Abstract of Rubyisms in Rails

By Jacob Harris

Rubyism is an examination of how the style of Ruby informs the design of Rails. In particular, it looks at a few specific examples of how Rails' internal code is implemented in Ruby to instruct about Ruby's design principles. The main goal is simply aesthetic appreciation. But, if you are a beginning programmer in Rails who is stymied in your understanding of Ruby or an intermediate Rails developer still writing code that looks like Ruby-tinged PHP or Java. This Shortcut will hopefully impart enlightenment and inspiration about the Ruby way of programming. It also reveals how the revolutionary design of the Rails framework can only be built upon the beauty of Ruby.

Preface

We will explore further but what the newcomer to Rails might miss is that Rails's design directly reflects Ruby's underlying beauty.

Rails's success is as much a measure of Ruby's expressive power as it is of Rails's conventions.

This digital shortcut is not a guide for learning Rails; you can already find many excellent books on that topic. However, all of them are designed to quickly drive their readers toward proficiency in Rails. They just do not have the luxury of exploring the Ruby foundation of Rails in any depth. Instead, they usually have to treat distinctive Ruby features things such as **symbols**, **blocks**, and **metaprogramming** in passing (if at all), like quaint towns and scenic overlooks glimpsed briefly from a car hurtling forward on an interstate highway. The problem here is that the beauty of Ruby gets short shrift. Therefore, many people who have come to coding in Ruby from Rails do not really get Ruby as much as they would like. That is where this digital shortcut comes in.

This is not an in-depth guide to the Ruby language, however. Instead, I share with you a few examples of code from within the Rails source code (and perhaps a few other places) that I think are beautiful.

The main goal is **aesthetic appreciation** and **philosophical alignment**, nothing more. However, I hope this approach affects you in subtler ways improving your understanding of why Ruby works and how you can become a better programmer in it.

The Philosophy of Ruby

> For me, the purpose of life is, at least 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 Matsumoto

Like Perl, Ruby is what Larry Wall termed a "**postmodern programming language**." This means that Ruby's creator, Yukihiro Matsumoto, created Ruby's syntax and features by deconstructing and recombining the functionality he admired the most in other existing languages. Perl was chief among his influences (hence Ruby's gemlike name), but useful features and functionality were also integrated from such languages as disparate as Ada, C++, CLU, Lisp, Java, Python, PHP, and Smalltalk. Despite all these wide-ranging influences, Ruby has a tightly consistent design in its syntax and libraries, reflecting the conscious effort of its creator to keep to guiding principles. In an introduction to the language from 2000, Matsumoto articulated three guiding principles that Ruby strives for:

1. **Conciseness** Writing code in Ruby should involve the minimum amount of commands necessary. Code should be terse but also understandable.
2. **Consistency** Ruby coding should follow common conventions that make coding intuitive and unambiguous.
3. **Flexibility** There is no one right way. You should be able to pick the best approach for your needs and be able to even modify the base classes if necessary.

By keeping to these principles, Ruby assimilates the best features from many different languages while maintaining a clean, simple, and harmonious style. Understanding these principles is the key to understanding how to code effectively in Ruby. I call examples of this style Rubyisms, and this digital shortcut takes a closer look at selected Rubyisms to illustrate how Ruby works.

Rubyisms can be subtle. For instance, you might be working on a sample program (perhaps a blogging engine) and want to highlight titles posted less than 30 minutes ago. In Rails, you could express this idea simply with the following code:

```
@highlight_title = true if (posting_time < 30.minutes.ago)
```

Let's look at that again, thinking about the three principles described earlier. For starters, it certainly is concise.

The principle of flexibility is directly inherited from Perl, where it has long been expressed with this phrase: There's more than one way to do it. The beauty of this approach is that you can program in your most comfortable style.

You can write Java-like Ruby, but you'll do much better in the long run if you don't. There are more ways than one to do it, but some ways are better than others, and those ways are Rubyisms.

