

Is Ruby the new Java?

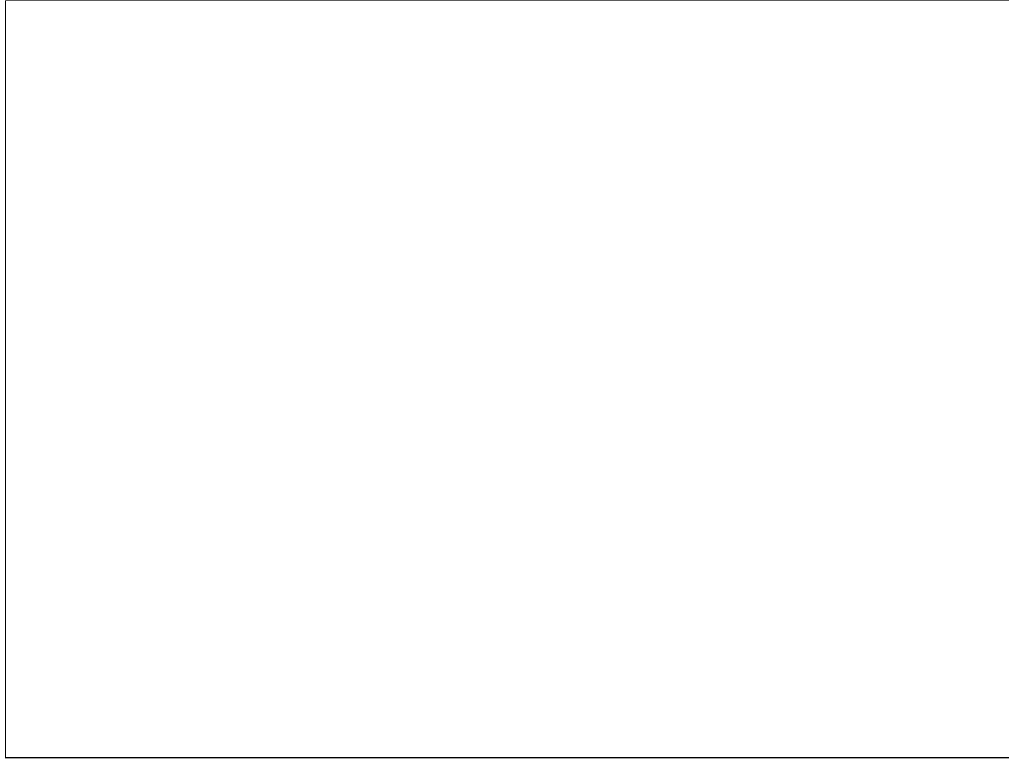
Is Java the new C++?

Is C++ the new Cobol?

Is Basic the new Cobol?

Is Java the new Basic?

Is Ruby the new PHP?



No, I'm not going to answer these questions. Is Ruby the new Java? Who knows? Who cares? A question which, I think, is much more interesting, is: *why* are people considering Ruby to be the next Java? What is so special about Ruby?

Today, I will show you a little bit of what Ruby is, what Ruby on Rails is, and what they might hold for you, as Java developers. So who am I to tell you?

Danny Lagrouw

PROF ICT

My name is Danny Lagrouw. I work for a company called Profict (www.profict.nl). We're specialized in doing application integration projects in Java, for clients all over the country.

I wrote my first computer program back in 1982, and I've loved coding ever since. For the last six years, I've been mainly doing Java projects, either as team lead or as developer.

```
5.times { print "Hello world!" }
```

```
exit unless "restaurant".include? "aura"
```

```
['toast', 'cheese', 'wine'].each { |food| print food.capitalize }
```

Like a lot of people, I first heard about Ruby somewhere last year. There were all these discussions going on on the ServerSide, about Ruby being better than Java, being Java's successor... So at first I thought: this is probably some hype, Java is cool, back to work. But a co-worker gave it some more attention, and he showed me things, code, that was really... interesting. It looked so simple, so clean, so elegant. I started playing around with it myself, and I quickly got hooked.

