David and Ruby: a twenty-first century love story

People ask me why I program in Ruby. Is it the clean syntax? the dynamism of objects? the low ratio of code lines to task accomplished?

And let's be clear about this: I do more than program in Ruby. I resigned from a tenured university professorship to pursue a life of Ruby programming, writing, and teaching. My professional life revolves around Ruby, and in an extended sense so does much of my personal life. This isn't a hobby or sideline. This is some serious stuff.

So when people ask me why I program in Ruby, I try to provide the most rigorous, robust, technically precise explanation I can. What I tell them is: love.

I fell in love with Ruby in early November of 2000, when the first edition of Programming Ruby by Hunt and Thomas (the “Pickaxe book”) had just come out. I spotted a copy of the book on the shelf of my local Borders bookstore. I squinted at the book's spine and thought, “Hmmm.” I've always been a bit of a language collector, so I was intrigued. I pulled the book off the shelf and opened it--and here I am, more than ten years later, still in love.

“Love” doesn't sound very technical. But technical or not, it's accurate. Moreover, it's why everyone who uses Ruby uses Ruby. I've never seen anyone convince anyone to use Ruby through rhetoric or argumentation. The language does all the courting.

That's not to say we don't have Ruby advocacy. Of course we do, and I'm heavily involved in it. But no one chooses a general-purpose programming language because of a check-list of features or a chart of comparisons with other languages. Ruby advocacy means creating opportunities for people to see Ruby up close, and creating conditions favorable to further love affairs.

I hope the ACM Ruby Learning Path will provide you with some walks through the park, late-night chats on the front porch, and other moments where love between you and Ruby may take hold and flourish.
A brief, highly selective Ruby chronology

- February 24, 1993: the birthdate of Ruby, according to its creator Yukihiro “Matz” Matsumoto. Matz comes to describe Ruby as optimized for programmer pleasure.

- 1993-2000: Ruby gains great popularity in Japan. Numerous books about Ruby are published. Ruby is still relatively unknown outside of Japan, but is being “discovered” by a few keen practitioners.

- October 2000: the first edition of *Programming Ruby* (called “the Pickaxe book” or “the pickaxe,” after its cover illustration) appears. Written by Dave Thomas and Andy Hunt, and published by Addison-Wesley, the Pickaxe introduces Ruby to a significant programmer population outside of Japan.

- October 2001: the first International Ruby Conference (“RubyConf”) is held in Tampa, Florida. Attendees include Matz, Dave Thomas, Andy Hunt, Jim Weirich, Nathaniel Talbott, and about thirty other Rubyists, many of whom were to become well-known practitioners, authors, and teachers in the Ruby world.

- 2001: Jan Arne Petersen releases the first version of JRuby, a Ruby interpreter written on the Java VM. JRuby would eventually be taken up by Sun and later by Engine Yard, and spearheaded by Charles Nutter and Thomas Enebo.

- November 2002: the second International Ruby Conference. Masayoshi Takahashi, a prominent Japanese Rubyist (for whom the “Takahashi Method” of slideset presentation is named) brings with him from Japan one copy of each Ruby book published there. Laid out on a table, there are twenty-two books. There are still only a handful of non-Japanese Ruby books in print.

- Summer, 2004: Danish programmer David Heinemeier Hansson releases his Ruby on Rails Web development framework, extracted from the Basecamp project of the company 37 Signals. David presents Rails at RubyConf 2004, and interest soars.

- Fall 2005: attendance at RubyConf triples between 2004 and 2005, due largely to the huge wave of Ruby interest generated by Rails.

- 2006: Evan Phoenix begins work on Rubinius, a Ruby interpreter based on the principle of writing as much of Ruby in Ruby as possible.

- December 25, 2007: Ruby 1.9.0 released, incorporating the YARV (Yet Another Ruby VM) bytecode generator by Koichi Sasada.

- 2010: Conferences, books, user groups, open-source projects, interpreters -- the Ruby world continues to thrive and to grow.
The uniquely global Ruby community

“Uniquely global” is perhaps an overstatement. But Ruby occupies an unusual position among technical projects and areas of interest.

Back in 2002, at the second International Ruby Conference (“RubyConf”), the prominent Japanese Rubyist Masayoshi Takahashi (he of the “Takahashi Method” fame) brought with him from Japan, in a cardboard box, one copy of each of the Ruby books then in print in Japanese. He laid them out on a table—all twenty-two of them. This was at a time when there were perhaps three or four Ruby books available in Western languages.

The sight of twenty-two Japanese Ruby books had a big impact on those of us who saw them in 2002. It helped us understand just how popular Ruby already was, even though it had only recently been “discovered” by the non-Japanese-speaking technical world. Those twenty-three books also raised important questions about “the Ruby community.” Just who was this community? Were there really multiple communities? And how could they all communicate?