> Ruby inherited the Perl philosophy of having more than one way to do the same thing. I inherited that philosophy from Larry Wall, who is my hero actually. I want to make Ruby users free. I want to give them the freedom to choose. People are different. People choose different criteria. But if there is a better way among many alternatives, I want to encourage that way by making it comfortable. So that's what I've tried to do.
>
> Yukihiro Matsumoto

One other concept often used to characterize Ruby is the **principle of least surprise**. This is commonly (but mistakenly) understood to mean that good Ruby code works exactly the way we expect it to, displaying both conciseness and consistency.

```
return if no_harm?

puts "No Comments" unless @comments.any?

def quality
  case score
    when (67..100) then :good
    when (34..66) then :okay
    when (0..33) then :bad
    else :unknown
  end
end

def category_links(article)
  "Posted in " + article.categories.collect { |c|
       link_to c.name,
       { :controller => "articles",
         :action => "category",
         :id => c.permalink
       },
       :rel => "tag"
  }.join(", ")
end
```

The fourth example creates an output string, such as "Posted in Ruby, Rails, Howtos," with hyperlinks in it, that is placed at the foot of every blog post.

Even if you had barely any understanding of Ruby, you should be able to

look at the first three examples and grasp immediately what they are doing. Like that other high-level dynamic interpreted language Python, Ruby is sometimes lovingly described as "pseudo-code that runs."

Not necessarily readable by the complete outsider, this code still makes complete sense to someone who understands Ruby. And that is the correct meaning of **least surprise.** Ideally, code should be intuitive to noncoders, but it most definitely should always be nonsurprising to people familiar with the Ruby language.

Objects, Classes, and Modules

Like many modern programming languages, Ruby is object oriented. However, Ruby could be considered to be a purer object-oriented language than many. To see what I mean, let's look closer at that expression **30.minutes.ago** from the previous example. You might have naturally assumed that it is a bit of syntactic sugar. Indeed, that is how it would have to work in languages such as C++, Java, and PHP, where integers, floats, and sometimes strings are treated as primitives below the OOP framework. In Ruby, however, everything is an object even integer.

```
irb> 30.class
=> Fixnum
irb> Fixnum.superclass
=> Integer
irb> Integer.superclass
=> Numeric
irb> Numeric.superclass
=> Object
irb> Object.superclass
=> nil
```

We can call methods on the integer directly and traverse up its class hierarchy to the **Object** class, which is the root class for all other classes in Ruby.

```
irb> (30.public_methods - Object.public_methods).sort
["%", "&", "*", "**", "+", "+@", "-", "-@", "/", "<<", ">>", "[]", "^", "abs",
"between?", "ceil", "chr", "coerce", "div", "divmod", "downto", "floor",
"id2name", "integer?", "modulo", "next", "nonzero?", "prec", "prec_f", "prec_i",
"quo", "remainder", "round", "singleton_method_added", "size", "step", "succ",
"times", "to_f", "to_i", "to_int", "to_sym", "truncate", "upto", "zero?", "|",
"~"]
```

This ability to query objects and see their classes and methods is called **reflection**, and it is part of Ruby's powerful metaprogramming capabilities.

```
~rails/myproject> script/console
>> (30.public_methods - Object.public_methods).sort
=> ["%", "&", "*", "**", "+", "+@", "-", "-@", "/", "<<", ">>", "[]", "^",
"abs", "ago", "between?", "byte", "bytes", "ceil", "chr", "coerce", "day", "days",
"denominator", "div", "divmod", "downto", "even?", "exabyte", "exabytes",
"floor", "fortnight", "fortnights", "from_now", "gcd", "gcd2", "gcdlcm",
"gigabyte", "gigabytes", "hour", "hours", "id2name", "integer?", "kilobyte",
"kilobytes", "lcm", "megabyte", "megabytes", "minute", "minutes", "modulo",
"month", "months", "multiple_of?", "next", "nonzero?", "numerator", "odd?",
"ordinalize", "petabyte", "petabytes", "power!", "prec", "prec_f", "prec_i",
"quo", "rdiv", "remainder", "round", "rpower", "second", "seconds", "since",
```

```
  "singleton_method_added", "size", "step", "succ", "terabyte", "terabytes",
"times", "to_bn", "to_f", "to_i", "to_int", "to_r", "to_sym", "truncate",
"until", "upto", "week", "weeks", "year", "years", "zero?", "|", "~"]
```