And now, here I stand, spreading the word. Not just because I believe that, as Java developers, we can benefit from Ruby and Rails, but also because I like Ruby, I'm enthusiastic about it, and I want to show you why.

(Code samples are from Why's (Poignant) Guide To Ruby (<http://poignantguide.net/ruby/>), an illustrated introduction to Ruby and chunky bacon.)



The story begins in 1993, when there was an economic crisis in Japan, and someone called Yukihiro Matsumoto, a.k.a. Matz, wanted to create something that would make him happy. And so he created a programming language called Ruby. That's right, 1993, two years before Java came to life. That means that Ruby is by now a reasonably mature language and has lots of extension libraries. It did take quite some time however, before Ruby became popular outside Japan.

Ruby

object oriented

interpreted

dynamic

DRY

So what is Ruby like?

Ruby is completely *object oriented*. It was set up as an object oriented language from the very first moment. As we'll see in a moment, you might even say that Ruby takes object orientation a step further than Java.

Ruby is an *interpreted* language. The first thing people ask when you say 'interpreted' is: how about performance? And they're right. You probably won't be using Ruby for a time-critical application.

Because Ruby is interpreted, so not statically compiled, Ruby can also be a *dynamic* language. Dynamic meaning: a Ruby program can alter itself during its execution.

And something you will often see connected with Ruby is the term *DRY*. We'll come to that in a moment.

Java

```
class PetStore {
    Collection removeFleas(Collection pets) {
        FleaSpray fleaSpray = new FleaSpray();
        for (Iterator it = pets.iterator(); it.hasNext(); ) {
            Pet pet = (Pet) it.next();
            fleaSpray.applyTo(pet);
        }
        return pets;
    }
}
```

This is a simple code sample, in Java, that we'll use to demonstrate some of Ruby's features. It's about a pet store which offers many services besides just selling pets. For example, here's a method that will remove the fleas from a given collection of pets. It uses a fresh new can of flea spray that will be applied to each of the pets in turn. Afterwards, the pets are returned to their owners.

That looks simple enough, but we can do even simpler... in Ruby.

Ruby

```
class PetStore
  def remove_fleas(pets)
    flea_spray = FleaSpray.new
    pets.each do |pet|
      flea_spray.apply_to pet
    end
    pets
  end
end
```

How do we do the same thing in Ruby? Here you see. It already looks less verbose than the Java code, don't you think? To begin with, we can skip some things in Ruby.

Like semi-colons; they're optional.

Parentheses for method calls are usually optional too.

There's no return statement. Ruby always returns the result of the last expression executed. So in this method, it returns the `pets` collection.

Those are just small things, although they come in handy as we'll see later on.

How about type declarations? There are none, as you can see. There's a `pets` collection coming into the method with no type specification. So how does the code know what type `pets` is? Well, it doesn't. It simply asks the `pets` variable to execute a method called 'each'. Ruby assumes that if you want to do an 'each' method on this `pets` object, that you will provide a `pets` object that is able to handle an 'each' method. We call this 'duck typing': if the object walks like a duck and talks like a duck, then it must be a duck.

What happens if you send it an object that is not a collection? Obviously you won't get a compiler error, as Ruby is interpreted. At runtime, Ruby will try to execute the 'each' method on `pets`. If it's not there, a special method will be called, called 'method_missing'. That will raise a runtime exception. You can however override the `method_missing` method, to handle the unknown method call more gracefully.

If situations like these don't raise compiler errors, won't your code be more error-prone? Well, yes, obviously. For that reason, there's a lot of emphasis on test-driven development in Ruby and Rails, in order to weed out potential bugs by unit testing.

