also shows a call to the `mars:mars_age` function and the printout of the response it received from the server: the Mars years equivalent of 17, which is 9.038720195638883.

JInterface Erlang Client

```
-module(mars_client).  
-export([mars_age/1]).  
  
mars_age(Age) ->  
    {'mars_server', 'mars_server@localhost'} ! {self(), Age},  
    receive  
        Mars_Years ->  
            io:format("~p~n", [Mars_Years])  
    end.
```

Erlectricity Erlang Client

```
mars_age(Age) ->  
    Port = open_port({spawn, "ruby mars.rb"}, [{packet, 4},  
                                                {nouse_stdio, exit_status, binary}],  
    port_command(Port, term_to_binary({mars_age, Age})),  
    receive  
        {Port {data, Data}} ->  
            {result, Mars_Years} = binary_to_term(Data),  
            io:format("~p~n", [Mars_Years])  
    end.
```

This split screen slide compares the Erlang-based JInterface client we've been looking at with the Erlang-based Erlectricity client we looked at a little earlier.

On the Erlang side, there's virtually no difference between communicating with a node created via JInterface and an Erlang-based node -- but using the Erlectricity requires that the Erlang client use special port protocols and handle conversions to and from binary.

An Erlang-based Erlectricity client does have the option of using the more idiomatic send operator (!) in lieu of `port_command`, making the client look a little more like a typical Erlang consumer of an Erlang-based service -- but there are special rules governing the message syntax for an Erlang process connected to a Port. For example, the message must include the `command` atom. Using the alternative syntax, the following could be used in place of the `port_command` line: `Port ! {command, term_to_binary({mars_age, Age})}`



Erlang Server

```
$erl -sname venus_server@localhost -setcookie venus  
-noshell -run venus_server
```

```
-module(venus_server).  
-export([start/0]).  
  
start() ->  
    Pid=spawn(fun venus_age/0),  
    register(venus_server,Pid).  
← venus_age ->  
    receive  
        {Caller, Age} ->  
            Caller ! ((365.26/224.68) * Age),  
            venus_age()  
    end.
```

Now we'll walk through a scenario with a server written in Erlang and a JRuby-based client.

The `venus_server` module's `start` function invokes its `venus_age` function in a new a process. The `venus_age` function receives and processes messages that include a caller process id and an age in Earth years. The spawned process calculates the age in Venus years and returns the result to the caller.

Upcoming slides provide a more detailed discussion of the `venus_age` function.

The command in the console representation at the top of the slide kicks off the Erlang-based Venus server, with the node name (`venus_server@localhost`), the security cookie name (`venus`), and the `-run` option indicating that the `venus_server` module's `start` function should be invoked (if we specify a module name, but no function name following the `-run` flag, Erlang will try to invoke the module's `start` function).



JInterface Client

```
require 'erlang'

Erlang::client("venus_client@localhost", "venus") do |client|
  venus_server = Erlang::OtpPeer.new("venus_server@localhost")
  connection = client.connect(venus_server)
  connection.send("venus_server", Erlang::tuple(client.pid,
                                                    Erlang::num(17)))

  venus_age = connection.receive
  puts venus_age.doubleValue
end
```

```
module Erlang
  ...
  def client(name, cookie)
    yield OtpSelf.new(name, cookie)
  end
  ...
end
```

This slide shows a JRuby-based consumer of the service exposed by the `venus_server` module and the source for the `client` method from Ola Bini's 'Connecting Languages' library.

The JRuby-based consumer passes the `client` method its node name (`venus_client@localhost`) and a security cookie value that matches the cookie specified at server start-up time (`venus`).

The JRuby client also supplies ...

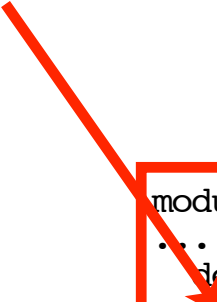


JInterface Client

```
require 'erlang'

Erlang::client("venus_client@localhost", "venus") do |client|
  venus_server = Erlang::OtpPeer.new("venus_server@localhost")
  connection = client.connect(venus_server)
  connection.send("venus_server", Erlang::tuple(client.pid,
                                                    Erlang::num(17)))

  venus_age = connection.receive
  puts venus_age.doubleValue
end
```



```
module Erlang
  ...
  def client(name, cookie)
    yield OtpSelf.new(name, cookie)
  end
  ...
end
```

... a block that expects the newly created client node to be passed in as the single block argument and that makes JInterface API calls on that client node to connect to the server, send a message to the server and receive a reply from the server.

In the next couple of slides, I'll discuss each line of the block I highlighted in red.



Interface Client

```
require 'erlang'  
  
Erlang::client("venus_client@localhost", "venus") do |client|  
  venus_server = Erlang::OtpPeer.new("venus_server@localhost")  
  connection = client.connect(venus_server)  
  connection.send("venus_server", Erlang::tuple(client.pid,  
                                                    Erlang::num(17)))  
  
  venus_age = connection.receive  
  puts venus_age.doubleValue  
end
```



Erlang Server

```
-module(venus_server).  
-export([start/0]).  
  
start() ->  
  Pid=spawn(fun venus_age/0),  
  register(venus_server,Pid).  
  
venus_age()->  
  receive  
    {Caller, Age} ->  
      Caller ! ((365.26/224.68) * Age),  
      loop()  
  end.
```

This slide highlights where the registered name of the server (venus_server) is referenced in the client code.



Interface Client

```
require 'erlang'

Erlang::client("venus_client@localhost", "venus") do |client|
  venus_server = Erlang::OtpPeer.new("venus_server@localhost")
  connection = venus_client.connect(venus_server)
  connection.send("venus_server", Erlang::tuple(client.pid,  

Erlang::num(17)))

  venus_age = connection.receive
  puts venus_age.doubleValue
end
```



Erlang Server

```
-module(venus_server).
-export([start/0]).

run()->
  Pid=spawn(fun venus_age/0),
  register(venus_server Pid).

venus_age()->
  receive
    {Caller, Age} ->
      From ! ((365.26/224.68) * EarthAge),
      loop()
  end.
```

The client sends the server its process id (`client.pid`) and the number 17, which are mapped to `Caller` and `Age` on the server side.



Interface Client

```

require 'erlang'

Erlang::client("venus_client@localhost", "venus") do |client|
  venus_server = Erlang::OtpPeer.new("venus_server@localhost")
  connection = venus_client.connect(venus_server)
  connection.send("venus_server", Erlang::tuple(client.pid,
                                                    Erlang::num(17)))

  venus_age = connection.receive
  puts venus_age.doubleValue
end

```



Erlang Server

```

-module(venus_server).
-export([start/0]).

run() ->
  Pid=spawn(fun venus_age/0),
  register(venus_server,Pid).

venus_age()->
  receive
    {Caller, Age} ->
      Caller ! ((365.26/224.68) * Age),
      loop()
  end.

```

The server uses the send operator (!) to reply to the consumer of its service with the formula for Venus years applied to the `Age` it received.

The client deposits the message it receives into the `venus_age` variable and then prints it out.



JInterface Client (Connecting Languages)



```
require 'erlang'

Erlang::client("venus_client@localhost", "venus") do |client|
  venus_server = Erlang::OtpPeer.new("venus_server@localhost")
  connection = venus_client.connect(venus_server)
  connection.send("venus_server", Erlang::tuple(client.pid,
Erlang::num(17)))

  venus_age = connection.receive
  puts venus_age.doubleValue
end
```



erl-interface Client (Erlx)



```
require "erlix"

ErlixNode.init("client_node", "venus")
connection=ErlixConnection.new("venus_server@localhost")
connection.esend("venus_server", ErlixTuple.new([ErlixPid.new(connection),
ErlixInt.new(17)]))

t=Thread.start("thread recv"){ |name|
  while true do
    m=connection.erecv
    puts m.message
  end
}
t.join
```

This split screen shows the JInterface client we were just looking at (that uses the erlang.rb library from Ola Bini's 'Connecting Languages' blog post) and a comparable Erlx client that can be used with C-Ruby. For each JInterface method, there's a corresponding Erlx call.

Erlx does not yet support Ruby-based servers, but that functionality is on the Erlx roadmap.



Tony Arcieri

<http://github.com/tarcieri/reia>

<http://groups.google.com/group/reia>

When a multi-lingual person is speaking or writing in one language and then inserts words or phrases from another language, linguists call it “code switching.” I think that’s an apt name for the kind of language integration we’ve been looking at so far: programs that are predominantly written in one language with some calls to a library written in another language.

Another form of linguistic “code switching” is mixing the words from one language with the grammar of another.

Reia, a language created by Tony Arcieri, exemplifies this second form of code switching. Its syntax is Ruby-like, but it runs on the Erlang virtual machine. The ways Tony combines languages are fascinating. Ruby’s dot operator (.) is sometimes referred to as a “message-sending” operator as opposed to a “method invocation” operator, and in an early version of Reia Tony experimented with modeling objects as Erlang processes using a dot (.) as the notation for sending messages to them in lieu of Erlang’s send operator (!). The roadmap for Reia calls for revisiting this vision of “concurrent objects”.

Another example of “code switching” in Reia is support for destructive assignment on top of the Erlang VM. In other words, Reia allows you to assign a value to a variable that already has a value, which is expressly prohibited in Erlang.



Erlang: demo.erl

```
-module(demo).  
-export([test/0]).  
  
test() ->  
    X = 23,  
    Y = 4 * X + 3,  
    X = 19,  
    io:format("~p~n", [X]).
```



Reia: demo.re

```
module Demo  
  def test  
    x = 23  
    y = 4 * x + 3  
    x = 19  
    "#{x}".puts()  
  end  
end  
  
Demo.test()
```

Towards the beginning of this talk, we looked at how Erlang creator Joe Armstrong demonstrated Erlang's single assignment policy in his Programming Erlang book.

The source code for this console-based demo is on the top of this slide in the form of a function in an Erlang module.

The contents of a file called "demo.re", a Reia module based on the same code, is displayed on the bottom half.

As you can see, the module definition conventions in Reia are very similar to those for Ruby. Notice that Erlang's statement separators (commas) and terminator (period) are not used in the Reia version. The ability to place free-standing code in a file with a module definition (the invocation of `Demo.test()`) is also Ruby-like.



