# IronRuby: white paper

*“A language that doesn't affect the way you think about programming is not*

*worth knowing.”*

**- Alan Perlis**

## What is IronRuby?

IronRuby is an implementation of the Ruby language on the .NET Framework. That means when it’s complete it will have the same language features as Matz’s Ruby Implementation (MRI) 1.8.6 but backed up by the intrinsic power that the .NET Framework harnesses.

Yukihiro “ Matz” Matsumoto publicly released the Ruby programming language towards the end of 1994. It took a while for the language to reach outside of Japan but it was growing slowly. In 2005 David Heinemeier Hansson released the Rails project – written in Ruby –, which became very popular and well-known among web developers at first.

Matz wrote the original Ruby in C and since then there have been a lot of extensions in C contributed to the Ruby language in the form of Gems. People familiar with the CLR can think of a gem as an assembly or a library.

In other words when gems or libraries that don’t depend on C extensions but are purely implemented in the Ruby language will also run on IronRuby. The gems that have C-extensions will need to have the C-extension ported to a .NET language. You have the full power of the CLR at your disposal, which could potentially greatly simplify the porting of the extension and perhaps ultimately result in a better library.

*Actually, I'm trying to make Ruby natural, not simple.*   
-- **Yukihiro “Matz” Matsumoto**

## Why should I care?

Ruby is a pleasant language to work with, partly because its well-chosen keywords, but mostly because it represents a kind of best breed implementation of a programming language. One of the goals ruby tries to achieve is to make the language you use when you’re writing your algorithms is a lot like a natural language. For example ruby has both the keywords if and unless, and you can apply those in 2 different ways.

if some\_affirming\_condition  
 collection.collect do |item|  
 [item.id, item.name]  
 end  
end

unless some\_negating\_condition  
 collection.collect do |item|  
 [item.id, item.name]  
 end  
end

do\_some\_work if true

do\_some work unless true

Unless does the opposite of what if does. So unless is a pretty way of writing if not. Ruby also has the notion statement modifiers, which is what the 2 last examples show.

### Best of breed

The Ruby language has built-in regular expression literals and has very strong text processing support very much like the kind of support you might find in Perl. Another language feature that is most interesting is probably blocks and iterators and those are in turn borrowed from Smalltalk. And so far we’ve only begun to scratch the surface of niceties the Ruby language has to offer.

*If Python was the result of Lisp and C++ having a baby,  
Ruby is the result of Perl and Smalltalk having a baby.***--MeowMeow Jones, 11/8/2001 on slashdot.**

One of the more interesting concepts of the language is probably blocks. When we call yield the program executes code that was defined somewhere else and bound to that lexical context than in the function and got implicitly passed into the method.

class MyCollection

def initialize(items=[])

@items = items

end

def each

for item in items

yield item if block\_given?

end

end

end

collection = MyCollection.new [1, 4, 5, 6, 7, 9]

collection.each { |item| puts “current item: #{item}” }

# Outputs the following:

# current item: 1

# current item: 4

# current item: 5

# current item: 6

# current item: 7

# current item: 9

The code above actually also demonstrates iterators. In Ruby you generally use iterators to process enumerable objects like arrays etc. You do this by passing a function that will be called on each iteration of 1 item on that item. Some other iterator functions that Ruby has to offer are collect, select, inject, map, reduce,…

### Duck Typing

Ruby is classified as a dynamically typed language. This means that it cares more about the behavior of an object than about it’s type hierarchy. It does care about the type hierarchy, including mixins, when it does method lookup. It doesn’t mean it’s weakly typed because it only allows “safe” operations against an object ie. i = 1 + “1” .to\_i. Conversions between types have to be done explicitly. This doesn’t mean that ruby will stop you to override the operators on the classes to allow for implicit typing, but at that point you are in control of what’s happening. Ruby’s typing system is sometimes referred to as duck typing. The introduction of the term duck typing has to be followed by: – “If it looks like a duck and acts like a duck it might as well be a duck” –

So (Iron)Ruby is a dynamically but strong typed language. This may confuse people that have been working with static languages all their lives. Let’s look at this with an example:

a\_variable = 5  
a\_variable = “IronRuby rocks!”