(continued on next slide)

Ruby

```
class PetStore
  def remove_fleas(pets)
    flea_spray = FleaSpray.new
    pets.each do |pet|
      flea_spray.apply_to pet
    end
  end
end
```

Now, about this construction in the heart of the method. “pets.each do |pet|...”. It looks like some kind of loop... and it is. In Ruby we consider looping through the items in a collection to be the responsibility of the collection. Using an iterator construction like we do in Java, assumes that you have knowledge about the collection’s implementation. Maybe this collection must iterate its elements backwards. Maybe it’s a special collection where some of the elements must always be skipped. We don’t know. We don’t want to know. That’s the collection’s responsibility. All we want to do is give it a block of code and say: execute this for each of your elements. A block of code in Ruby is enclosed by do/end, or by accolades. The block can receive a number of parameters; in this case, it receives one element at the time in the pet variable. Inside the block, you can manipulate the pet any way you want.

Java

```
List names = new ArrayList();
for (Iterator it = pets.iterator();
     it.hasNext(); ) {
    Pet pet = (Pet) it.next();
    names.add(pet.getName());
}
```

Ruby

```
names = pets.map { |pet| pet.name }
```

Ruby's way of looping offers more elegant ways of doing things with collections. Consider this example. We want to extract from the `pets` collection, a new collection containing just the names of the pets. So a new `String` collection with just the names. In Java we have to declare a new collection, iterate over the elements, retrieve each element from the iterator, then add the name of the pet to the new names collection.

In Ruby we just call the `map` method. The `map` method iterates over the collection's elements, just like the `each` method; but `map` takes the result of the block of code that you pass, and puts that into a new collection. Finally, that new collection is returned into the `names` variable.

**Don't
Repeat
Yourself**

Now I told you that Ruby is all about staying DRY. DRY means: Don't Repeat Yourself. And that means not just in *your* own code, but in the way you can use Ruby as well.

Java

```
class Pet {  
    private String name;  
    private Double weight;  
    String getName() {  
        return name;  
    }  
    void setName(String name){  
        this.name = name;  
    }  
    Double getWeight() {  
        return weight;  
    }  
    void setWeight(Double weight){  
        this.weight = weight;  
    }  
}
```



A great example is the way Ruby encapsulates instance variables. This is how we do that in Java: we type in the instance variables, we press Alt+Shift+S, R in Eclipse, and our getters and setters are generated for us. Works great, until you change the type of a variable, but okay.

One of the first things people ask me when I tell about Ruby is: which IDE can I download to write Ruby code? Is there an Eclipse plugin? And this, code generation, refactoring, are reasons why people, usually Java people, want to use an IDE. If I tell them that you don't particularly need an IDE, I usually detect a little panic. And I understand. I wouldn't want to code Java without all these handy shortcuts and templates and tricks either. It's just too much work. But let's look at how Ruby does things.

Ruby

```
class Pet
  attr_accessor :name, :weight

  def name
    @name
  end
  def name=(s)
    @name = s
  end

  def weight
    @weight
  end
  def weight=(w)
    @weight = w
  end
end
```

This is it. This is what you do in Ruby to declare two instance variables, `name` and `weight`, *and* their encapsulation. I told you before that Ruby is a dynamic language. Your program can alter itself during execution. And that is exactly what will happen here. When this code will be executed, a *method* called `attr_accessor` is called. That method will insert getter and setter methods into your class—all still at runtime. This is the end result; but you will never see this code, as it's all done at runtime. Of course, if you want to write your own methods, because you need special coding, you can. There are also alternatives to `attr_accessor` that will only generate either a getter or a setter method.

Ruby

```
class PetStore
  def remove_fleas(pets)
    flea_spray = FleaSpray.new
    pets.each do |pet|
      flea_spray.apply_to pet
    end
    pets
  end
end
```

We go back to the store for two last features of Ruby I want to show you. This time we notice that a single spray is not enough to kill all the fleas. So what do we do?

Ruby

```
class PetStore
  def remove_fleas(pets)
    flea_spray = FleaSpray.new
    pets.each do |pet|
      5.times do
        flea_spray.apply
        wait_for 10.minutes
      end
    end
    pets
  end
end

class Integer
  def minutes
    self * 60
  end
end
```

We spray five times in a row, and wait for 10 minutes after each spray. As you can see here, even primitives are objects in Ruby, and they have methods that we can call. Also, if we need a method that's not there already, like '10.minutes', we can add it ourselves to the base class. That's right, classes in Ruby are never final; we can extend them ourselves.