To understand is to see how seriously Ruby takes the principle of **flexibility**; it allows users to override and extend the functionality of its base classes if needed. And that is what Rails does here. In languages such as Java, base classes are precompiled and extended only through inheritance.

```
class Numeric
  def seconds
    self
  end
  alias :second :seconds

  def minutes
    self * 60
  end
  alias :minute :minutes

  def hours
    self * 60.minutes
  end
  alias :hour :hours

  # Reads best without arguments: 10.minutes.ago
  def ago(time = ::Time.now)
    time - self
  end

  # Reads best with argument: 10.minutes.until(time)
  alias :until :ago

  # Reads best with argument: 10.minutes.since(time)
  def since(time = ::Time.now)
    time + self
  end

  # Reads best without arguments: 10.minutes.from_now
  alias :from_now :since
end
```

Although this might be acceptable if you are a Java programmer and have no other choice, writing Ruby is supposed to be creative and immediate; it is just not supposed to look like that. And this is where Ruby's principle of flexibility really shines. This is what Matsumoto meant by the "freedom to

choose": Any user is allowed to tweak and extend Ruby's core libraries for his or her own ends. And it makes the other two principles attainable; by allowing flexibility, Ruby enables code to be concise and consistent.

Before moving on, we should cover one other mechanism for extending functionality in Ruby: the mixin. Like Java, Ruby supports only single inheritance for classes. However, Ruby also allows users to define modules that contain methods, constants, classes, and even other modules. This is normally used for namespacing (it is the technique that eliminates confusion between the classes **ActiveRecord::Base** and **ActionController::Base**), but it is also the basis for mixing in code to classes. Mixins enable you to easily add functionality to multiple places without repeating yourself, which reduces cut-and-paste coding.

```
module Enumerable
  def inject
    ......
  end

  def partition
    ......
  end

  ......
end

class Array
  include Enumerable
end

class Hash
  include Enumerable
end
```

Duck Typing

In a strongly typed language such as Java, type is paramount. An object's type is its class, and only variables of the same class can use its methods. Variables whose type is a superclass or interface can also use the object, but they can only call methods defined at or above their level, an information-hiding principle generally known as abstraction. Thanks to C++ and Java, this approach has largely become canon.

In the early days of OOP, types were needed so that the compiler could determine exactly which methods would be invoked where; this in turn allows the compiler to optimize method calls because it has all the information it needs at compilation time. In most modern statically typed object-oriented languages, however, method resolution is performed at runtime anyway. Instead of being an optimization technique, static type checking is a security mechanism screening callers: A caller can use an object only if it matches the type he is expecting. Otherwise, the program does not compile. This assurance is known as type safety, and the goal of it has been to build a web of types and expectations to ensure that nothing unexpected occurs in the program.

Bugs happen in both statically typed and dynamically typed programs, and the best defense is unit testing or some other more-rigorous testing approach.

Because it is not compiled and has no static types, Ruby might seem to be more dangerous to develop in; but in some sense, it can actually be safer to program in: Because developers have no illusions that type checking is a serious defense against bugs, they will not confuse compilation with correctness.

Ruby takes a more enlightened approach to type: Ruby just does not care. Classes are still useful in Ruby as a way of defining methods for objects, but they are not important as a typing framework. Ruby does not check that a variable has the right class type for a call; instead, the only important thing Ruby cares about is this: Does this object have this method I'm calling at this instant? Methods are all that matters; the class is not used at all. This approach is commonly referred to as **duck typing**, after the aphorism "**If it talks like a duck and walks like a duck, it is a duck.**" The class of the object has no effect, and you can call a method without needing to cast the object appropriately. The result is cleaner, simpler, and more-understandable code.

Don't feel bad if you still have that preoccupation with classes and types; old habits die hard, and the importance of types has been drummed into your head for years by every computer science textbook. If you really want to program in the Ruby way, however, you need to get over that. Forget about figuring out object types. And when you get used to duck typing, you will find it hard to go back.