Many Japanese programmers speak English; but very few non-Japanese programmers speak Japanese. The integration of the Ruby community around the world must be credited largely to the Japanese Rubyists who have participated on English-language mailing lists, attended English-language conferences (Takahashi has attended ten RubyConf in a row!), read and disseminated English-language writings on Ruby, invited English-speaking presenters to Ruby events in Japan, and generally shown the greatest imaginable collegiality in welcoming and supporting Ruby users around the world.

It leaves those of us in the West a bit in awe. And we know that there's much we don't know about Ruby in Japan; not every Japanese programmer participates in the English-language forums, and many resources (including Matz's blog) are in Japanese. We're just glad to be on board with this beautiful programming language and its thriving worldwide usership.
Ruby's thriving conference culture

Wherever you live, there's a good chance that a Ruby users group exists, and even a good chance that you're not too far from one or more annual Ruby conferences.

For reasons no one fully understands (but who needs to?), the Ruby world is full of conferences. The first and oldest such event is the International Ruby Conference (“RubyConf”), which has taken place annually since 2001, each time in a different city in the U.S. Other noteworthy annual events include:

- RailsConf
- EuRuKo, the European Ruby Conference (since 2004)
- GoRuCo, the Gotham (New York City) Ruby Conferences
- Lone Star Ruby Conference
- Mountain West Ruby Conferences
- Scottish Ruby Conference (previously Scotland on Rails)
- RubyConf Brazil
- Ruby Kaigi (Japan)
- The Ruby Hoedown
- Ruby East

and many more.

Even the smaller, ostensibly regional Ruby conferences often draw a significant crowd and feature world-class speakers. (Matz has attended Lone Star Ruby Conference as well as RubyConf, just to give one example.) The quality of Ruby Conferences tends to be extremely high. Often

The popularity of Ruby conferences is such that they've even started conflicting with each other. The community of organizers, who meet regularly via the Ruby Regional Conference Organizers Google group, negotiate dates and try to plan far enough in advance to avoid or at least minimize conflicts.

In 2006, Ruby Central, Inc., a non-profit organization chartered to support and produce events and projects in the Ruby world, started offering grants to organizers of regional Ruby conferences. The grant program kick-started several events, but most of them are self-sustaining based on registration fees and/or sponsorship. The Ruby Hoedown, a major Ruby conference held in the southeastern United States, charges nothing for attendance; it operates purely on sponsorship, and offers a first-rate speaker and presentation lineup.

Many of these conferences are small, often one-track events. Others are multi-track and go on for several days. Overall, the level of quality among Ruby conferences is consistently high. It's worth checking for one near you and making a point of attending it.
Ruby's object model: a starting point for understanding

Ruby falls somewhere between the strictly class-based object-oriented style of, say, Java, and the prototype, classless OO of JavaScript. Ruby isn't classless, of course; but it uses a class structure to implement a truly object-centric flavor of OO where classes are more of a useful shortcut than a strict architectural necessity.

Every Ruby object is an instance of a class; but Ruby object can live its own life, independent of any constraints imposed on it by its class. A simple example makes the point:

```ruby
string = "Hello"
def string.shout
  self.upcase + "!!!"
end
string.shout     # HELLO!!!
```

At this point, thanks to the singleton method `shout`, the string `string` can shout, but no other string can:

```ruby
"other string".shout   # NoMethodError
```

The ability for objects to diverge from their “birth class” constraints also applies to class objects. For example, the Regexp class (but not the Array class, nor Fixnum, String, etc.) has an escape method:

```ruby
Regexp.escape("abc.?")  # abc\.\?
```

Singleton methods on class objects, like `Regexp.escape`, are one way that Ruby implements class methods, very roughly the equivalent of static methods in C-like languages.

Ruby has a full toolkit for object individuation (divergence from the API of the object's original class), including the ability for individual objects to be extended from one or more modules, and a class-based interface to each object's singleton class, which contains method definitions for the object's singleton methods. The presence of classes, combined with the full complement of per-object behavior tools, makes Ruby a particularly supple and dynamic member of the object-oriented language family.
Ruby's sweet spots

The popularity of the Ruby on Rails Web development framework continues to bring waves of enthusiasts to Ruby and to help the language flourish. At the same time, Rails has led to some confusion about the range of Ruby's strengths. Like Perl in the 1990s, Ruby sometimes gets pigeon-holed as a Web-specific development platform. In fact (and also like Perl!) Ruby is a general-purpose language with many sweet spots and a fairly small number of weak areas.

Without meaning to constrain anyone from using Ruby in yet more areas, the following areas might be identified as sweet spots for the language, in addition to the “obvious” area of Web development:

- **Text processing.** Ruby has a strong toolkit for pattern-matching, string scanning, replacement, data serialization, and text manipulation of many kinds.
- **Administrative scripting.** Ruby's iterative structures are ideal for repetitive and/or slightly divergent tasks. Also, the language integrates a procedural style into its object-oriented architecture; if you want to write a quick-and-dirty script to rename a group of files, you can do so easily, or you can work up something more structured for better maintainability and easier reuse.
- **Code generation.** One of the first English-language books focused on Ruby was *Code Generation in Action* by Jack Herrington (Manning Publications, 2003), which used Ruby for its examples.
- **Prototyping.** Without any reduction in respect for Ruby as a production-code language, it's also true that Ruby makes it relatively easy and pleasurable to prototype complex programs that are destined to be written for production in other languages. By dispensing with variable typing and initialization, Ruby lets you focus on implementing logic right from the start (and you may end up using your “prototype” in production anyway!).
- **API wrappers.** With its straightforward C API—not to mention the Java connectivity of JRuby—Ruby makes life easy for those wanting to wrap existing libraries in a higher-level language.