By the way, do you see this code here in the middle? 5 times do: flea spray apply to pet; wait for 10 minutes... It's hardly programming language anymore, it's almost something a non-programmer could read and understand. Maybe even a business user... Well guess what, this is exactly why Ruby is becoming very popular to create...

Domain **S**pecific **L**anguages

A DSL is a programming language created for a specific problem domain. The idea is to create a language that can be used by people who have good knowledge of a specific domain, but do not necessarily know how to program. By creating a language that is similar to natural language, but can be interpreted as a programming language, we enable the people who know most about something, to code their knowledge into a program.

DSL in Ruby

```
publishing agreement dated '20-9-2005'  
with_author 'Joe W. Author', sofinr('555-493-3920')  
for_title 'DSLs for Dummies'  
  
report do  
  calculate 'Royalties', as net_retail_sales  
    .during(last_six_months) * 20.percent  
end
```

This is an example that I found on the internet (http://jroller.com/page/obie?entry=expressing_contract_terms_in_a). It shows a DSL for specifying business rules regarding publishing contracts and royalty payments. You can see how Ruby is so well suited for making DSL's: the loose syntax, no semi-colons that would confuse non-programmers, and its dynamic nature allowing to create the language structures that you need.



And it's language features like these that have also made possible... Ruby on Rails. Last month, we had the first big Dutch Ruby and Rails meeting. About 120 people showed up. When the audience was asked how many of them had heard of Ruby or Rails before 2005, only about five of them raised their hands. And that's because, even though Ruby originates from 1993, Rails was only conceived in 2004. And Ruby wouldn't be what it is today if it hadn't been for Rails.

In 2004, a young Danish web developer called David Heinemeier Hansson released the web framework Ruby on Rails, based on Ruby. By now, both Rails and David are all over the internet, and magazines like Wired and Dr. Dobbs Journal. What's so special about Rails?

Ruby on Rails

MVC

Convention over Configuration

Like I said, Rails is a web framework based on Ruby. It has an MVC architecture, and it's based on the principle of 'convention over configuration'. In Rails, we have conventions, strict programming standards, about things that are the same in 80% of standard web applications. If you follow these conventions, there's no need to configure those same things over and over again. There are no XML files in Rails. And all this is made possible by Ruby's language features, like the dynamic programming. Let me show you an example.

URL

`http://www.petstore.com/pets/list`

Controller

`pets_controller.rb`

```
class PetsController < ApplicationController
  def list
    @pets = Pet.find_all
  end
end
```

In Rails, we have a convention that a URL is always constructed from a controller name (here: `pets`) and an action name (`list`). If we request a URL `pets/list`, Rails goes looking for a controller named `PetsController`, and inside that controller, for an action named `list`. Here you see the action filling an instance variable (`@pets`) with all the pets from the datastore (we'll come to models in a second).

URL

`http://www.petstore.com/pets/list`

Controller

`pets_controller.rb`

```
class PetsController < ApplicationController
  def list
    @pets = Pet.find_all
  end
end
```