Erlang

```
-module(demo).
-export([test/0]).
```

```
$ erl -run demo test -run init stop -noshell
{"init terminating in do_boot",{badmatch,19},[{demo,test,0},
{init,start_it,1},{init,start_em,1}]}
```

```
    I = 4 * X + 5,
    X = 19,
    io:format("~p~n", [X]).
```



Reia

```
module Demo
def test
```

```
$ reia reia_demo.re
Loading standard library from /Users/aok/reia/lib... done.
19
```

```
    "#{x}".puts()
end
end

Demo.test()
```

Not surprisingly, when we run the Erlang version (by passing the module name and the function name to `erl` following the `-run` flag), we see the same “badmatch” error we saw in Joe’s console session. The statement that prints out the value of `x` after the attempt to change its value to 19 is never reached.

The Reia version runs with no problem. The initial value of `x` is 23, but the final value that prints out is 19.



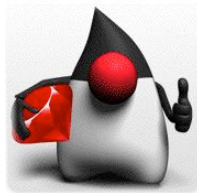
```
[{module,1,'Demo',
  [{module_type,module}],
 [{function,3,test,2,
  [{clause,3,
    [{tuple,3,[],{var,1,'_' }},
    [],
    [{match,3,{var,3,x_0},{integer,3,23}},
    {match,4,
      {var,4,y_0},
      {op,4,'+',
        {op,4,'*',{integer,4,4},{var,4,x_0}},
        {integer,4,3}},
      {match,5,{var,5,x_1},{integer,5,19}},
      {match,6,{var,6,x_2},{integer,6,20}},
      {match,7,{var,7,x_3},{integer,7,17}},
      {call,8,
        {remote,8,{atom,8,reia_dispatch},{atom,8,call}}
      [{call,8,
        {remote,8,
          {atom,8,reia_dispatch},
          {atom,8,call}},
        [{cons,8,{var,8,x_3},{nil,8}},
        {atom,8,join},
        {tuple,8,[],
          {atom,8,nil}}],
        {atom,8,puts},
        {tuple,8,[],
          {atom,1,nil}}]]]]]]],
  ]}]}
```

The image shows a side-by-side comparison of Erlang code. On the left, a function `test` is defined with several assignments to `x` and a `puts` statement. On the right, the corresponding Abstract Syntax Tree (AST) is shown. Green arrows and a green box highlight the correspondence between the code and the AST. Specifically, the `match` expressions in the AST correspond to the assignment statements in the code, and the `puts` statement in the code corresponds to the `puts` atom in the AST. The AST also shows the internal structure of the Erlang VM, including atoms, tuples, and lists.

This slide shows what's going on behind the scenes to keep the Erlang VM from rejecting the test function. I added a few additional pattern matching statements to `test` for good measure, and then I rigged, `reiac`, the Reia compiler, to print out the abstract syntax tree (AST) for the `Demo` module.

You can see that every time Reia encounters the pattern-matching operator (`=`) with `x` on the left-hand side, it creates a new variable (ie. `_x_0`, `_x_1`, `_x_2`, etc.) to bind the right-hand side value to in the “match” expression it generates. The variables prefixed with “`_x`” (which represent various versions of `x`) and their values are stored in a dictionary. When `puts` is called to print out `x` on the last line of the test function, Reia knows to use the latest version, `_x_3`.

Ruby/Scala Interop via JRuby



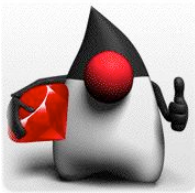
Because Scala runs on both the Java Virtual Machine (JVM) and the .NET Common Language Runtime (CLR), Ruby \Scala interop can be achieved via either JRuby or IronRuby. I'm going to focus using JRuby to bridge Scala and Ruby.

The Java Platform supports a common hosting API for scripting languages that run on the JVM. Any language that implements a scripting engine that complies with Java Service Request (JSR) 223 can be hosted in a Java application.

JSR 223 enables developers to support multiple scripting languages with the same host source code.

JRuby is packaged with an implementation of the JSR 223 API, and pretty much the same API calls can be used to invoke JRuby methods from within Java code or Scala code.

JRuby's JSR 223 implementation is built on top of its own core embedding API, known as RedBridge, which is more powerful and offers more configuration options than JSR 223. RedBridge also works similarly with both Java and Scala.



```
def mars_age(age)
  age / (686.98/365.26)
end
```



RedBridge

Scala

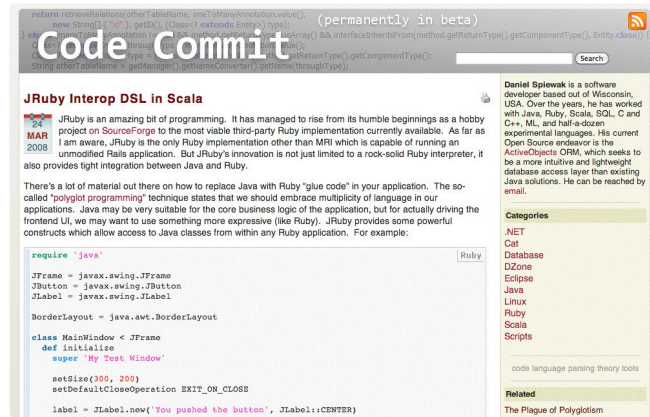
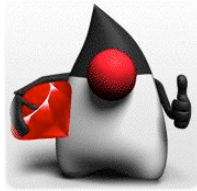
```
object Mars extends Application {
  val container = new ScriptingContainer()
  val receiver = container.runScriptlet(PathType.CLASSPATH,
    "mars.rb")

  val result = container.callMethod(receiver, "mars_age", 17.0,
    classOf[java.lang.Double])

  System.out.println(result)
}
```

If the Ruby method `mars_age` on the top of this slide was in a file called “`ruby.rb`”, then the Scala code on the bottom of the slide is the RedBridge code that could be used to invoke it.

Calling a top-level method involves obtaining a scripting engine, loading the Ruby source as a scriptlet, and calling the `callMethod` method with the following arguments: a reference to the loaded Ruby code, the method name, the arguments for the method, and a return type for the method return value.



ScalaRuby

JRuby Interop DSL in Scala

Daniel Spiewak

<http://www.codecommit.com/blog/ruby/jruby-interop-dsl-in-scala>

Daniel Spiewak detailed his thoughts about what natural JRuby\Scala interop might look like in a post called “JRuby Interop DSL in Scala” on his blog, Code Commit, and he posted the source code for a library he wrote that incorporates his ideas, called ScalaRuby.



ScalaRuby

```
object Main extends Application with JRuby {  
  require("mars")  
  val years:Double = 'mars_age(17.0)  
  println(years)  
}
```



RedBridge

```
object Mars extends Application {  
  val container = new ScriptingContainer()  
  val receiver = container.runScriptlet(PathType.CLASSPATH,  
    "mars.rb")  
  val result = container.callMethod(receiver, mars_age, 17.0,  
    classOf[java.lang.Double])  
  System.out.println(result)
```

The RedBridge-style method invocation we just looked at is on the bottom half of this slide, and the code required to invoke the same method using Daniel's ScalaRuby library is on the top half.

In the next couple of slides we'll do a line by line comparison.



ScalaRuby

```
require("mars")
```



RedBridge

```
val container = new ScriptingContainer()  
val receiver = container.runScriptlet(PathType.CLASSPATH,  
    "mars.rb")
```

Being able to use `require` to make a Ruby file accessible to a Scala program via ScalaRuby is a nice touch. The ScalaRuby library's `require` creates a string consisting of "require" and the base filename that it is passed (for example, in this case, "require 'mars'" would be constructed) and passing that string to the JSR 223 scripting API call that enables arbitrary Ruby code to be evaluated. In addition to providing an element of Ruby-like look and feel, this implementation has the advantage of being able to find files in locations on Ruby's load path, as well as those referenced on the CLASSPATH.

The RedBridge code that loads a script is considerably more verbose.



ScalaRuby



```
val years:Double = 'mars_age(17.0)
```



RedBridge



```
val result = container.callMethod(receiver, mars_age, 17.0,  
    classOf[java.lang.Double])
```

This slide highlights my favorite thing about Daniel's library.

It shows how the ScalaRuby library allows you to call a Ruby method from Scala just about the same way you would call it in Ruby (or just about the same way you would call a Scala function from Scala) -- with the exception of the single quote prefixing the method name, which designates the method identifier as a Scala symbol. Scala symbols are set off with a single quote the same way that Ruby symbols are prefixed with a colon.

I'm going to go over the two Scala features that make this quasi-natural method call possible, and then I'll explain how Daniel makes use of them in his DSL.



```
scala> 2
res0: Int = 2

scala> 2.compare(1)
res1: Int = 1

scala> 2.compare(2)
res2: Int = 0

scala> 2.compare(3)
res3: Int = -1
```

The first feature you need to be aware of in order to understand how Daniel's library works is Scala's implicit conversion mechanism.

In Scala, numeric literals are instances of the class `Int`. The implicit conversion mechanism makes it appear that it's possible to call methods like `max` and `compare` on numeric literals, even though these methods are not defined for `Ints`.

Here's a console session showing what happens when `compare` is called on an `Int`: Depending on whether the number passed to `compare` is less than, equal to or greater than the `Int` that `compare` is called on, either 1, 0 or -1 is returned. Looking at this console session, you'd never guess that there's no `Int#compare`.



```
implicit def intWrapper(x: Int)= new runtime.RichInt(x)
```

When Scala determines that a method does not exist for a particular receiver, it checks to see if an implicit conversion is defined to transform the receiver into an object that knows about the method call.

Here's the implicit conversion that transforms an `Int` into a `scala.runtime.RichInt`, which does support `compare`.

Like all implicit conversion definitions, this one includes the `implicit` keyword and a method that transforms an object of one type to an object of another type.



Parentheses, Parameters & Apply

```
scala> val ages = Array(17.0, 7.0)  
res0: Array[Double] = Array(17.0, 7.0)
```

Invokes

```
Array.apply(17.0, 7.0)
```

The second Scala feature we're going to look at before I can go into how function invocation works in Daniel's DSL is the way Scala handles parentheses. When Scala sees an object followed by a set of parentheses and there are one or more `apply` methods defined for that object, then Scala invokes the `apply` method with an arity that matches the number of items between the parentheses.

Anywhere that parentheses are used to delimit method parameters, `apply` can be called directly, instead.

The `apply` method is sometimes defined as a factory method. In this Scala console session, it's being used to create an `Array`. Incidentally, I populated `ages` with 17.0 and 7.0 because those numbers represent the ages of Ruby and Scala, respectively.



Parentheses, Parameters & Apply

```
scala> val ages = Array(17.0, 7.0)
res0: Array[Double] = Array(17.0, 7.0)
scala> ages(0)
res1: Double = 17.0
```

Invokes

Invokes

```
Array.apply(17.0, 7.0)
```

```
ages.apply(0)
```

The `apply` method is also frequently used by instances of collections classes to return an item from the collection at a specified index number.