It's only fair to list a couple of non-sweet spots, just in case:

- **Heavy number crunching.** Ruby has a full math library, of course; but if you're looking for the next prime number, you'll probably want something a bit lower-level.
- **Optimizations at the cycle-shaving level.** Ruby's performance has often been described, quite rightly, as “good enough.” Strides are being made, in JRuby and other interpreters as well as the original interpreter, toward greater performance. But Ruby's thorough dynamism comes at an irreducible price; it's never going to compete with, say, C at the cycle-shaving level.

Once you start using Ruby, you'll find yourself reaching for it more and more often—and for every time you decide to use a different language for a project, you may well choose Ruby for several others!
One language, many interpreters: What exactly is Ruby?

Once, there was just Ruby: a single interpreter, written and maintained by Matz, the creator of Ruby. Today there are numerous Ruby interpreters, including at least the following:

- **MRI** (“Matz’s Ruby Interpreter” -- the original Ruby, feature byte-code compilation based on Koichi Sasada's YARV VM since version 1.9)
- **JRuby** (Ruby on the JVM)
- **Rubinius** (an innovative approach to Ruby using a relatively small C++ kernel plus much of Ruby written in Ruby)
- **IronRuby** (Ruby for the .NET platform)
- **Ruby Enterprise Edition** (a performance-tuned Ruby based on the MRI codebase)
- **MacRuby** (“an implementation of Ruby 1.9 directly on top of Mac OS X core technologies” including the Objective-C garbage collector)

The proliferation of interpreters came about at a time (roughly 2007 onward) where Ruby was moving from version 1.8 to 1.9, with a possible 2.0 on the horizon and the status of 1.9 not always entirely clear. The question arose more and more urgently: what exactly is Ruby? Are we dealing with a Lisp-like situation, with many dialects rather than one standard? What does an interpreter have to do to be considered a Ruby interpreter?

The most important answer to this question came in the form of RubySpec, described on its webpage as an “executable specification for the Ruby programming language using RSpec syntax.” Here’s a sample of RubySpec, part of the specification of Ruby’s behavior with regard to parenthetical capture groups in regular expressions:

```ruby
describe "Regexp with grouping" do
  it 'support ()' do
    /(a)/.match("a").to_a.should == [%s", %s"]
  end

  it "allow groups to be nested" do
    md = /(hay(st)a)ck/.match('haystack')
    md.to_a.should == [%s',%s', %s']
  end
  # ...
end
```

RubySpec has gone a long way toward clarifying what a Ruby interpreter should do. At the same time, work is underway to make Ruby an ISO standard. Hopefully the standard and the test suites will operate in sync, and continue to make it clear what makes “a Ruby” a Ruby.

---

1 Ruby’s versioning scheme, through 1.8, had been that odd-numbered minor releases (e.g., 1.7) were development or experimental releases; even-numbered releases were production releases. This pattern was broken with 1.9, which was a production release—though 1.9.0 had some instabilities that made it not feel that way. The result was a certain amount of confusion about numbers and versions, though things stabilized again inside the 1.9 release series.

Ruby code-style conventions

Ruby lets you write code that looks like other languages, including a vaguely C-ish and/or procedural style:

```ruby
myNumber = 1;
if( myNumber == 1 )
begin
    printf( "You have set the variable to %d\n", myNumber );
end
end
```

That isn't good Ruby style, though. In general, it's best to follow conventions; make your code look as undistinguished as possible. Don't do things that most Rubyists don't do just because you can.

Here are a few guidelines to get you started. You'll see Ruby code that doesn't follow these guidelines, but they're based on the majority practice and will go a long way toward making your code blend in and keeping it clear.

- Indent two spaces (not tabs)
- Do not put semi-colons at the ends of lines! (They're for separating two or more statements on the same line, which you should do very rarely.)
- Use underscore_style (my_number) for local variables and method names
- Use CamelCase, with initial cap, for class and module names (MyClass)
- Use CAP_CASE for constants (DECK_SIZE = 52)
- Use parentheses for method arguments
- Omit parentheses for method arguments when there's a common idiom that doesn't use them (such as
  attr_accessor :name, where :name is the argument to attr_accessor)
- Use parentheses in method signatures: def concat(a, b) rather than def concat a, b

If you abide by these rules, you may find yourself in disagreement with some Ruby practitioners but you'll find generally that your code blends in nicely to the landscape.

And of course you'll find that even the authors of style guides don't always agree! But you'll get some good pointers from these guides and the links contained in them:

- http://www.caliban.org/ruby/rubyguide.shtml