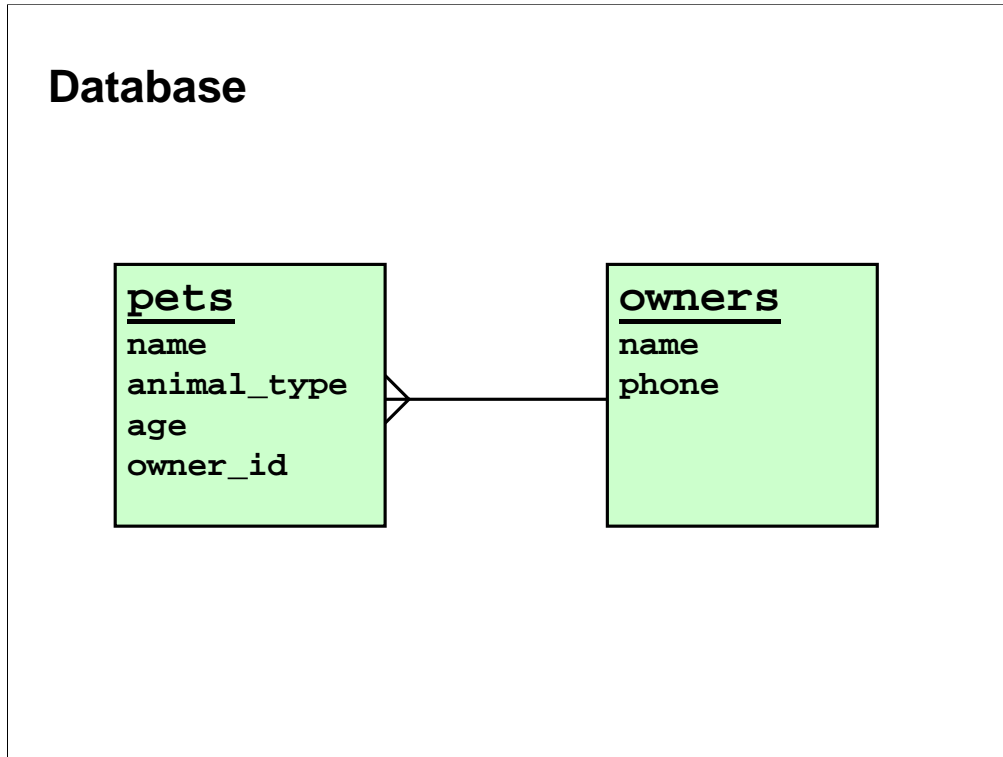
View

`pets/list.rhtml`

```
<ul>
  <% @pets.each do |pet| %>
    <li><%= pet.name %></li>
  <% end %>
</ul>
```

Also, if the list action is completed, Rails assumes you will want to show a view called list.rhtml in a pets subdirectory. The view can use any instance variable you have defined in the action, so this view lists all pets that the controller passed to it. That's really how easy it is. Likewise, you can have actions for things like editing, creating, deleting. But there's more to Rails. For instance, Rails has some very cool Ajax support. How long do you think it will take me to create a full CRUD application for those pets, complete with Ajax and object relational mapping? Let's find out.

Demo



Now I want to show you what a model class in Rails looks like. We've seen the controller and the view, so let's look at a model class. As you can see, it's quite... empty. Even so, all object relational mapping takes place in this class. Any idea why we don't see any database fields here? They're generated dynamically on the fly, as they're needed. By convention, a property in a model class will have the same name as the corresponding field in the database table. Just as the model class is named after the database table. If you need to deviate from this convention, you can by the way. Also in the model class we can add validations, like

```
validates_presence_of :name  
validates_uniqueness_of :name
```

Now what if we have a relation in the database between pets and their owners?

Model

```
class Pet < ActiveRecord::Base
  belongs_to :owner
end
```

```
class Owner < ActiveRecord::Base
  has_many :pets
end
```

```
owner.name = 'Martin Gaus'
pet.owner.name = 'Martin Gaus'
print owner.pets
```

This is how we model a relationship between two database tables in our model classes: `belongs_to` and `has_many`. After adding this, we can ask a pet for its owner's name, and ask an owner for a collection of his pets.