Here a call to `ages(0)`, which invokes `ages.apply(0)` and returns the value of the first element in `ages`, is added to the console session that I began by creating the `Array` called `ages`.



Parentheses, Parameters & Apply

```
scala> val venusAge = (age:Double) => age * (365.26/224.68)
venusAge: (Double) => Double = <function>
scala> venusAge(17.0)
res0: Double = 27.63672779063557
```

Invokes



```
venusAge.apply(17.0)
```

Here, `apply` is called on a function object: `venusAge`. One or more variable names followed by a colon and the variable type enclosed in parentheses (eg. `(age:Double)`) represent a function parameter list. The code to the right of the `=>` represents the function body.

As with any other type of object, when Scala encounters one or more values enclosed in parentheses directly following a function object, like `venusAge`, it invokes `apply` on that function, passing along the specified argument list. When `apply` is called on a function, the logic in the body of the function gets executed.



ScalaRuby

```
def mars_age(age)  
  age / (686.98/365.26)  
end
```

```
val years:Double = 'mars_age(17.0)
```

So now we have all the pieces we need to understand how Daniel's Scala\Ruby DSL manages to invoke method calls in Ruby scripts, like the `mars_age` method shown in the red box, with the syntax shown in the green box.



```
def mars_age(age)
  age / (686.98/365.26)
end
```

```
val years:Double = 'mars_age(17.0)
```

```
implicit def sym2Method[R](sym:Symbol):(Any*)=>R = send[R](sym2string(sym))
```

Daniel defines the `sym2Method` implicit conversation to convert a `Symbol` into a instance of `com.codecommit.scalaruby.RubyMethod`, a class that wraps Ruby methods.

The conversation is triggered because the symbol is followed by parameters flanked by parentheses. Upon detecting the parentheses, Scala sets out to call `apply` on the symbol. Since there is no `apply` method defined for Scala's `Symbol` class, Scala finds the implicit conversion Daniel defined.



```
def mars_age(age)
  age / (686.98/365.26)
end
```

```
val years:Double = 'mars_age(17.0)
```

```
implicit def sym2Method[R](sym:Symbol):(Any*)=>R = send[R](sym2string(sym))
```

```
def send[R](str:String)={new RubyMethod[R](str2sym(str))}
```

This slide shows how the source for the `send` method specified in the conversion definition creates an instance of `com.codecommit.scalaruby.RubyMethod`, which, as we will see, does include a definition of `apply`.

After creating an instance of `com.codecommit.scalaruby.RubyMethod` based on the symbol (in this case, `'mars_age'`), Scala will invoke its `apply` method and pass it any arguments enclosed in the parentheses that caused the implicit conversation to fire.



```
def mars_age(age)
  age / (686.98/365.26)
end
```

```
val years:Double = 'mars_age(17.0)
```

```
implicit def sym2Method[R](sym:Symbol):(Any*)=>R = send[R](sym2string(sym))
```

```
def send[R](str:String)={new RubyMethod[R](str2sym(str))}
```

```
class RubyMethod[R](method:Symbol) extends ((Any*)=>R) {
  override def apply(params:Any*) = call(params.toArray)
  ...
  private[scalaruby] def call(params:Array[Any]):R = {
    // JRuby's JSR 223 Scripting API Calls
  }
}
```

The apply method for an instance of the Daniel's `com.codecommit.scalaruby.RubyMethod` class



ScalaRuby

```
def mars_age(age)
  age / (686.98/365.26)
end
```

```
val years:Double = 'mars_age(17.0)
```

```
implicit def sym2Method[R](sym:Symbol):(Any*)=>R = send[R](sym2string(sym))
```

```
def send[R](str:String)={new RubyMethod[R](str2sym(str))}
```

```
class RubyMethod[R](method:Symbol) extends ((Any*)=>R) {
  override def apply(params:Any*) = call(params.toArray)
  ...
  private[scalaruby] def call(params:Array[Any]):R = {
    // JRuby's JSR 223 Scripting API Calls
  }
}
```

... wraps a call to the method in the Ruby script (in this case, `mars_age`) using JRuby's JSR 223 scripting API and passing it the method name and the parameters.



<http://github.com/mcamou/scuby>

Daniel painstakingly tackles a number of integration pain points in addition to the ones I covered, including those related to calling methods on Ruby objects -- but he cautions about his library: "Remember that it is extremely untested and very experimental".

One of the issues he chose to let go, citing his right as a blogger presenting a proof-of-concept, is one he characterized as a "concurrency killer." He wrote the library using an early version of JRuby's JSR 223 scripting engine with an implementation that differs considerably from the current RedBridge-based one. That older implementation specifies that variables shared across the host \scripting language barrier should be prefixed with a "\$", which makes them global variables in Ruby. As a simplification, Daniel decided to name all method parameters sequentially, starting with "\$res0" for the first parameter for a method call and using "\$res1" through "\$resN", depending on how many arguments the method takes.

Mario Camou and his colleagues at abstra.cc felt that Daniel's prototype could help them with a polyglot system they were building, but they needed to do a lot of re-writing to address the concurrency issues and port it to recent versions of Scala and JRuby. They also added a number of innovations of their own.

abstra.cc called their library "Scuby", and they are using it in production. Scuby was released on github during RubyConf 2009.



Scuby



abstra.cc

<http://github.com/mcamou/scuby>

```
// Create a proxy object for the Ruby BackEnd class
val backEnd = RubyClass('BackEnd)

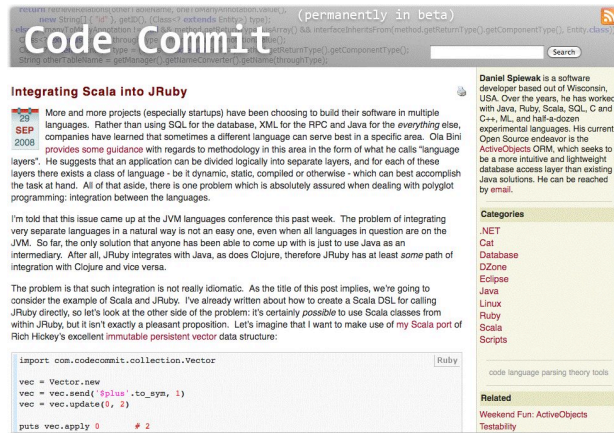
// Call a Ruby method with no parameters
backEnd ! 'prepare_data
```

This slide shows a couple of things you can do with Scuby, beginning with obtaining a reference to a Ruby class (`BackEnd` is an actual Ruby class, defined in an `.rb` file).

Scuby enables you to call methods on a proxy created via `RubyClass` using the `!` operator -- directly referencing Erlang's send operator and thereby combining idioms from 3 of the languages I am covering: Scala, Erlang and Ruby.

This first release of Scuby focuses on constructs that `abstra.cc` uses most often, like being able to call methods on objects easily.

Because `abstra.cc` didn't need to use top-level functions in their production code, Scuby's syntax for a top-level method call is not as slick as the symbol-based `ScalaRuby` notation we just analyzed, but making top-level method calls more natural is on the roadmap for the future, along with enabling developers to pass a JRuby block where Scala expects a function and wrapping JRuby collections for Scala consumption.



Integrating Scala into JRuby

Daniel Spiewak

<http://www.codecommit.com/blog/ruby/integrating-scala-into-jruby>

In addition to exploring Scala as a host language for JRuby, Daniel Spiewak has also given a lot of thought to what constitutes a good interop experience when accessing Scala from JRuby.

He created a library that enables Ruby `PROCS` and collections to behave more like their Scala counterparts (ie Scala functions and Scala collections) and that also enables JRubyists to integrate Scala code into their polyglot applications using idiomatic JRuby.

Before we look at what Daniel's library brings to the table, let's take a look at what you can do with Scala in a JRuby console session, out of the box.



```
object Venus {  
  def venusAgeMethod(age:Double):Double = age * (365.26/224.68)  
  val venusAgeFunction = (age:Double) => age * (365.26/224.68)  
}
```

Here's a Scala version of the Venus library for converting Earth years to Venus years.

It includes a Scala method definition as well as a Scala function definition. The method and the function provide the same functionality, but I'm including both in the Scala source because you need to invoke them differently from within JRuby.

I've chosen to define the Venus library as a singleton object. By virtue of my using the `object` keyword the way I did, Scala will create a singleton `Venus` object with all static methods (well, its one method will be static).

The method, `venusAgeMethod`, only exists within the context of the singleton object's class.

On the other hand, the function, `venusAgeFunction`, is a standalone object assigned to a variable, and can be passed to functions and methods that accept function arguments. In many ways, the function is no different than any other property, of any other type, like an `Int` or a `String`, that you might define for the `Venus` object and that you can pass to a function or a method as an argument.



```
object Venus {  
  def venusAgeMethod(age:Double):Double = age * (365.26/224.68)  
  val venusAgeFunction = (age:Double) => age * (365.26/224.68)  
}
```



JRuby

```
irb(main):001:0> require 'scala-library.jar'  
=> true  
irb(main):002:0> require 'venus.jar'  
=> true  
irb(main):003:0> Java::Venus.venusAgeMethod(17.0)  
=> 27.63672779063557  
irb(main):003:0> Java::Venus.venusAgeFunction.apply(17.0)  
=> 27.63672779063557
```

The bottom half of the slide shows how to invoke `venusAgeMethod` and `venusAgeFunction` in an interactive `jirb` session.

I have “required” `scala-library.jar`, which is part of the standard Scala distribution, but you can skip this step if you add `scala-library.jar` to JRuby’s classpath. I have also “required” `venus.jar`. After compiling the Venus library, I used the `jar` tool to package them in a file called “`venus.jar`”.

The Venus library can be accessed with syntax that resembles Ruby notation for nested modules. Because I have not placed `Venus` in a package, all I need to do is prefix its name with “`Java::`”. Additional package names would be appended together, following “`Java::`”. If I had placed `Venus` in a package called “`Demo.Test`”, I would have needed to follow “`Java::`” with “`DemoTest::`” (ie `Java::DemoTest::Venus`).

To invoke the method logic for `venusAgeMethod`, you can just use the dot notation to call `venusAgeMethod` on `Venus`.