Symbols

```
Category.find(:all, :order => :position, :limit => 1000)
```

A symbol in Ruby is basically an immutable string. This is a useful abstraction for representing enumerated values in terms of performance and memory consumption. Because symbols are immutable, multiple references can refer to a single spot of storage in memory. It does not matter whether there are one or one million variables with the value :**action** in your code; they all use a single shared location in memory. In addition, comparing whether two symbol references are the same is trivial, because you need merely check that they point to the same location in memory. Symbols are certainly efficient.

Rails's extensive use of symbols reflects a rather common but underacknowledged Rubyism: Symbols denote specialness. More explicitly, the use of a symbol indicates that the name or value is not an arbitrary string but one of a limited set of expected values. This is also known as an enumeration in other languages, but Ruby does not require you to define the list of allowed values beforehand.

The whole point of constants is that they represent something else (for example, **Math::PI** represents a numeric value), whereas symbols represent nothing but themselves.

Simple to use and well suited for their purpose, symbols are one of the unique features of Ruby.

Blocks

Instead of drivers having to pick a complex route over local roads, the highway limits their route. To enter a highway is to largely surrender navigational choices: There are no turns or forks in the road to choose; backtracking and U turns are forbidden; your only option is to drive forward until you reach your exit. You can drive faster because you have less to decide, and you can concentrate on driving because you have less to consider.

An iterator is like a highway for a block of code. The iterator acts by starting the code at the first element (the entrance ramp if you will) and then moving the code forward through each element until it finishes with the last (the exit ramp). The result is much cleaner code. Callers who need to iterate over the element no longer need to worry about keeping track of where they are in the array and how to get to the next element; the iterator handles that. All the calling code needs to do is apply whatever logic it needs against each array element.

In Java, if you want to iterate over a collection, you call a method to get an object that implements the **Iterator** interface, which abstracts away the internals of the target object. However, you still have to write logic in your loop to drive through the elements of the iterator.

```
Iterator it = comments.iterator();
While (it.hasNext()) {
    Comment comment = (Comment) item.next();
    System.out.println("%s\n", c.text());
}
```

Ruby's iterators are more comprehensive. Not only do they abstract away the internals of the object, they do the driving through the array for you.

```
for c in @comments do
  puts c.text
end
```

All you need to provide is a block of code to be applied at each position, and Ruby does the lower-level work of iterating through the object for you. In Ruby, the preceding command is an example of syntactic sugar provided as a bridge for PHP and Perl programmers. Ruby actually executes that iteration as follows:

```
@comments.each { |c| puts c.text }
```

This syntax might seem a little more obscure to express, but it is a lot more powerful to use. To understand how this works, you need to understand **blocks**. And understanding **blocks** is the single most important step to understanding Ruby.

Block is a generic computer science term for a chunk of logic, but in Ruby it has a specific meaning. In Ruby, a block is essentially a **nameless method**: a section

of executable code with zero or more input arguments. In Ruby's syntax, blocks are demarcated either by **{}** or the keywords **do** and **end**. (The usual convention is to use **{}** for single-line blocks and the keywords for multiline blocks.) The input parameters into a block are surrounded by vertical pipes.

If the block is the yin of Ruby iteration, **yield** is the yang. This keyword in Ruby indicates where within the method you want the block to be applied. After the **yield** statement, you can specify zero or more variables, which are passed in as parameters to the block. This is a bit of an unusual influence. Ruby actually borrowed this feature from CLU.

The **Enumerable** module is a selection of 28 methods that can be mixed into any object that defines an iterator named **each**. Each method in **Enumerable** could be described as a higher-level iterator, in that they abstract away using the **each** method within a block-driven call.

```
call_scientists if birdlist.any? {|bird| bird.endangered? }
```

Internally, the any? function works by running over the object's **each** method with the block and breaking and returning true if the block's logic evaluates to true for an element, and false otherwise. That is the genius of Ruby's block-driven iteration approach.