The above statements are perfectly valid in Ruby. To some people it might look like we changed the variable *a\_variable* from type, but what we actually did was change the binding of the name/label *a\_variable* from an integer object to a string object. Ruby won’t allow you however to change the type of an integer object without an explicit conversion step; in this conversion step it will create a new object and not change the shape of the original object, and that is what makes it strong typed. The fact that Ruby defers checking for a type until the last responsible moment and that it allows you to add methods etc to types at run-time is what makes it dynamically typed. And this brings us to duck typing.

This presents a different outlook on programming. In a CLR language like C# or Visual Basic .NET a type is basically the same as a class. And prior to .NET 3.5 we would have had to define *a\_variable* like this:

string a\_variable = “IronRuby rocks!”;

and now with C# 3.0 you could write:

var a\_variable = “IronRuby rocks!”;

Both have the same effect from then on a\_variable will always have to be of the type System.String. While in Ruby we don’t care about that type that much at all. In Ruby classes are a way to initially group methods together but when you are actually running the program ruby doesn’t really care about the type but more about the methods that are on the type.

def get\_count(item)  
 item.size  
end

string\_var = “IronRuby rocks!”  
array\_var = [1, 3, 5, 6, 7]

get\_count(string\_var) # result: 15  
get\_count(array\_var) # result: 5

In the above example we define a method *get\_count* with one parameter *item* and it will return the number of characters or the number of items in a collection. As long as the object contained by the variable *item* has a method size this method will execute properly and return the value returned by *item.size*.

Another way of explaining the difference in the typing systems would be to say that a language like C# has a more aristocratic view on a class because the heritage and its family is very important for the compiler as opposed to a language like Ruby that has perhaps a more realistic view on an object because it is based primarily on ability rather than heritage.

### Flexibility

Ruby is also a very flexible language that supports many programming styles. It’s designed as an object oriented language but its flexibility allows for it to be used in a procedural style defining only methods The following language rules support the functional style: *all* expressions have a return value (including “if” statements), and *all* methods return the result of the last statement. There are more things in ruby to support this functional programming style, we’ll take a look at them shortly.

*For me the purpose of life is partly to have joy. Programmers often feel joy*

*when they can concentrate on the creative side of programming, So Ruby is*

*designed to make programmers happy.*

- **Yukihiro “Matz” Matsumoto**

#### Procedural programming

Procedural programming occurs when you define all your functions in the global namespace. For Ruby this global namespace is Object. To load a file, show its contents on the screen and then renaming it to have an extension of .read could be written like this in a procedural style:

require “ftools”

def open\_file(name)

File.open name

end

def read\_contents(file)

counter=1

while l = file.gets

puts "#{counter}: #{l}"

counter = counter + 1

end

end

def close\_file(file, name)

file.close

File.move(name, "#{name}.read")

end

def read\_file(name)

file = open\_file(name)

read\_contents(file)

close\_file(file, name)  
end

read\_file("todo\_list")

The above code is about as procedural I could come up with aggregated in the method – or should I say procedure – *read\_file*. What happens here is that we are actually defining all these methods in the global namespace of *Object* in Ruby. The next example shows the proof for the previous statement. There is a reason for that to which we’ll return later but in ruby all method calls have a receiver and the most basic receiver is *Object*.

# remember the original instance methods on object

original\_methods = Object.instance\_methods

# define a method test in the global namespace

def test; puts “inside test method”; end

# print a list of the new instance\_method names on Object

puts (Object.instance\_methods – original\_methods).sort.join(“\n”)

# outputs: test

# Create a new instance of object and call test on it

Object.new.test

#### Functional programming

Ruby also has support for a functional style of programming. An important feature the functional style of programming needs, are closures. Closures are blocks of code that capture variable bindings when it is created. It can then use those variables when it executes. To create a closure in ruby you have to use the keyword *lambda* and pass it a block. There are more constructs for passing chunks of code around in the language but they don’t have a purpose in this discussion. The important thing to know right now is that lambda’s are the only way to get true closures in Ruby. The earlier discussed blocks don’t behave in the same way as a lambda does with respect to the return keyword.

def closure(var\_val)

a = “Some text”

lambda { puts “#{a} #{var\_val}” }

end

closure1 = closure(“is in here”)

closure2 = closure(“is different”)

closure1.call

closure2.call

# Outputs the following:

#

# Some text is in here

# Some text is different

As you can see the lambda that is returned is bound to the instance variables of its enclosing scope. And that’s way we get 2 different results for our 2 created closures.