You use the dot notation to access `venusAgeFunction` and then call `apply` on `venusAgeFunction` to invoke the function logic.



```
object Venus {  
  def venusAgeMethod(age:Double):Double = age * (365.26/224.68)  
  val venusAgeFunction = (age:Double) => age * (365.26/224.68)  
}
```



JRuby

```
irb(main):001:0> require 'scala-library.jar'  
=> true  
irb(main):002:0> require 'venus.jar'  
=> true  
irb(main):003:0> Java::Venus.venusAgeMethod(17.0)  
=> 27.63672779063557  
irb(main):003:0> Java::Venus.venusAgeFunction.apply(17.0)  
=> 27.63672779063557  
irb(main):003:0> Java::Venus.venus_age_method(17.0)  
=> 27.63672779063557  
irb(main):003:0> Java::Venus.venus_age_function.apply(17.0)  
=> 27.63672779063557
```

As this slide shows, you can use either the Ruby method naming convention of separating the words that constitute a method name with underscores or you can use camel case style naming convention favored by Scala and Java.

I tend to use the Scala-style naming conventions when invoking methods or functions that were written in Scala. I think being able to use the “dot” notation makes the interaction with Scala-based objects and functions seem easy enough. Taking the step of “Rubifying” the Scala function or method names feels to me like taking an English word or phrase that has its origins in another language, like “rendezvous” and using and pronouncing the “rend” to rhyme with “end” and the “dez” to rhyme with “fez.”



JRuby Support for Passing Ruby Blocks to Scala Methods

```
> ages = Java::ScalaCollectionMutable::ListMap.new
=> #<Java::ScalaCollectionMutable::ListMap:0xeb04d34>
> ages.update("Ruby", 17)
=> nil
=> ages.update("Scala", 7)
=> nil
> mars_ages = ages.mapValues {|age| age / (686.98/365.26)}
=> #<#<Class:01x29516070>:0x49dd63c9>
> mars_ages.mk_string(", ")
=> "Ruby -> 9.038720195638883, Scala -> 5.316894232728754"
```

In many cases, no special Scala/Ruby bridge code is needed in order to invoke a Scala method that accepts a function argument and pass it a Ruby block in lieu of a Scala function.

An example of a Scala method that takes a function argument is `mapValues`, which can be called on a Scala `ListMap` to generate a `ListMap` with the same keys as the original `ListMap` pointing to values that are the result of applying the supplied function to each value of the original `ListMap`.

The first few lines in this `jirb` session create a `ListMap` and populate it. `ListMap`'s `update` method takes a key parameter and a value parameter and is used to add the specified key/value pair to the `ListMap`. Here, the `ListMap` is populated with keys that are the names of programming languages ("Ruby" and "Scala") and values that are the ages of those languages. As I noted earlier, Ruby is 17 years old. Scala is 7 years old.

Here, `mapValues` is used to generate a `ListMap` with keys that are names of programming languages and with corresponding values that are the ages of those languages in Mars years.

Notice that I was able to specify the calculation to perform on each value in the `ListMap` by passing a Ruby block to the Scala method `mapValues` (ie `{|age| age / (686.98/365.26)}`).

The `mk_string` call on the last line entered in this `jirb` session returns a string representation of the key/value pairs in the `ListMap`, separated by the specified separator. As you can see, the `mars_age` `ListMap` generated with `mapValues` contains the ages of the included languages in Mars years.



JRuby Support for Java Interfaces



```
ok_button=JButton.new( 'OK' )  
ok_button.add_action_listener( )
```

javax.swing
Class JButton

↑
Expects an ActionListener

Method Summary

void	addActionListener(ActionListener l) Adds an ActionListener to the button.
------	--

java.awt.event

Interface ActionListener

Method Summary

void	actionPerformed(ActionEvent e) Invoked when an action occurs.
------	--

The reason why it's possible to pass a Ruby block to `mapValues` or any other Scala function or method that takes a function argument has to do with the way that JRuby's built-in support for Java Interfaces works.

A Java interface defines a group of methods that any class that implements the interface needs to provide implementations for.

`JButton`'s `add_Action_Listener` is an example of a method that expects an interface to be passed to it. As the JavaDoc for `addActionListener` shows, `addActionListener` takes an `ActionListener` argument, and as the JavaDoc for `ActionListener` shows, `ActionListener` is an interface.

This slide actually shows the entire method summary for `ActionListener`. An `ActionListener` only needs to provide a definition for a single method: `actionPerformed`.

The `ActionListener`'s `actionPerformed` method is invoked when the `JButton` is pressed. Linking a button to a particular action is achieved by assigning it an `actionListener` with an `actionPerformed` implementation that carries out that action.



JRuby Support for Java Interfaces



ButtonAction implements ActionListener

```
class ButtonAction
  include ActionListener
  def action_performed(e)
    puts "AOK!"
  end
end
```

java.awt.event

Interface ActionListener

Method Summary

void	actionPerformed (ActionEvent e) Invoked when an action occurs.
------	--

ButtonAction is a sample ActionListener implemented in JRuby. It meets the minimum requirements for an ActionListener that can be passed to addActionListener. That is, it not only implements actionPerformed, it includes ActionListener (via an include statement).

The include statement is important because Java does not support duck-typing. Even if a JRuby class provides definitions for all of the methods required by an interface, Java will not recognize it as an implementation of that interface if it doesn't reference the interface name in an include statement.

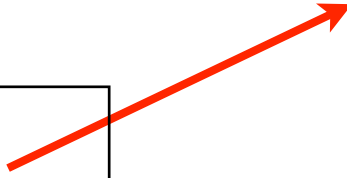


JRuby Support for Java Interfaces



```
ok_button=JButton.new('OK')  
ok_button.add_action_listener(ButtonAction.new)
```

```
class ButtonAction  
  include ActionListener  
  def action_performed(e)  
    puts "AOK!"  
  end  
end
```



If we pass a new `ButtonAction` to `add_action_listener` for `ok_button`, the text “AOK!” will be printed out when the button is pressed.



JRuby Support for Java Interfaces



```
ok_button=JButton.new('OK')  
ok_button.add_action_listener(ButtonAction.new)
```

```
{ puts "AOK!" }
```

```
class ButtonAction  
  include ActionListener  
  def action_performed(e)  
    puts "AOK!"  
  end  
end
```

JRuby also provides a shortcut for supplying an interface implementation to a Java method that takes an interface argument.

In lieu of requiring you to create a class that implements an interface and passing an instance of that class to a method expecting that interface, JRuby allows you to supply the logic for an interface's required method in the form of a Ruby block.

When JRuby detects that you have passed a block to a Java method that expects a class that implements an interface, JRuby creates a class that implements the interface for you -- behind the scenes. JRuby uses the body of the block as the method body.

This slide shows how I can simply pass the block `{ puts "AOK" }` to `add_action_listener` if I want "AOK" to be printed out when the button is pressed.



JRuby Support for Scala Traits vis-a-vis Java Interfaces



scala



Function1

```
def andThen[A](g: (R) ⇒ A): (T1) ⇒ A
```

```
(f andThen g)(x) == g(f(x))
```

```
def apply(v1: T1): R
```

```
attributes: abstract
```

```
def compose[A](g: (A) ⇒ T1): (A) ⇒ R
```

```
(f compose g)(x) == f(g(x))
```

```
def toString(): String
```

```
Returns a string representation of the object.
```

So how does this handling of Java interfaces relate to Scala functions? The tie-in is that in Scala functions are represented as traits -- and Scala's traits are comparable to Java's interfaces.

In the Scala there is trait defined for a function that takes 0 arguments, a function that takes one argument, a function that takes 2 arguments -- up through a function that takes 22 arguments.

This slide shows a portion of the Scaladoc for `Function1`, which represents a function that takes a single argument. The "t" to the right of the "Function1" label stands for "Trait" to signify that `Function1` is a trait.

One main difference between traits and interfaces is that traits may provide implementations for all or any of their constituent methods, while interfaces do not supply any implementations. `Function1` provides implementations for 3 of its methods. It does not provide an implementation for the `apply` method, which I have circled to highlight that it is tagged as `abstract`.

Ultimately, Scala code compiles into Java bytecode -- and at the Java bytecode level a trait is represented by interface that includes method signatures for all of the trait's methods, both abstract and concrete -- as well as classes that use the definitions provided by the trait to implement the concrete methods. The Scala compiler delegates to these generated classes when it encounters a class that extends a trait (`extend` is the Scala key word you use to incorporate a trait) that includes concrete methods.



JRuby Support for Java Interfaces



```
ok_button=JButton.new( 'OK' )  
ok_button.add_action_listener { puts "AOK!" }
```



JRuby Support for Scala Traits



```
> mars_ages = ages.mapValues {|age| age / (686.98/365.26)}  
=> #<#<Class:01x29516070>:0x49dd63c9>  
> mars_ages.mk_string(", ")  
=> "Ruby -> 9.038720195638883, Scala -> 5.316894232728754"
```

I'm only able to tap JRuby's Java interface handling when I supply a Ruby block argument to `mapValues` in the `jirb` session on the bottom half of this slide because `mapValues` calls `apply` on its argument, and `apply` is defined as an abstract method on the `Function1` trait.

JRuby does not distinguish between an abstract method that started life as part of a trait in a Scala source file and one that originated in a Java interface definition. Just as JRuby creates an instance of a class that implements `actionPerformed` based on the logic in the body of the block passed to `addActionListener` for the code fragment on the top half of the slide, it creates a class that implements `apply` with the body of the block passed to `mapValues` as the method body in the `jirb` session on the bottom part of the slide.

If instead of `apply`, the `mapValues` logic called one of `Function1`'s concrete methods ...

Scala: Using Function1#compose

scala



```
def andThen[A](g: (R) => A): (T1) => A
```

```
(f andThen g)(x) == g(f(x))
```

```
def apply(v1: T1): R
```

```
attributes: abstract
```

```
def compose[A](g: (A) => T1): (A) => R
```

```
(f compose g)(x) == f(g(x))
```

```
def toString(): String
```

```
Returns a string representation of the object.
```

... that is, if it called `andThen` or `compose`, (which I circled here, along with its description), you'd see an error in the console.

The concept that a Scala trait can implement one or more of its methods is outside of JRuby's frame of reference. Not only does JRuby not know how to access the generated classes that implement a trait's concrete methods, JRuby is not aware that these classes even exist.

What if you wanted to pass a Ruby block to a Scala method that takes a function argument and calls `compose` on that function argument?

This is where Daniel's library, called `scala.rb`, comes in. It enables you to create Ruby `Proc` wrappers that behave just like Scala functions for most intents and purposes.

I'm going to show sample JRuby code that calls `compose` on one of these Ruby `Proc` wrappers. But first, I'll show how `compose` is used in Scala. It is part of Scala's support for function composition. It creates a composite function that passes the results of one function to another.

Scala: Using Function1#compose

scala



Function1

```
def andThen[A](g: (R) => A): (T1) => A
```

```
(f andThen g)(x) == g(f(x))
```

```
def apply(v1: T1): R
```

```
def compose[A](g: (A) => T1): (A) => R
```

```
(f compose g)(x) == f(g(x))
```

```
def toString(): String
```

```
Returns a string
```

$f(g(x))$

Here I've blown up the Scaladoc description of `Function1`'s implementation of `compose`.