By allowing you to layer iterators above iterators, Ruby enables you to write your logic at the highest level where it makes sense without worrying about wrangling iteration at the lowest level.

```
Selection
common = bird.select {|bird, count| count > 5 }
rare = bird.reject { |bird, count| count > 5 }

Accumulation (summing up values)
sightings = birds.inject {|total, bird| total += b.count}

Detection
indie = jukebox.any? {|x| x.artist.hip? }

Searching
Winner = contestants.find {|p| p.has_answer? }

Reordering
titles = jukebox.sort_by {|x| x.title }

Partitioning
quick, dead = people.partition {|p| p.can_outrun? :bear }

Recombination
blind_dates = restaurants.zip(men, women)
```

Ruby's block/**yield** approach is much richer than mere iteration, however. The block approach allows the caller to perform arbitrary calculations against an object, but the target object still maintains control over what the caller can do. This approach also proves useful where the target object has to allow access while maintaining data integrity and properly managing resources.

```
File.open(path, 'rb') do |file|
  until file.eof
    puts file.gets
  end
end
```

There is no need to call **File.close**; when this block exits, the **File** object automatically closes the file. The **File** object can thus ensure that files are never left in an inconsistent state; even in cases where exceptions are thrown, the file is safely closed and handles released.

Similarly, you can also use blocks for other limited resources that need to be used safely: locks, database connections, and server sockets, among others.

Blocks also prove useful when it is easier to tackle a problem as nested parts rather than as a complicated whole. The Builder library used in Rails's **.rxml** files to create XML is a prime example of this approach. XML has proven to be a popular format for data exchange with its human-readable, hierarchical structure; but XML contains many subtle gotchas that can stump developers who just need to output a file. The **Builder** class for Rails solves this issue by abstracting the process of building XML files as a series of nested blocks, with elements posing as methods and attributes as parameters.

Builder mirrors the hierarchical structure of the result XML with a hierarchical nesting of blocks.

```
xml.instruct! :xml, :version => "1.0", :encoding => "UTF-8"
xml.feed "xml:lang" => "en-US", "xmlns" => 'http://www.w3.org/2005/Atom ' do
  xml.title @feed_title
  @items.each do |item|
    render :partial => "atom10_item ", :locals => {:item => item, :xml => xml }
  end
end
=======
<?xml version="1.0" encoding="UTF-8"?>
<feed xml:lang="en-US" xmlns=http://www.w3.org/2005/Atom>
  <title>Odds &amp; Ends</title>
  <item>...</item>
  <item>...</item>
</feed>
```

Metaprogramming

Metaprogramming is an umbrella concept that describes a variety of techniques for manipulating the very building blocks of the language. What we did with blocks and strings and objects was **programming**. **Metaprogramming** manipulates classes and methods instead and comprises a few related concepts.

1. Reflection
2. Message sending
3. Method missing

```
irb> Object.public_methods.sort
=> ["<", "<=", "<=>", "==", "===", "=~", ">", ">=", "__id__", "__send__",
"allocate", "ancestors", "autoload", "autoload?", "class", "class_eval",
"class_variables", "clone", "const_defined?", "const_get", "const_missing",
"const_set", "constants", "display", "dup", "eql?", "equal?", "extend",
"freeze", "frozen?", "hash", "id", "include?", "included_modules", "inspect",
"instance_eval", "instance_method", "instance_methods", "instance_of?",
"instance_variable_get", "instance_variable_set", "instance_variables",
"is_a?", "kind_of?", "method", "method_defined?", "methods", "module_eval",
"name", "new", "nil?", "object_id", "private_class_method",
"private_instance_methods", "private_method_defined?", "private_methods",
"protected_instance_methods", "protected_method_defined?",
"protected_methods", "public_class_method", "public_instance_methods",
"public_method_defined?", "public_methods", "respond_to?", "send",
"singleton_methods", "superclass", "taint", "tainted?", "to_a", "to_s", "type",
"untaint"]
```

We also used Ruby's reflection techniques when we extended classes dynamically with new methods, and these capabilities are popular anywhere you might need to dynamically examine or alter the structure of a class. For instance, the Unit Testing library in Rails uses reflection to automatically discover and execute all the tests defined within a suite.

One particularly useful method among these is the **respond_to?** method. This method allows you to conditionally apply duck typing against any Ruby object. This call is a useful way to determine whether a method or attribute is defined, and can be used to avoid a dreaded **NoMethodError** when calling a method against a completely unknown object:

```
irb> i = 27
irb> str = "foo, bar, baz"
irb> str.respond_to? :split     => true
irb> i.respond_to? :split       => false
```

In Ruby an object's methods are not fixed at any compilation time but can be dynamically extended or modified at any point. For now, we have called the methods directly by name, but it is possible instead to invoke generically any object method by using a string or symbol variable to specify the target method.