## How could I use it?

Ruby takes away much of the monkey work programmers have to deal with and like Smalltalk, Lisp, etc. Ruby belongs to a class of languages that allow for runtime evolution of the program. This is referred to as metaprogramming, we’ll take a closer look at the necessary concepts behind this shortly.

*The computer should be doing the hard work. That's what it's paid to do, after all.*  
**-- Larry Wall**

### Metaprogramming

People speak of metaprogramming when they can get the program to modify itself at runtime. That means when you’re using metaprogramming you’re defining methods on instances and classes at runtime. You could very well generate most of a program at runtime.

Metaprogramming is often used when the data drives the code. In addition to some of the concepts you need for this style of programming there are also a couple of other tools that Ruby has to offer to help you with the task of metaprogramming. You tell Ruby what you want it to do and Ruby will happily do the heavy lifting for you and generate all the model classes from a database schema for example. The first tool Ruby has to offer is responding to undefined methods, the famous *method\_missing*, a second tool that ruby has to offer is defining methods programmatically. The third tool that ruby in its arsenal is string evaluation. We won’t discuss these tools now, I just wanted to mention them because they are an important part of the Ruby language.

*I don’t think the idea here is to save memory or speed. The idea behind metaprogramming is to teach Ruby your conventions and let it do some guessing, in order to save you some code.* – **Why, author of why’s poignant guide to Ruby**

#### Classes are open

One of the core features of Ruby is that classes are open for modification at run-time. You are free to add methods, variables and so forth to them. You can change entire classes or just instances of a class. Below is an example of opening the existing string class and adding a method to\_lolcat\_haz to it. Later we’ll call that method.

class String

def to\_lolcat\_haz  
 "I haz #{self.gsub(/s$/,'z')}"  
 end

end

puts "major props".to\_lolcat\_haz  
# Outputs: I haz major propz

The code above defines a new method on the class String and that method changes the existing string by putting *I haz* in front of it and changing the ending letter s of any words in the string to z.

#### All method calls have a receiver

Ruby is built around the idea of message passing. Whenever you call a method on an object you are actually sending a message to the object. Ruby implements this idiom explicitly because it has a method send on the Kernel that is available on any object (it’s a module included in Object). To continue our example from above, we wrote:

puts "major props".to\_lolcat\_haz

Instead we could have written:

send('puts', "major props".send(:to\_lolcat\_haz))

Where send is defined on Object so that makes Object the receiver of this message. This means that invoking methods through their string names is very easy. Methods that don’t have a receiver are invoked on the current object.

#### Everything is an object

In Ruby everything is an object. This means that nil, the value that signifies no value is an object, albeit a special one. You can instantiate the Class class, which is the reason why instances of objects are instantiated the way they are in Ruby (*Song.new*). The .new method is just a class method on the object Class. There are no primitives like int and char in ruby; everything is an object.

*I invented the term Object-Oriented, and I can tell you I did not have C++ in mind.*  
**-- Alan Kay**

>>> 0.zero?

=> true

>>> 5.zero?

=> false

>>> 120/6

=> 20

>>> 873.class

=> Fixnum

>>> (1000 \*\* 1000).class

=> Bignum

>>> a = nil

=> nil

>>> a.nil?

=> true

>>> a.class

=> NilClass

## Great! That was mostly Ruby how about the Iron?

So far we’ve talked about what makes Ruby such a great language but the subject of this paper is **Iron**Ruby so what does the Iron add to this great language. The Iron stands for the fact that this is an implementation of the Ruby language on the .NET Framework. So in addition to having this great and powerful language at your disposal you also get the inherent benefits that come with using the .NET framework.

Your IronRuby application will run on every environment where the .NET framework can run. Because mono is around and mono runs on a bunch of platforms and devices you have an incredible reach for your app. With this there are some limitations like mono doesn’t support WPF at the moment so you won’t be able to use that when running IronRuby on Mono.

You do get some of the other benefits of using .NET like the generational garbage collector, threading, good integration with the windows OS and in IIS.  
So far much of the Ruby fame is thanks to the Ruby on Rails framework when you use .NET you get a whole bunch of other frameworks with that too.

One of the things IronRuby will immediately be able to do for you is simplifying your unit test code and especially mocking objects will become a lot easier when you use a dynamic language.