If `f` is a function that takes a single argument and `g` is a function that takes a single argument, then calling `compose` on `f` with `g` as the argument to `compose` yields a third function that takes a single argument.

If you invoke this third function and pass it an argument called `x`, it will do the following:

1. invoke `g` with `x` as its argument and
2. invoke `f` with the return value of the call to `g` as its argument.

Scala: Using Function1#compose

```
scala> val planetCatAge = (planetAge: (Double) => Double) =>
    {
      val catAge = (age: Double) => age * 6.2
      planetAge.compose(catAge)
    }
planetCatAge: ((Double) => Double) => (Double) => Double =
<function1>
```

This multi-line entry in a Scala console shows an example of a Scala function that leverages `compose`.

It takes a single argument. Its argument, which is referenced as `planetAge` in the function definition, represents a function that calculates an age on a particular planet.

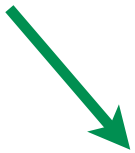
The `planetCatAge` function defines a local variable called `catAge`, which represents a function that returns the age passed to it in cat years.

The `planetCatAge` function then calls `compose` on `planetAge` with `catAge` as the argument to `compose`. This creates a new function that calculates an age in cat years on a planet.

I'll show `planetCatAge` in action on the next slide, so you can see how it works.

Scala: Using Function1#compose

```
scala> val planetCatAge = (planetAge: (Double) => Double) =>
    {
      val catAge = (age: Double) => age * 6.2
      planetAge.compose(catAge)
    }
planetCatAge: ((Double) => Double) => (Double) => Double =
<function1>
```



Using planetCatAge in Scala

```
scala> val venusAge = (age: Double) => age * (365.26/224.68)
venusAge: (Double) => Double = <function1>
```

The planetCatAge function definition is displayed on the top part of this slide for reference.

In the console session on the bottom part of this slide, as a first step for using planetCatAge to create a function that calculates an age in cat years on Venus, I created a function to pass to planetCatAge. It's the function that returns the age passed to it in Venus years, which we've seen a number of times. In this console session, I called it venusAge.


Scala: Using Function1#compose

```
scala> val planetCatAge = (planetAge: (Double) => Double) =>
  {
    val catAge = (age: Double) => age * 6.2
    planetAge.compose(catAge)
  }
planetCatAge: ((Double) => Double) => (Double) => Double =
<function1>
```

Using planetCatAge in Scala

```
scala> val venusAge = (age: Double) => age * (365.26/224.68)
venusAge: (Double) => Double = <function1>

scala> val venusCatAge = planetCatAge(venusAge)
venusCatAge: (Double) => Double = <function1>
```



Next, I passed `venusAge` to `planetCatAge` to generate a new function that takes an age as a parameter and returns that age in cat years on Venus.

Since a Scala interactive console session shows the type of each entered expression, we can confirm that passing `venusAge` to `planetCatAge` yielded a new function. Scala's notation for a function that takes a `Double` and returns a `Double` is: `(Double) => Double`.

Scala: Using Function1#compose


```
scala> val planetCatAge = (planetAge: (Double) => Double) =>
  {
    val catAge = (age: Double) => age * 6.2
    planetAge.compose(catAge)
  }
planetCatAge: ((Double) => Double) => (Double) => Double =
<function1>
```

Using planetCatAge in Scala

```
scala> val venusAge = (age: Double) => age * (365.26/224.68)
venusAge: (Double) => Double = <function1>

scala> val venusCatAge = planetCatAge(venusAge)
venusCatAge: (Double) => Double = <function1>

scala> venusCatAge(17.0)
res1: Double = 171.34771230194056
```



In the final line of this Scala console session, we pass the age 17.0 to `venusCatAge` and see that the equivalent of 17 Earth years in cat years on Venus is around 171.35.

Now we're ready to invoke `planetCatAge` in a `jirb` session, with a Ruby `Proc` wrapper passed in to supply the planet age calculation in lieu of a Scala function.

Using a Ruby `Proc` Wrapper in Lieu of `Function1`



```
object SpaceCats {  
  val planetCatAge = (planetAge: (Double) => Double) =>  
  {  
    val catAge = (age: Double) => age * 6.2  
    planetAge.compose(catAge)  
  }  
}
```



JRuby



```
> mars_age = proc {|age| age / (686.98/365.26)}
```

I placed `planetCatAge` inside a singleton object I named `SpaceCats`. On the top half of this slide is the source for `SpaceCats`, which I compiled and packaged in a jar so I could access it from a `jirb` session.

To save space I'm not going to show the `require` statements I used to allow me to access the jar containing `SpaceCats` and Daniel's library. You can find the full source for this and the rest of the examples at: <https://github.com/A-OK/RubyIsFromMars>

In the `jirb` session on the bottom half of the slide, I've created `mars_age`, a `Proc` to pass to `planetCatAge` using `scala.rb`'s `Proc` wrapper.

Using a Ruby `Proc` Wrapper in Lieu of `Function1`



```
object SpaceCats {  
  val planetCatAge = (planetAge: (Double) => Double) =>  
  {  
    val catAge = (age: Double) => age * 6.2  
    planetAge.compose(catAge)  
  }  
}
```



JRuby

```
> mars_age = proc {|age| age / (686.98/365.26)}  
> mars_cat_age = Java::SpaceCats.planetCatAge.apply(mars_age.to_function)
```

Next, on the line highlighted in red, I use `planetCatAge` to create `mars_cat_age`, a Scala function that returns its age argument in cat years on Mars.

Even with Daniel's library, I can't pass a `Proc` like `mars_age` to `planetCatAge` directly. But I can use the `to_function` call, which the red arrow is pointing to, to obtain a `Proc` wrapper that can be passed to `planetCatAge`. The `Proc` wrapper exposes all the methods a Scala `Function1` exposes.

Notice that I called `apply` on `planetCatAge` to invoke it. As I discussed earlier, a Scala function's `apply` implementation executes the function's logic. In Scala, you rarely see an explicit call to `apply` because parentheses following a function name automatically trigger a call to the function's `apply` method. The parentheses do not translate to an `apply` call in JRuby, so `apply` must be called explicitly in order to invoke a Scala function.

Just as `planetCatAge` returned a function that calculates an age in cat years on Venus when we passed it the `venusAge` function in the Scala console, `planetCatAge` creates a function that returns an age in cat years on Mars when we pass it the `mars_age` `Proc` wrapper.

Using a Ruby Proc Wrapper in Lieu of Function1



```
object SpaceCats {  
  val planetCatAge = (planetAge: (Double) => Double) =>  
  {  
    val catAge = (age: Double) => age * 6.2  
    planetAge.compose(catAge)  
  }  
}
```



JRuby

```
> mars_age = proc {|age| age / (686.98/365.26)}  
> mars_cat_age = Java::SpaceCats.planetCatAge.apply(mars_age.to_function)  
> mars_cat_age.apply(17.0)   
=> 56.0400652129611
```

On the last line of this session, I call `apply` on `mars_cat_age` to invoke its function logic, and I pass it the age 17.

Now that you've seen an example of how to use a wrapped Ruby `Proc` that supports `Function1`'s abstract and concrete methods, I'll walk through the parts of Daniel's `scala.rb` library that support this feature.



scala.rb: Scala Function Wrappers from Ruby Procs



scala



Function1

```
def andThen[A](g: (R) => A)
  (f andThen g)(x) == g(f(x))
def apply(v1: T1): R
  attributes: abstract
def compose[A](g: (A) => T)
  (f compose g)(x) == f(g(x))
def toString(): String
  Returns a string representati
```

module ScalaProc

class ScalaFunction

```
def initialize(delegate)
  @delegate = delegate
end
```

```
def apply(*args)
  # invoke @delegate with args
end
```

```
for n in 0..22
  eval "\
```

```
class Function#{n} < ScalaFunction
  include Scala::Function#{n}
end
end
end
```

First we'll account for `Function1`'s one abstract method: `apply`.

Daniel's `ScalaProc` module contains the `ScalaFunction` class, which takes a `Proc` in its constructor and implements `apply` to invoke the body of the `Proc`, which is represented by `@delegate`. I'm not showing all of Daniel's `apply` implementation here so I can fit more of the `ScalaFunction` definition onto this slide. I'm just showing a comment that represents the actual logic.



scala.rb: Scala Function Wrappers from Ruby Procs



```
scala
Function1

def andThen[A](g: (R) => R)(f: (R) => R)
  (f andThen g)(x) == g(f(x))

def apply(v1: T1): R
  attributes: abstract

def compose[A](g: (A) => T)(f: (T) => R)
  (f compose g)(x) == f(g(x))

def toString(): String
  Returns a string representation of the function
```

```
module ScalaProc
  class ScalaFunction
    def initialize(delegate)
      @delegate = delegate
    end

    def apply(*args)
      # invoke @delegate with args
    end
  end

  for n in 0..22
    eval "\
class Function#{n} < ScalaFunction
  include Scala::Function#{n}
end"
  end
end
```

To support `Procs` with different arity, the code highlighted in this slide dynamically defines a different class to correspond to each of the 23 function traits defined in the Scala source. The code will create a `Function0` class, a `Function1` class ... and so on, up to `Function22`.

I'm using the green arrow to point out that these `Function0-22` classes are subclasses of `ScalaFunction`, which, as we just discussed, wraps the `Proc` the you pass to its constructor and implements `apply` to invoke the block linked to the `Proc`.

These `Function0-22` are the `Proc` wrappers that you can pass to Scala code that expects a Scala function. Now that we've seen that these wrappers inherit `apply` from `ScalaFunction`, we'll look at how Daniel's `scala.rb` library handles a function's concrete methods.



scala.rb: Scala Function Wrappers from Ruby Procs



scala



```
def andThen[A](g: (R) => R)
  (f andThen g)(x) == g(f(x))
```

```
def apply(v1: T1): R
  attributes: abstract
```

```
def compose[A](g: (A) => B)
  (f compose g)(x) == f(g(x))
```

```
def toString(): String
  Returns a string representation
```

module ScalaProc

```
class ScalaFunction
  def initialize(delegate)
    @delegate = delegate
  end

  def apply(*args)
    # invoke @delegate with args
  end
end
```

for n in 0..22

eval "\

class Function#{n} < ScalaFunction

include Scala::Function#{n}

end

"

end

end

The classes `Function0` through `Function22` support the concrete methods implemented by their corresponding function traits by virtue of a dynamically generated `include` statement (ie “`include Scala::Function#{n}`”).