To illustrate a use of **send**, consider another common programming technique: the **dispatch table**. Sometimes you might find yourself needing to create a method that dynamically invokes other methods based on the types of input.

```
def search(type, terms)
  case type
    when "title" then title_search(terms)
    when "author" then author_search(terms)
    when "subject" then subject_search(terms)
    else raise NoMethodError
  end
end
```

Under Ruby's dynamic duck typing model, however, we can radically simplify this code:

```
def search(type, terms)
  send("#{type}_search", terms)
end
```

Philosophically, Ruby's object-oriented model is inspired by Smalltalk; in that language, programs do not call methods but instead send messages to an object. This might seem like splitting hairs, but a significant semantic difference exists. In languages such as Java or C++, programs can only call methods, because it is impossible for objects to dynamically modify their responses to calls while the program is running. (They can only execute the methods linked during compilation.) In a language such as Ruby, however, it is possible for objects to do quite a few things with that message. Normally, most objects just invoke a method, but an object could also raise an error, delegate the message to another object, or even allow a message for which it has no methods defined.

For completeness, Ruby also includes **Method** objects for those who prefer an object-based mechanism for dynamic execution. The **Object#method** call returns a reference to a specific method on an object:

```
split = "foo, bar, baz".method(:split)
split.call
--> ["foo,", "bar,", "baz"]
```

This might prove useful if you want to defer execution of a method or store a workflow as a sequence of actions on multiple objects. You might also think this would be the mechanism for parameterized operations such as quick sorts, but that case is covered quite adeptly by blocks. In addition, the **Method** object

is not just a reference to a generic class method, but a method within a particular object.

In Ruby's dynamic dispatching, it is even possible to execute a method that does not exist. How is it possible to invoke a method if it is not defined? Meet **method_missing**, another method defined in **Object**. In Ruby, **method_missing** is the catchall method invoked if you attempt to call a method that is not there. By default, it is simply defined as follows:

```
class Object
  def method_missing(method_id, *arguments)
    # code that constructs the error string elided
    raise NoMethodError, "undefined method..."
  end
end
```

Notice, however, that this error is not being thrown by the Ruby interpreter but by a method within the **Object** classa method that, like any other method, can be overridden in a subclass.What if you define **method_missing** in a subclass to not raise an error but do something useful instead? You could then have an object that could run any arbitrary request thrown at it!

When you first began learning Rails, you might have noticed that it provided a comprehensive selection of methods for searching your back-end model.

```
User.find(4)
User.find_all
User.find_by_name("Jake")
User.find_all_by_country("Switzerland")
User.find_by_first_name_and_last_name("Jacob" , "Harris")
```

You now know magic is really just another name for **method_missing.**

```
def method_missing(method_id, *arguments)
  if match = /find_(all_by|by)_([_a-zA-Z]\w*)/.match(method_id.to_s)
    finder = determine_finder(match)

    attribute_names = extract_attribute_names_from_match(match)
    super unless all_attributes_exists?(attribute_names)

    conditions = construct_conditions_from_arguments(attribute_names,
arguments)
    options = { :conditions => conditions }
    set_readonly_option!(options)
    send(finder, options)
  else
```

```
      super
    end
  end

  def determine_finder(match)
    match.captures.first == 'all_by' ? :find_every : :find_initial
  end

  def find_initial(options)
    # Code to execute a query against the DB with LIMIT 1
  end
  def find_every(options)
    # Code to execute a query against the DB
  end
```

Notice how several Rubyisms we have covered already work in harmony within this method. Symbols are used to indicate methods. **send** is used to dynamically invoke a choice of finder methods. Reflection against fields in the databases determines what conditions are valid. And finally, Ruby's concise and consistent code makes the implementation of **method_missing** completely understandable and maintainable. In turn, this technique adds flexibility and power to Rails, enabling users to more concisely and clearly define their controllers in their Rails applications. Therefore, you might see why I think it will be impossible to create true Rails clones on top of other languages; Rails and Ruby are perfectly intertwined.