### Closed CLR classes now open for business

We’ve seen earlier how classes are open in Ruby. We hope you can also see the possibilities this offers to you as a developer. Well now the classes that were previously closed in the CLR like System.String or System.Object are now open and can be extended at will.

Let’s start by something fluffy and safe, we’ll open the class *System::String* and add a method *to\_lolcat\_haz* to it, just like in the example above.

class System::String

def to\_lolcat\_haz

"I haz #{self.gsub(/s$/,'z')}"

end

end

s = “major props”

s.class # ==> String

s.to\_lolcat\_haz # Throws a NoMethodError

clr\_string = s.to\_clr\_string

clr\_string.class # System::String

lc = clr\_string.to\_lolcat\_haz

puts lc

# Outputs the following

#

# I haz major propz

You’re not limited to just adding methods to you can override all methods in the class. At this point I think it would be wise to mention the great words: “With great power comes great responsibility, use it wisely!” That being said, let’s move on to an example to prove this point. Why don’t we open up *System::Object* and override the method *to\_string* in that class and see where that gets us.

>>> require 'mscorlib'

=> true

>>> class System::Object

... def to\_string

... "Overriding oh my"

... end

... end

=> nil

>>> System::Object.new.to\_string

=> "Overriding oh my"

>>> System::Collections::Generic::List.of(System::String).new.to\_string

=> "Overriding oh my"

There is some pretty awesome power there that might scare some people of. I could find a good quote around this subject. A hypothetical question by Piotr Kochanski: “*Isn't this dangerous? What if someone changes the logic of the + operator for math expressions*. No, because if one of your programmers overrides methods that break things - you take them out in the parking lot and beat them with a rubber hose! The language shouldn't prohibit us from doing powerful things.” ([source](http://raibledesigns.com/rd/entry/oscon_monday_morning))

### All ur libraries are belong to us!

This subject works both ways because .NET comes with its set of libraries but the ruby community is a vibrant one and they also have a whole bunch of libraries.

*I have always imagined that Paradise will be a kind of library.***-- Jorge Luis Borges**

#### What .NET brings to the table

The .NET Framework contains a Base Class Library (BCL) these libraries drastically simplify tasks like creating GUI applications, Webservices or web applications. They also make lower level tasks vastly easier like sending emails, threading, …

If .NET developers have to agree on something then I think it would on the fact that the .NET framework is huge. In addition to the technologies like Silverlight etc. it has a whole bunch of libraries that make it easier to do performance counting. .NET has a very granular system for allowing code to run on a computer called Code Access Security. Applications like MS Office and the likes expose .NET interfaces and extension points so they become a lot more accessible through IronRuby from an automation point of view.

You will maybe find that there is a library in python for example, the same conditions apply it cannot rely on a C-based extension unless that extension has been ported to IronPython too. And this library is the bees knees, it’s all you need and dreamt of to solve your current problem. In that case you can use the DLR infrastructure to load the python library and use it from ruby as if it was a ruby object. In my opinion that is some pretty good stuff right there.

#### Of course Ruby is already at the table

The Ruby side of this then introduces a bunch of ruby libraries that have the potential to make developing applications on the CLR a little bit more fun. The library RSpec springs to mind. Because IronRuby is Ruby running on .NET you are able to use existing ruby tools with .NET applications. RSpec makes Domain Driven Development easier.

One of the really nice things you get almost for free when you use a dynamic language is that mocking becomes vastly easier. You don’t have to use a big mocking framework that does all kinds of black magic to trick the compiler into thinking that the object you are creating is actually of said type. We’ve talked about how in Ruby classes are open and how Ruby embraces duck typing then it should come to no surprise that mocking is really easy with ruby. Below is an example that shows how a couple of tests for a class that shortens urls through an online service could be tested.