The green arrows from each of the concrete methods (`andThen`, `compose` and `toString`) to the `include` statement represent `scala.rb`’s support for Scala mixins for JRuby.

JRuby’s `include` keyword does not provide any support for concrete methods defined for Scala traits without some enhancements from Daniel’s `scala.rb` library.

Daniel opens the `Module` class and redefines the `include` method to enable Ruby developers to mix in concrete methods defined on a Scala trait much the same way that they are able to mix in methods defined on a Ruby module.

Daniel knows that the Scala compiler generates classes that implement a trait’s concrete methods. Because he knows the naming conventions for these generated classes, he can dynamically define methods that use Java reflection to invoke these generated implementations, and he can dynamically add these method definitions to the class or module that the `include` statement targets.



scala.rb: Scala Function Wrappers for Ruby Procs

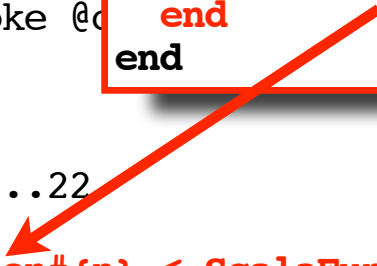


```
module ScalaProc
  class ScalaFunction
    def initialize(delegate)
      @delegate =
    end

    def apply(*args)
      # invoke @delegate
    end
  end

  for n in 0..22
    eval "\
class Function#{n} < ScalaFunction
    include Scala::Function#{n}
end
"
  end
end
```

```
class Proc
  def to_function
    eval "ScalaProc::Function#"
  end
end
```



Here's where Daniel opens up the `Proc` class to add the `to_function` definition, which returns an instance of the `Function0-22` class that wraps the `Proc` and exposes all of the methods exposed by a Scala function with the same arity as the `Proc`.



Collections, Parentheses, Parameters & Apply

```
scala> val ages = Array(17.0, 7.0)
res0: Array[Double] = Array(17.0, 7.0)
scala> ages(0)
res1: res1: Double = 17.0
```

Invokes



```
ages.apply(0)
```

The final feature of Daniel's library that we're going to look at is its support for accessing and populating Scala collection elements using square-bracket notation as opposed to the Scala's standard parentheses-based notation.

We saw earlier that calling `apply` on an `Array` and passing it a number returns the element at that index number, and we saw that as an alternative syntax, you can place the index number flanked by parentheses next to the `Array` name to likewise get the element at that index number. I explained how the parentheses syntax actually triggers an `apply` invocation.

The next few slides survey Scala's use of parentheses for working with collection elements, where many other languages, including Ruby, use square brackets.

The `Array` on the top of this slide is the `Array` of ages of programming languages I've used in previous examples. The first element, 17, represents how old Ruby is. The second element represents how many years Scala has been around.



Collections, Parentheses, Parameters & Update

```
scala> val ages = Array(17.0, 7.0)
res0: Array[Double] = Array(17.0, 7.0)
scala> ages(0)
res1: res1: Double = 17.0
scala> ages(0)=18
```

Invokes

```
ages.update(0, 18)
```

Just as placing a number in parentheses next to a collection name triggers a call to `apply` on the collection, the “`()=`” syntax invokes the `update` method on the collection. The collection’s `update` method sets the specified element value.

The `update` method takes 2 parameters: an index number and a value for the element at that index number (or if the collection is `Map`-like vs. `Array`-like, `update` takes a key/value pair).

To assign a value to an element in an `Array` or `Array`-like collection, you can specify the index in parentheses next to the name of the collection instance on the left-hand side of an equals sign and specify a value for the element at that index number on the right-hand side of the equals sign. For example the following expression, sets the first element of `ages` to 18: `ages(0)=18`.

For a `Map`-like collection, you specify a key/value pair with the key inside the parenthesis on the left-hand side of the equals sign and a value to associate with that key on the right-hand side of the equals sign.



Collections & Square-Bracket Notation

```
scala> val ages = Array(17.0, 7.0)
ages: Array[Double] = Array(17.0, 7.0)

scala> ages[0]
<console>:1: error: identifier expected but integer
literal found.
    ages[0]
         ^

scala> ages[0]=18
<console>:1: error: identifier expected but integer
literal found.
    ages[0]=18
         ^
```

As you can see from the errors in this Scala console session, using square-bracket notation to get or set collection values is not valid Scala syntax.



Getting & Setting Scala Mutable Collection Values from JRuby



```
> ages = Java::ScalaCollectionMutable::ListMap.new
=> #<Java::ScalaCollectionMutable::ListMap:0x3a0db598>
> ages("Ruby")=17
SyntaxError: (irb):5: syntax error, unexpected '='

ages("Ruby")=17
              ^
    from (irb):5
> ages("Ruby")
NoMethodError: undefined method `ages' for main:Object
    from (irb):6
irb(main):007:0>
```

In this `jirb` session, I've created a Scala `ListMap`.

As you can see from the console errors, parentheses-based syntax for getting and setting collection values (ie “`()`” and “`()=`”) is not valid in JRuby. This is because JRuby is not wired to convert those patterns into `apply` or `update` calls.



Getting & Setting Scala Mutable Collection Values from JRuby



```
> ages = Java::ScalaCollectionMutable::ListMap.new
=> #<Java::ScalaCollectionMutable::ListMap:0x3a0db598>
> ages["Ruby"]=17
NoMethodError: undefined method `[]=' for
#<Java::ScalaCollectionMutable::ListMap:0x2278e185>
  from (irb):2
> ages["Ruby"]
NoMethodError: undefined method `[]' for
#<Java::ScalaCollectionMutable::ListMap:0x2278e185>
  from (irb):3
```

In this `jirb` session, I show how using square-bracket notation to populate or get values from Scala `ListMap` is likewise not supported by JRuby.



Getting & Setting Scala Mutable Collection Values from JRuby



```
> ages = Java::ScalaCollectionMutable::ListMap.new
=> #<Java::ScalaCollectionMutable::ListMap:
0x3a0db598>
> ages.update("Ruby", 17)
=> nil
> ages.apply("Ruby")
=> 17
```

So, when you work with Scala collections in JRuby without Daniel's library, the only way you can get and set values in those collections is via their `apply` and `update` methods, as I've done in this `jirb` session.

I used `update` to pair the key "Ruby" with the value 17 (its age), and then I've used `apply` to obtain the value paired with the key "Ruby" (17).



scala.rb : support for []?



Mutable Collections

```
> require 'scala.rb'
=> true
> ages = Java::ScalaCollectionMutable::ListMap.new
=> #<Java::ScalaCollectionMutable::ListMap:
0x3a0db598>
> ages["Ruby"]=17
=> 17
> ages["Ruby"]
=> 17
```

In this slide, I show how I requiring Daniel's library enabled me to use square-bracket notation in conjunction with a Scala `ListMap`.

He added this feature to his library to make working with Scala collections in JRuby feel more 'natural'. Arguably, `ages["Ruby"]=17` is more Ruby-like than `ages.update(0, 17)`.



scala.rb : support for []?



```
alias_method :__old_method_missing_in_scala_rb__,  
             :method_missing  
def method_missing(sym, *args)  
  
  ...  
  
  if str == '[]'  
    eval(gen_with_args.call('apply', args))  
  elsif sym.to_s == '[]='  
    # doesn't work right  
    eval gen_with_args.call('update', args)  
  else  
    __old_method_missing_in_scala_rb__(sym, args)  
  end  
end
```

Here's the part of the scala.rb source that implements support for using [] = and [] to get and set Scala collection values from JRuby.

The lines highlighted in red show where `method_missing` is structured to invoke `apply` when it detects “[]” and `update` when it detects “[]=”

But as Daniel points out in his blog post and as a comment in his code (notice the comment “#it doesn't work right” sandwiched between the fragments highlighted in red), there are some issues with feature.



scala.rb : support for []? Mutable Collections



```
> ages = Java::ScalaCollectionMutable::ListMap.new
=> #<Java::ScalaCollectionMutable::ListMap:
0x3a0db598>
> ages.update("Ruby", 17)
=> nil
```

```
> require 'scala.rb'
=> true
> ages = Java::ScalaCollectionMutable::ListMap.new
=> #<Java::ScalaCollectionMutable::ListMap:
0x3a0db598>
> ages["Ruby"] = 17
=> 17
```

There is a subtle problem with using “[]=” in lieu of `update` with a mutable collection, and a not-so-subtle problem when you’re working with an immutable scala collection

If we take a close look what happens when you use `apply` vs. “[]” in an `irb` session, we can see that the two expressions have different return values, which represent two very different ways of thinking about populating data structures.

The expression using `update` evaluates to `nil`, while “[]=” returns the value passed to it.

In Scala, the method signature for `update` (`def update(key: A, value: B): Unit`) indicates that its return type is `Unit`. The `Unit` type in Scala is defined as a type that has only one potential value: `()`. A method that returns a `Unit` type in Scala is comparable to a method in Ruby that returns nothing.

In Scala, an action that produces side-effects (eg changing the value of a data structure at a particular index) will return `Unit` as opposed to a meaningful value to underscore that its side-effect is its *Raison d'être*.

In Ruby, “[]=” is classified as an “assignment-like” statement, and as such, it will always return the value on the right-hand side of the expression



Considering Assignment-like Setters in Ruby

martian.rb

```
class Martian
  attr_reader :age

  def age=(earth_age)
    @age= age / (686.98/365.26)
  end
end
```

```
> require './martian.rb'
=> true
> martian = Martian.new
=> #<Martian:0x192878>
> martian.age = 17
=> 17
> martian.age
=> 9.038720195638883
```

Here's an example that shows how Ruby evaluates an "assignment-like" expression when a setter is overridden to modify the value it gets passed.

The Martian class that defines an assignment-like setter (`age=`) to assign the Mars years equivalent of its argument to `@age`. If we call the setter in an `jirb` session, we can see that the expression, "`martian.age=17`" evaluates to 17, which is the value passed to it, even though the final expression in "`age=`" method evaluates to 9.038720195638883.

You might reasonably expect "`martian.age=17`" to return 9.038720195638883. But Ruby, which is so easily customizable in so many ways, is hard-wired to always return the expression on the right-hand side of the equals sign for an assignment-like setter.

Just as Ruby ensures that "`age=`" will return whatever is on the right-hand side of the equals sign, JRuby ensures that "`[]=`" will return whatever is on the right-hand side of the equals sign, even though Daniel's library intercepts the "`[]=`" call in method missing to invoke `update`, which does not return anything.