Domain-Specific Languages

It is time to look at the one other aspect of Rails's success: how Rails writes complicated code using only simple declarations. Let's examine our final Rubyism of this digital shortcut: **domain-specific languages**.

I have thus far neglected to touch upon one other important aspect of highways: signage. That would be a serious oversight, because highway signs are one of the most effectively designed examples of communication on the planet. They provide necessary and unambiguous information to motorists who have only seconds to read them. Consequently, they succeed only through simplicity. All highway signs are tediously similar in style. They are made from the same types of reflective material, use similar phrasing (Exit 12 1 mile), and have only a rigorously regulated selection of decorative navigational arrows and facility symbols allowed. Essentially, highway signs are a language constructed from a limited vocabulary of symbols and words. This is easy for motorists to understand because the same language is used everywhere. (An exit is an exit is an exit.) It is also easy for sign creators, who have a consistent way of laying out signs. And it is good for navigation, because a leg of any route can be expressed in the same language. ("Take Route 1 21 miles to Exit 7" is clearer to follow than "Go down the Old Boston Post road until you see intersection with a gas station; take a right to Pawcatuck.") In computer programming terms, highway signs are an example of a **domain-specific language** (DSL).

A DSL is a limited computer language subset targeted to a particular use, as opposed to a general-purpose programming language. DSLs provide several important advantages as a programming technique. For starters, because a DSL is targeted toward a specific problem domain, it can be understood even by a user who does not know the broader language. In addition, this specific focus necessitates simpler abstractions and models, which makes it easier to achieve conciseness and consistency. Even for an expert coder, a DSL makes sense because it allows you to code solutions as higher-level concepts rather than more-verbose lower-level implementations. This results in code that is easier to design and maintain.

```
validates_presence_of :name, :email
```

And what if we want to change our validation for e-mail later? It is easier to modify that one line than look for an explicit method declaration. And that is the point of DSLs in a nutshell: They enable you to express complicated operations as much simpler expressions. In essence, they trick you into coding without even realizing it!

Martin Fowler, who coined the term for DSLs, distinguishes between two types of DSLs. **External DSLs** are those cases where the DSL is a file format that can be parsed into a configuration or operations in the general language.

```
development:
 adapter: mysql
 database: myapp_dev
 host: localhost
 username: root
 password: changeme
```

External DSLs require additional parsing logic to work, which make them harder to create, but that is the limit of what you can do in statically typed languages. However, Ruby's flexibility also supports a second type of DSL. An **internal DSL** is a sublanguage specified within the higher-level language itself. Internal DSLs are general-purpose code disguised as a simplified language.

In fact, Rails itself could essentially be described as a DSL for web applications. Or rather, it is a collection of various separate DSLs focused on specific tasks, including the following:

- Associations
- Validations
- Routing
- Rendering
- Builder for XML
- acts_as_* plug-ins

Associations are an excellent example of how Rails defines DSLs internally. In Rails, you can specify the relationship between two tables in your database using the has_one, has_many, and belongs_to declarations.

```
class Review < ActiveRecord::Base
  belongs_to :user
end
```

By specifying the relationship of your model to another table, Rails automatically adds all these additional methods to your Review class:

- user, which returns a reference to the user a review is associated with
- user=, which can be used to reassign the association to user
- build_user, which creates a user and links this Review to it
- create_user, which works like build and automatically saves the association, too
- Automatic code to update foreign keys in the table when the object is saved

You just need to make the declaration, and it adds the methods and builds

the SQL you need without you needing to think at that levelwhich is precisely the point of DSLs. By limiting you to a specific sublanguage, they give you the precise high-level way of specifying what you want to do.