describe "SniprUrl" do

before(:each) do

@snipr = SniprUrl.new "flanders.co.nz/my-long-url-folder/my-long-url-file"

end

it "should ensure that the url starts with at least http" do

@snipr.ensure\_http.should == "http://flanders.co.nz/my-long-url-folder/my-long-url-file"

end

it "should build a snipr request url" do

@snipr.build\_request\_url.should == "http://snipr.com/site/snip?r=simple&link=http://flanders.co.nz/my-long-url-folder/my-long-url-file"

end

it "should fetch a shortened url for a given long url" do

expected\_url = "http://snipr.com/x3di2"

mock\_request = Object.new

mock\_stream = MemoryStream.new(expected\_url.convert\_to\_bytes)

mock\_response = Object.new

mock\_response.should\_receive :get\_response\_stream, :returning => mock\_stream

mock\_request.should\_receive :get\_response, :returning => mock\_response

@snipr.should\_receive :request, :returning => mock\_request

@snipr.shorten.to\_s.should == expected\_url

end

it "should return the original url if it's shorter than the snipr one" do

@snipr = SniprUrl.new "flanders.co.nz"

@snipr.shorten.to\_s.should == "http://flanders.co.nz"

end

end

The most important test here is the one that describes “should fetch a shortened url for a long url”. That test contains mocks to avoid having to rely on a running webserver or network connection to run the test.

If the combination of Ruby and .NET would be a type of ninja I think it would be the sensei ninja or Chuck Norris for that matter ☺.

### Technology galore

If you put a bunch of the .NET libraries together you can see some groups emerging. We’ve got the BCL with libraries for Winforms applications as well as Windows Presentation Foundation, Silverlight/Moonlight, ASP.NET, Windows Communication Foundation,…

When you use IronRuby you can program those technologies with Ruby. This makes it easy to create very compelling applications on a variety of platforms. There is a caveat because Mono doesn’t currently support WPF so WPF is a windows only technology for the time being. Below is an example of creating a simple browser application with IronRuby and WPF.

p "loading components, please wait..."

require 'wpf'

require 'wpf\_elements'

class WpfApplication < Application

def initialize()

run yield

end

end

start\_url = "http://www.ironruby.net"

title = "Biffy"

WpfApplication.new do

# build the window

obj = Wpf.build Window, :title => title, :height => 500, :width => 826, :name => "Biffy" do

add DockPanel, :name => "dock\_panel" do

add TextBlock, :text => title, :font\_size => 36, :background => :alice\_blue, :dock => :top, :name => "text\_block"

add StackPanel, :orientation => :horizontal, :dock => :top, :name => "stack\_panel" do

add TextBox, :text => start\_url, :width => 750, :name => "web\_url"

add Button, :content => "Show site", :name => "get\_url\_button"

end

add Frame, :source => start\_url, :name => 'web\_page\_display'

end

end

# attach the event handler

obj['get\_url\_button'].click do

obj["web\_page\_display"].source = obj["web\_url"].text

end

obj

end

The example presented above shows a possible improvement IronRuby can make over using another imperative language like C#. It has the benefit that because of its very malleable structure you can make it look like almost anything you want. DSL’s are very much in vogue in the ruby community. What makes this code work is all contained in the lines require ‘wpf’ and require ‘wpf\_elements’. I took those files from a code sample that John Lam wrote for a [demo](http://www.iunknown.com/2007/11/ironruby-on-sil.html). These 2 files contain all the code that uses metaprogramming techniques to convert that to the same imperative commands you would have write in C#.

### Some more benefits

As Microsoft adds performance enhancements and functionality to the .NET framework you get these improvements completely free without any updates to your code.

Ruby 1.8.6 doesn’t have native OS threads but instead time slices the execution of the different threads at well-known points. This has as side-effect that an error in one thread could potentially kill your application. With IronRuby you can take advantage of the more advanced CLR threading mechanism that does take advantage of native OS threads and errors in one thread don’t necessarily affect the other threads.

## Conclusion

Ruby tries to assist the programmer to focus more on the creative side of programming instead of having him deal with the scientific side of it. It does so by implementing a mixture of language features that can be found in other older languages.

It is a very flexible language that now brings all its power to the .NET Framework. Developers armed with a good knowledge of .NET and Ruby can be vastly more productive than they were in the past. The .NET framework itself comes with a wealth of libraries and technologies that will make it even easier to create great applications.

In addition to programming applications IronRuby can also help a lot in terms of unit testing existing CLR applications as well as for automating tasks and or building applications. IronRuby is definitely a very welcome addition to the .NET languages and if the ruby language is new to you I hope it will change the way you think about programming.

*go\_to(‘http://ironruby.net’).get(ironruby).play.create.have\_fun*  
**--Adapted from Tobias Reif (.signature), 16/8/2001, ruby-talk**