Considering Assignment-like Setters in Ruby

martian.rb

```
class Martian
  attr_reader :age

  def age=(earth_age)
    @age= age / (686.98/365.26)
  end
end
```

```
> martian_1 = Martian.new
=> #<Martian:0x197968>
> martian_2 = Martian.new
=> #<Martian:0x19675c>
> martian_1.age = martian_2.age = 17
=> 17
> martian_2.age
=> 9.038720195638883
> martian_3.age
=> 9.038720195638883
```

Why does Ruby behave his way?

Consider Ruby's support for parallel assignment. If you wanted to create twin Martians, you could do something like the code highlighted in red in this `jirb` session.

If “`martian_2.age=17`” were to return 9.038720195638883 instead of 17, then “`martian_1.age = martian_2.age = 17`” would pass 9.038720195638883 as the earth age to “`martian_1.age =`”, thereby setting the age of `martian_1` to 4.80579192794413. In other words, if “`martian_2.age=17`” were to return 9.038720195638883 instead of 17, then `martian_1` and `martian_2` would not be the same age following the parallel assignment statement.



scala.rb : support for []?

Immutable Collections



```
> ages = Java::ScalaCollectionImmutable::ListMap.new
=> #<Java::ScalaCollectionImmutable::ListMap:
0x1b6b7f83>
> ages.update("Ruby", 17)
=> #<Java::ScalaCollectionImmutable::ListMap::Node:
0x6446154e>
```

```
> require 'scala.rb'
=> true
> ages = Java::ScalaCollectionImmutable::ListMap.new
=> #<Java::ScalaCollectionImmutable::ListMap:
0x2d95bbec>
> ages["Ruby"]=17
=> 17
```

While using “[]=” instead of `update` in a `scala.rb`-enhanced JRuby program doesn’t work quite right for mutable Scala collections, it’s virtually unusable for immutable collections.

Consider the the immutable `ListMap` I’m creating in this `jirb` session. A immutable `ListMap`’s `update` method returns a version of the collection with the supplied key pointing to the supplied value. It does not modify the collection it is invoked on.

Mirroring the Scala behavior for the `update` function, the `update` invocation on `ages` in the `jirb` session in this slide, returns an immutable `ListMap`. Calling “[]=” on the other hand, returns 17.

In terms of human language, I’d say that mapping “[]=” to a `update` for a *mutable* collection is like leaving out connotations or nuances in a translation. It’s similar to translating “venerable” as “old”.

However, I think using “[]=” in lieu of `update` for an *immutable* collection is comparable to a patently bad translation. It’s like translating “a textbook case” literally as “a covering for a text book” instead of as “a classic example.”



scala.rb : support for []? Immutable Collections



```
> require 'scala.rb'
=> true
> ages = Java::ScalaCollectionImmutable::ListMap.new
=> #<Java::ScalaCollectionImmutable::ListMap:0x6039a07>
> ages.send("[]=", "Ruby", 17)
=> #<Java::ScalaCollectionImmutable::ListMap::Node:
0x42d6f628>
```

Incidentally, if you use `send` to invoke “[]=” on a Scala collection in a `scala.rb`-enhanced JRuby program, the whole expression will match the return value of `update` and not the value of the second argument to `update`.

In other words, the expression highlighted in red will return a `ListMap` that includes the key/value pair specified in the `send` call -- and not 17.

But “`ages.send("[]=", "Ruby", 17)`” is arguably even less natural than “`ages.update("Ruby", 17)`”.

Really, I don’t think having to call `update` on a Scala collection is all that bad. In my opinion, being able to use dot notation to invoke methods written in another language “phrase book style” constitutes a good interop experience.

Overall, I think `scala.rb` is a successful prototype. The only reason I spent some time on the less-than-ideal square-bracket feature is that it was a points to some interesting differences between Scala and Ruby.



Dynamic Language Runtime(DLR) Common Language Runtime(CLR)

The path to Ruby and F# integration goes through IronRuby, the .NET implementation of Ruby.

The .NET platform facilitates language interoperability between its Common Language Runtime (CLR) languages, like F#, and its Dynamic Language Runtime (DLR) languages, like IronRuby. Languages implemented on top of either the DLR or the CLR both compile to .NET IL (Intermediate Language).

When I started doing research for this presentation, Microsoft was payrolling an IronRuby development team. I was not aware the Microsoft was starting to cut back on the amount of resources it was devoting to IronRuby development. Currently no one is paid to develop IronRuby fulltime. For more details about the status of IronRuby, see the following blog posts by IronRuby core team member Jimmy Schementi: <http://blog.jimmy.schementi.com/2010/08/start-spreading-news-future-of-jimmy.html> and <http://blog.jimmy.schementi.com/2010/10/leadership-of-ironruby-and-ironpython.html>



F# Hosting Ruby

Mars.fs

```
#light

open IronRuby

let engine = Ruby.CreateEngine()
let scope = engine.CreateScope()

engine.Execute("
def mars_age(age)
  age / (686.98/365.26)
end
", scope) |> ignore

let marsAge = engine.Operations.InvokeMember(scope,
  "mars_age", [ |box 17.0| ] )

printfn "%A" marsAge
```

The DLR Hosting API is similar to the mechanism that enables you to embed JRuby scripts in IronRuby code. Here is how the same `mars_age` function defined in the `mars.rb` file we loaded into a Scala program would be loaded into a Windows program and invoked.

We get a handle on the scripting engine, load the Ruby code via `Execute`, and use `InvokeMember`, passing it the function name and an array of arguments.



IronRuby Hosting F#



Venus.fs

```
namespace Venus
module Calculations =
    let venusAge (age:double) : double =
        age * (365.26/224.68)
```



```
>>> require "FSharpCore.dll"
=> true
>>> require "Venus.dll"
=> true
>>> Venus::Calculations.venusAge(17.0)
=> 27.6367277906356
```

Calling F# functions from IronRuby is not unlike calling methods on Scala classes from JRuby.

The top half of this slide shows the F# source file Venus.fs, which contains an F# version of the `venusAge` function we have seen in Scala and Erlang. In an F# function definition, any arguments to an F# function are specified in parenthesis following the function name, with a colon and the argument type appended to each argument name. A colon separates the argument list from the return type specifier.

I used the F# compiler to generate a Dynamic Link Library (DLL) called Venus.dll.

I used Ruby `require` statements to load FSharp.dll, which contains the most commonly used FSharp routines and Venus.dll.

In the IronRuby `irb` session on the bottom of the slide, I inserted a double colon (`::`) between the namespace (`Venus`) and the module name (`Calculations`) to create a fully qualified reference to the `Calculations` module. I could then access the `venusAge` function using the dot (`.`) notation, and pass its parameter between parentheses.

Handling Tuples in IronRuby



```
namespace Venus
module Calculations =
  let venusAndCatAges(age:double) : double * double =
    venusAge(age), age * 6.2
```

Venus.fs



```
>>> Venus::Calculations.venusAndCatAges(17.0)
=> (27.6367277906356, 105.4)
```

In addition to making phrase book-style calls to IronRuby library routines, you can access and manipulate F# data structures from your IronRuby code, provided you know the particulars of the IronRuby syntax for the particular data structure.

The next few slides are about with handling tuples -- ordered, comma-delimited lists, often used to group related items.

The `venusAndCatAges` function shown on the top half of the slide returns a two-element tuple in which both elements are `doubles`, with the first representing the supplied age on Venus and the second representing the supplied age in cat years. The asterisk in the return types specifier is not the symbol for multiplication. Rather it is used to delimit types in the return tuple. The return type for a function returning a pair of tuples -- a triple (tuple with 3 elements) consisting of two integers and a double and a pair consisting of a string and a double -- would be: `(int * int * double) * (string * double)`

Looking at how `irb` evaluates the return value of `venusAndCatAges` (ie `(27.6367277906356, 105.4)`) on the bottom half of the slide -- it looks like you should be able to utilize Ruby's parallel assignment support to assign values to ...

Handling Tuples in IronRuby



```
namespace Venus
module Calculations =
  let venusAndCatAges(age:double) : double * double =
    venusAge(age), age * 6.2
```

Venus.fs



```
>>> Venus::Calculations.venusAndCatAges(17.0)
=> (27.6367277906356, 105.4)
>>> venus_age, cat_age = Venus::Calculations.venusAndCatAges(17.0)
=> [(27.6367277906356, 105.4)]
```

...populate a `venus_age` variable and a `cat_age` variable in a single statement, as shown in the line highlighted in red in the slide.

As you can see, the session does not choke on this statement, however, as we will see in the next slide, you can't always map a construct from one language to a construct in another language because they appear to be comparable. It would be like making the mistake of assuming that "dove" in Hebrew is a kind of bird. It's actually the word for "bear."

Handling Tuples in IronRuby



```
namespace Venus
module Calculations =
  let venusAndCatAges(age:double) : double * double =
    venusAge(age), age * 6.2
```

Venus.fs



```
>>> Venus::Calculations.venusAndCatAges(17.0)
=> (27.6367277906356, 105.4)
>>> venus_age, cat_age = Venus::Calculations.venusAndCatAges(17.0)
=> [(27.6367277906356, 105.4)]
>>> venus_age
=> (27.6367277906356, 105.4)
>>> cat_age
=> nil
```

If we let `irb` evaluate `venus_age` and `cat_age`, we can see that `venus_age` received the value of the entire tuple, while `cat_age` is `nil`.

Handling Tuples in IronRuby



```
namespace Venus
module Calculations =
  let venusAndCatAges(age:double) : double * double =
    venusAge(age), age * 6.2
```

Venus.fs



```
>>> ages = Venus::Calculations.venusAndCatAges(17.0)
=> (27.6367277906356, 105.4)
>>> ages.item1
=> 27.6367277906356
>>> ages.item2
=> 105.4
```

This slide shows how to access the individual items in the tuple returned by `venusAndCatAges`. Here, I'm assigning the return value of `venusAndCatAges` to `ages`.

You can access the first element by invoking `item1` on a tuple, and the second by using `item2`. If `venusAndCatAges` returned a three-item tuple, you would be able to access the third item by invoking `item3` on the tuple, and so on.

F# Discriminated Union Example

Luis Diego Fallas: 'Some Notes on Using F# Code from IronPython'

<http://langexplr.blogspot.com/2009/01/some-notes-on-using-f-code-from.html>

```
type MathExpr =  
    | Addition of MathExpr * MathExpr  
    | Subtraction of MathExpr * MathExpr  
    | Literal of double
```

Now we'll look at an example of how a more complex construct defined in F#, a discriminated union, can be used in a Ruby program.