It might still surprise you at this point, but belongs_to is not a keyword or special construct, but just a normal method within ActiveRecord::Base. DSLs are prime examples of how useful Ruby's metaprogramming can be. In this case, this simple declaration of one line automatically creates methods and adds them to the class.

```
   def belongs_to(association_id, options = {})
     reflection = create_belongs_to_reflection(association_id, options)

     if reflection.options[:polymorphic]
       # Code elided for brevity
     else
       association_accessor_methods(reflection, BelongsToAssociation)
       association_constructor_method(:build,                    reflection,
BelongsToAssociation)
       association_constructor_method(:create,                   reflection,
BelongsToAssociation)

       module_eval do
          before_save <<-EOF
            association = instance_variable_get("@#{reflection.name}")
            if !association.nil?
               if association.new_record?
                  association.save(true)
               end
               if association.updated?
                  self["#{reflection.primary_key_name}"] = association.id
                end
            end
           EOF
        end
      end
   end
```

This example also illustrates another metaprogramming technique when the method calls module_eval, which evaluates a string in the current context. (In this case, the context is Review.) To make this more comprehensible, the actual code being evaluated at this point with our example is this:

```
   association = instance_variable_get("@user")
   if !association.nil?
     if association.new record?
       association.save(true)
```

```
    end
    if association.updated?
      self["user_id"] = association.id
    end
  end
```

The addition of this method makes sure the foreign key association between this object and its owner is correctly stored in the database whenever the object is saved if we change the value of that association.

But how are the methods such as user and create_user added to the model? Let's look at how belongs_to adds methods to your model.

```
  def association_accessor_methods(reflection, association_proxy_class)
    define_method(reflection.name) do |*params|
      force_reload = params.first unless params.empty?
      association = instance_variable_get("@#{reflection.name}")

      if association.nil? || force_reload
        association = association_proxy_class.new(self, reflection)
        retval = association.reload
        unless retval.nil?
          instance_variable_set("@#{reflection.name}", association)
        else
          instance_variable_set("@#{reflection.name}", nil)
          return nil
        end
      end
      association
  end

  define_method("#{reflection.name}=") do |new_value|
    association = instance_variable_get("@#{reflection.name}")
    if association.nil?
      association = association_proxy_class.new(self, reflection)
    end
    association.replace(new_value)
      unless new_value.nil?
        instance_variable_set("@#{reflection.name}", association)
      else
         instance_variable_set("@#{reflection.name}", nil)
         return nil
      end

      association
    end
```

```
end
```

The answer for the first case is as follows:

```
def user(*params)
  force_reload = params.first unless params.empty?
  association = instance_variable_get("@user")
  if association.nil? || force_reload
    association = BelongsToAssociation.new(self, reflection)
    retval = association.reload
    unless retval.nil?
      instance_variable_set("@user", association);
    else
      instance_variable_set("@user", nil);
    end
  end
  association
end
```

Notice that the reflection object passed to define_method is used within the method and becomes part of the method's scope at creation. This is a technique known as a **closure**.

There are many other interesting examples of DSLs to be found, but the important thing to grasp here is how Rails uses metaprogramming to construct its DSLs (and how the flexibility of this metaprogramming allows Rails to provide a DSL for specifying database associations in a way that is concise and consistent [the three principles yet again]). DSL is a powerful technique for containing complexity when building powerful systems.

Conclusion

I hope this has been educational and inspirational in your understanding of Ruby, but all I can really wish is that you now have a sense of Ruby's beauty. You can learn Ruby's syntax and structure, you can learn its classes and modules, you can even learn its metaprogramming; but without understanding its beauty and philosophy, you might as well learn nothing at all.

It became clear to me that learning Ruby without understanding its philosophy was like studying French without hearing it being spoken by fluent speakers an interesting exercise in grammar but ultimately as dry and lifeless as Latin or ancient Greek.

To my chagrin, I realized I did not really know Ruby myself, because I did not know its principles and how those guided its design. Although I knew how to write small Ruby programs, I had no knowledge of the way fluent speakers use it. I needed a good example to learn from; luckily for me, I did not really know Rails either. So I began reading through Rails source and finding there was no better teacher about how Ruby really can be fluently spoken. And so, my originally dry presentation began life again as a rich exploration into Ruby's style and how Rails relies on it.

Now I hope it continues for you. A trip like this is easy enough to start (just run gem unpack rails to save Rails's gem source to a directory or retrieve the latest version of Rails from the subversion repository); all it takes is a question. So, start asking yourself some questions: How do validations work? Where did they add polymorphism to Rails's associations? How does the controller really know where to find the view? See where these questions take you. I think you will love the journey.