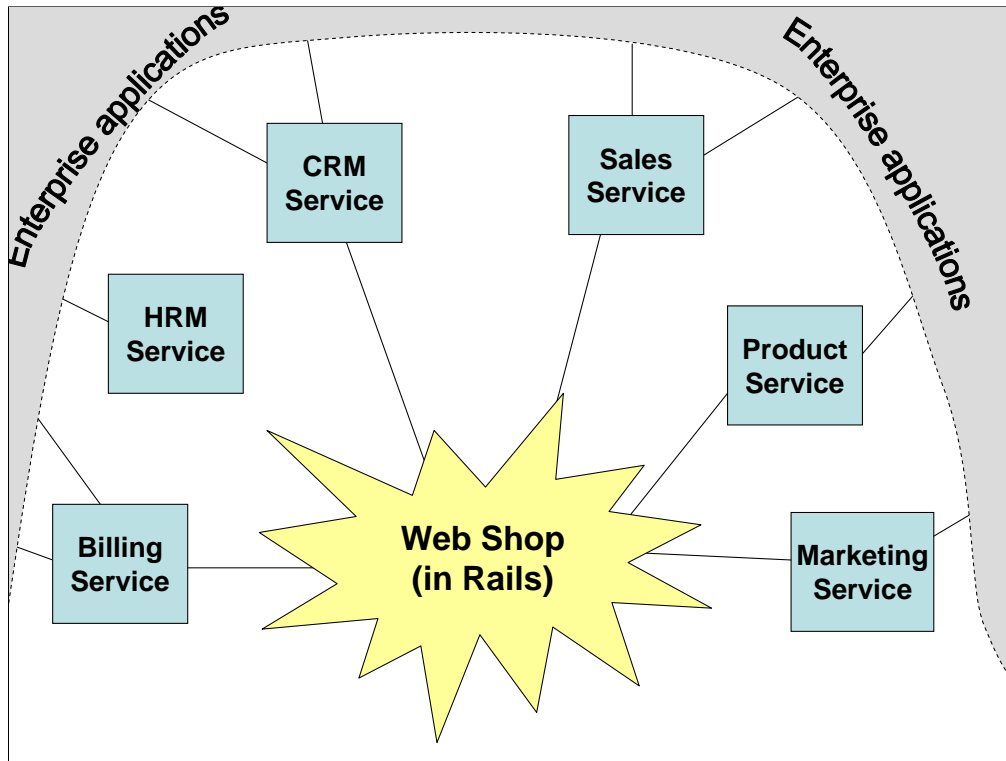
So much for Rails.

Now what does all this have to do with us, Java developers? We've invested so much time and effort in Java. Isn't that enough? There's enough work to do in Java. Why learn something new? Something that will probably not perform as well as Java. Something that should probably only be used to create simple web apps.

Personally, I think we have lots to gain from learning Ruby and Rails. For one thing, it might make you a better programmer. It might show you new ways of thinking, new ways of solving problems. Maybe we can use those in our day-to-day programming. Maybe we can use them to enhance Java itself. Look at the new EJB3 Persistence API, where you don't need to write XML configuration files anymore. Convention over configuration is entering the Java world already.

Or, we can use Ruby where it's most useful (best tool for the job), by running Ruby scripts from within Java code. And that's exactly what the new Scripting API is for.

Or... If you can develop web apps with Rails so easily, so fast... Should we accept that Rails might just be a better tool for developing web apps than Java? Could we take the strength of Java and Rails each, and combine them? For example...



...like so. In this setup, we write business services in Java, and publish them as web services or EJBs; and we create our web app with Rails, and have that call the services? To make that happen, we would need Ruby and Rails to be able to communicate with a JVM. Or better still, to run inside the JVM. That way, we wouldn't have to change the infrastructure that's already present. Ruby and Rails become just another tool in the Java toolbox, making it much easier to use in existing Java environments.

Fortunately, there's a very active group of people working on that Ruby/Java interface, called Jruby (www.jruby.org). At JavaOne they were able to demonstrate running a Rails application under JRuby. Rails already offers support for calling web services. The next step is calling EJBs from Ruby.

Demo

RubyEnRails.nl

`ruby.pagina.nl`

If you want to know more about Ruby and Rails, check out these two sites to begin with. RubyEnRails.nl is an all-Dutch blog portal, featuring the latest news about Ruby and Rails in the Netherlands and Belgium. Also, a good starting point for all sorts of information about Ruby and Rails is `ruby.pagina.nl`. It has links to manuals, documentation, books, weblogs, you name it.

danny@RubyEnRails.nl

ruby.pagina.nl

You're welcome to contact me by email if you have any further questions that haven't been answered today, at danny@rubyenrails.nl.

Java

```
c = a;
```

```
a = b;
```

```
b = c;
```

Ruby

```
a, b = b, a
```

Oh, and one more thing. If you ever hear people claim that developing in Ruby is 3 times faster than in Java... it's true!