The sample F# code comes from a blog post by Luis Diego Fallas called "Some Notes on Using F# Code from IronPython". Using F# from IronPython is similar enough to using F# from IronRuby that I was able to translate most of his sample code pretty easily. I like his discriminated union example because it is small enough to fit on a couple of slides, but it still gives you an idea of how a discriminated union could be used to navigate any sort of tree structure.

Before we look at how to use the `MathExpr` discriminated union from IronRuby, I'll show you how to use it in F#.

`MathExpr` is an F# discriminated union definition that describes 3 types of `MathExprs` (ie `Addition`, `Subtraction` and `Literal`) in terms of other `MathExprs` and standard types.

F# Discriminated Union Example

Luis Diego Fallas: 'Some Notes on Using F# Code from IronPython'

<http://langexplr.blogspot.com/2009/01/some-notes-on-using-f-code-from.html>

```
type MathExpr =  
    | Addition of MathExpr * MathExpr  
    | Subtraction of MathExpr * MathExpr  
    | Literal of double
```

```
let ten = Literal(10.0)  
let sum = Addition(Literal(1.0), Literal(2.0))
```

In the code fragment highlighted with a green background, I'm creating a couple of `MathExpr`s, using the type names almost the way you would use constructors -- with argument lists that mirror the type definitions.

I pass a double to `Literal` to create a `Literal`. An `Addition MathExpr` takes two `Literals`.

F# Discriminated Union Example

Luis Diego Fallas: 'Some Notes on Using F# Code from IronPython'

<http://langexplr.blogspot.com/2009/01/some-notes-on-using-f-code-from.html>

```
type MathExpr =  
    | Addition of MathExpr * MathExpr  
    | Subtraction of MathExpr * MathExpr  
    | Literal of double
```

```
let sum = Addition(Literal(1.0), Literal(2.0))
```

```
let rec mathEval (m:MathExpr) =  
    match m with  
    | Addition(m1,m2) -> mathEval(m1) + mathEval(m2)  
    | Subtraction(m1,m2) -> mathEval(m1) - mathEval(m2)  
    | Literal(m1) -> m1
```

Typically developers write a function like the `mathEval` function highlighted with a green border to evaluate a type defined via discriminated union.

Here's a step-by-step description of how the `mathEval` function evaluates `sum`:

First, `mathEval` finds that `sum` matches the `Addition` clause. Then `mathEval` hits the expression linked to the `Addition` clause, which invokes `mathEval` with `sum`'s first argument as an argument (ie `Literal(1.0)`). The call to `mathEval(Literal(1.0))` matches the `Literal` clause, which simply returns the `Literal`'s only argument, which, in this case is 1.0. Next `mathEval` is invoked with `sum`'s second argument as an argument, which yields, 2.0. The `+` indicates that the result of the `mathEval(Literal (1.0))` and `mathEval(Literal(2.0))` should be added together, so `sum` evaluates to 3.0.

Discriminated Union Example



Venus.fs

```
namespace Venus
module DU =
  type MathExpr =
    | Addition of MathExpr * MathExpr
    | Subtraction of MathExpr * MathExpr
    | Literal of double
```



```
>>> Venus::DU::MathExpr.methods
=> ['new_literal', 'new_addition', 'new_subtraction',...]
```

If we include the `MathExpr` definition in a module called `DU` in `Venus.fs` and package it in `Venus.dll`, we can access it in IronRuby by prefixing it with `Venus::DU`.

On the bottom part of the slide, I'm showing some of what `irb` displays when you invoke `methods` on `Venus::DU::MathExpr`. IronRuby provides a method prefixed with “new_” for each alternative type defined in the discriminated union.

Discriminated Union Example



Venus.fs

```
namespace Venus
module DU =
  type MathExpr =
    | Addition of MathExpr * MathExpr
    | Subtraction of MathExpr * MathExpr
    | Literal of double
```



```
>>> one = Venus::DU::MathExpr.new_literal(1.0)
=> Venus.DU+MathExpr+Literal
>>> two = Venus::DU::MathExpr.new_literal(2.0)
=> Venus.DU+MathExpr+Literal
>>> sum = Venus::DU::MathExpr.new_addition(one, two)
=> Venus.DU+MathExpr+Addition
```

Here I've used these "new_" methods to create some `MathExpr`s in the IronRuby console -- a couple of `Literals` and an `Addition`.

In the next slide I'm going to show some of the methods that get displayed in the console when you invoke methods on `sum`, one of the `Addition MathExpr`s I created.

Discriminated Union Example



Venus.fs

```
namespace Venus
module DU =
  type MathExpr =
    | Addition of MathExpr * MathExpr
    | Subtraction of MathExpr * MathExpr
    | Literal of double
```



```
>>> sum.methods
=> ['get_Item1', 'get_Item2', 'get_IsLiteral',
'get_IsSubtraction', 'get_IsAddition',...]
```

These are some of the methods that `sum` exposes that can be used to construct a Ruby-based `eval` method similar to the one we looked at in F#.

Discriminated Union Example



```
let rec mathEval (m:MathExpr) =  
    match m with  
    | Addition(m1,m2) -> mathEval(m1) + mathEval(m2)  
    | Subtraction(m1,m2) -> mathEval(m1) - mathEval(m2)  
    | Literal(m1) -> m1
```



```
def r_math_eval(expr)  
    if expr.get_IsAddition  
        r_math_eval(expr.get_Item1) + r_math_eval(expr.get_Item2)  
    elsif expr.get_IsSubtraction  
        r_math_eval(expr.get_Item1) - r_math_eval(expr.get_Item2)  
    else  
        expr.get_Item  
    end  
end
```

Here's a side-by-side comparison of the F# version of the `MathExpr` evaluator, `mathEval`, and an IronRuby version, `r_math_eval`.

Discriminated Union Example



```
>>> one = Venus::DU::MathExpr.new_literal(1.0)
=> Venus.DU+MathExpr+Literal
>>> two = Venus::DU::MathExpr.new_literal(2.0)
=> Venus.DU+MathExpr+Literal
>>> sum = Venus::DU::MathExpr.new_addition(one, two)
=> Venus.DU+MathExpr+Addition
>>> def r_math_eval(expr)
...   if expr.get_IsAddition
...     r_math_eval(expr.get_Item1) + r_math_eval(expr.get_Item2)
...   elsif expr.get_IsSubtraction
...     r_math_eval(expr.get_Item1) - r_math_eval(expr.get_Item2)
...   else
...     expr.get_Item
...   end
... end
=> nil
>>> r_math_eval(sum)
=> 3.0
```

Here's the `irb` sequence that shows a `sum` being evaluated using `r_math_eval`.

Creating an List

```
let ages = 17.0 :: 1.92 :: 5.0 :: []
```

Creating the Same F# List from IronRuby

```
class Array
  def to_fslist_of_floats
    L = Microsoft::FSharp::Collections::FSharpList[Float]
    result = L.get_Empty
    reverse.each { |x| result = L.cons(x, result) }
    result
  end
end

ages = [17.0, 1.92, 5.0].to_fslist_of_floats
```

Sometimes you can say something in one language that takes a paragraph to explain in another language.

An F# list is a non-modifiable data structure. The expression on the top half of the slide looks like a simple list expression with two colons (: :) being used as delimiters in lieu of commas. Actually the two colons (::) constitute the `cons` operator, which is used to append a value to the head of a list in F#.

Creating an F# list in IronRuby is an example of something that you can express succinctly in one language, but that involves jumping through some hoops in another. This slide is based on code IronRuby core team member Tomas Matousek kindly sent me when I was initially having trouble figuring out how to work with an F# list in IronRuby.

The bottom part of this slide adds a method to `Array` that converts an `Array of Floats` to an F# list. As you can see, creating a `Microsoft::FSharp::Collections::FSharpList` involves a lot more than passing a collection of values to a constructor.

I'll break it down step by step: We start with an empty list, by calling `getEmpty` on `L`, which is an `FSharpList`. Then we `reverse` the `Array`, because we are going to construct the F# list using the `cons` method (the method invoked by the `cons` operator (: :)), which appends values to the head of a list. Then we need to loop through the values in the `Array` in reverse order, appending each to the head of the list to create a list with the same contents as the `Array` in the same order.

Language-Neutral Approaches



For the most part I've talked about polyglot programs in terms of phrase books that translate one specific language into another. Before concluding I'd just like to touch on the concept of language-neutral approaches.

In terms of human languages, a language-neutral approach might involve using universal symbols. No matter what languages you speak, the sign in this slide indicates that you should expect to see flying saucers in the vicinity.



Language-Neutral Approaches



Protocol Buffers

<http://code.google.com/p/protobuf/w/list>

Apache Thrift

<http://incubator.apache.org/thrift/>

BERT & BERT-RPC

<http://bert-rpc.org/>

This slide lists 3 projects that embrace language-neutral approaches to language integration.

The first two, Protocol Buffers and Apache Thrift, rely on an interface definition language (IDL). You define an interface for a service indicating the service name, the inputs the service expects and the outputs the service returns using the IDL syntax. Code generators translate the IDL into routines that can be called by clients or services to send or receive data in a variety of languages. Using the same IDL you can generate client code in one language and service code in another.

Where Protocol Buffers and Apache Thrift define an IDL, BERT-RPC is build around a binary-level protocol. The BERT-RPC home page lists serializers that can translate BERT binaries to and from Erlang, Ruby and Scala as well as serializers that support other languages.

ACKNOWLEDGEMENTS

Thanks Very Much

Mario Camou
Killy Draw
Dave Fayram
Daniel Kwiecinski
Tomas Matousek
Daniel Spiewak
Brian Takita
Jonathan Tran
Chuck Vose
Ezra Zygmuntowicz

CREDITS

Blog Post Titles on Slide #6:

<http://blogs.msdn.com/b/tess/archive/2009/04/03/developers-are-from-mars-ops-people-are-from-venus-or-it-looked-good-on-paper.aspx>

<http://m.zdnet.com/blog/collaboration/engineers-are-from-mars-marketers-are-from-venus/367>

<http://blog.datamation.com/blog/2005/10/cios-are-from-m.html>

“First Steps to Scala” Quote on Slide #17:

<http://www.artima.com/scalazine/articles/steps.html>

Erlang Reference Manual Excerpt on Slide #30:

<http://www.erlang.org/doc/man/erlang.html>

Erlang Reference Manual Excerpt on Slide #32:

http://ftp.sUNET.se/pub/lang/erlang/doc/man/erl_eval.html

RUBY IS FROM MARS, Functional Languages Are from Venus

Integrating Ruby with Erlang, Scala & F#

ANDREA O. K. WRIGHT
Chariot Solutions
aok@chariotsolutions.com

The source for all of the examples can be found at: <https://github.com/A-OK/RubyIsFromMars>