Programming in Ruby

Dave Thomas Andy Hunt[1]

September 13, 2000

[1]Dave and Andy are authors of *Programming Ruby* and *The Pragmatic Programmer*, both from Addison-Wesley. They run an independent consultancy from offices in Dallas, TX and Raleigh, NC. Contact them via `www.pragmaticprogrammer.com`

Take the pure object-orientation of Smalltalk, but remove the quirky syntax and the reliance on a workspace. Add in the convenience and power of Perl, but without all the special cases and magic conversions. Wrap it up in a clean syntax based in part of Eiffel, and add a few concepts from Scheme, CLU, Sather, and Common Lisp. You end up with Ruby, a language that is already more popular than Python in its native Japan.

Ruby is a pure, untyped, object-oriented language—just about everything in Ruby is an object, and object references are not typed. People who enjoy exploring different OO programming paradigms will enjoy experimenting with Ruby: it has a full metaclass model, iterators, closures, reflection, and supports the runtime extension of both classes and individual objects.

Ruby is being used world-wide for text processing, XML and web applications, GUI building, in middle-tier servers, and general system administration. Ruby is used in artificial intelligence and machine-learning research, and as an engine for exploratory mathematics.

Ruby's simple syntax and transparent semantics make it easy to learn. Its direct execution model and dynamic typing let you develop code incrementally: you can typically add a feature and then try it immediately, with no need for scaffolding code. Ruby programs are typically more concise than their Perl or Python counterparts, and their simplicity makes them easier to understand and maintain. When you bump up against some facility that Ruby is lacking, you'll find it easy to write Ruby extensions, both using Ruby and by using low level C code that adds new features to the language.

We came across Ruby when we were looking for a language to use as a prototyping and specification tool. We've used it on all of our projects since. We have Ruby code performing distributed logging, executing within an X/Windows window manager, precompiling the text of a book, and generating indexes. Ruby has become our language of choice.

This article looks at just a few of the interesting and innovative features of the Ruby language.

Everything's An Object

Everything you manipulate in Ruby is an object, and all methods are invoked in the context of an object. Let's look at several examples.

```
"gin joint".length  →  9
"Rick".index("c")   →  2
-1942.abs           →  1942
sam.play(aSong)     →  "duh dum, da dum de dum ..."
```

In Ruby and Smalltalk jargon, all method calls are actually messages sent to an object. Here, the thing before the period is called the *receiver*, and the name after the period is the method to be invoked.

The first example asks a string for its length, and the second asks a different string to find the index of the letter "c." The third line has a number calculate its absolute value. Finally, we ask the object "sam" to play us a song.

It's worth noting a major difference between Ruby and most other languages. In (say) Java, you'd find the absolute value of some number by calling a separate function and passing in that number. In Ruby, the ability to determine absolute values is built into numbers—they take care of the details internally. You simply send the message `abs` to a number object and let it do the work.

```
number = Math.abs(number) // Java
number = number.abs       // Ruby
```

The same applies to all Ruby objects: in C you'd write `strlen(name)`, while in Ruby it's `name.length`, and so on. This is part of what we mean when we say that Ruby is a genuine OO language.

And one last thing (for now) on calling methods: the parentheses are optional unless the result would be ambiguous. This is a big win for parameterless methods, as it cuts down on the clutter generated by all those `()` pairs.

Classes and Methods

As example 1 shows, Ruby class definitions are remarkably simple: the keyword `class` followed by a class name, the class body, and the keyword `end` to finish it all off. Ruby features single inheritance: every class has exactly one superclass, which can be specified as shown in example 2. A class with no explicit parent is made a child of class `Object`–the root of the class hierarchy, and is the only class with no superclass. If you're worried that a single inheritance model just isn't enough, never fear. We'll be talking about Ruby's mixin capabilities shortly.

Let's get back to the definition of class `Song` in example 1. The class contains two method definitions, `initialize` and `to_s`. The `initialize` method participates in object construction. To create a Ruby object, you send the message `new` to the object's class, as shown in the last line of the example. This `new` message allocates an empty, uninitialized object, and then sends the message `initialize` to that object, passing along any parameters that were originally given to `new`. This makes `initialize` roughly equivalent to constructors in C++ and Java.

Our class `Song` also contains the definition of the method `to_s`. This is a convenience method. Ruby sends `to_s` to an object whenever it needs to represent that object as a string. By overriding the default implementation of `to_s` (which is in class `Object`) we get to control how our objects are printed, for example by tracing statements and the debugger, and when interpolated in strings.

In example 2, we create a subclass of class `Song`, overriding both the `initialize` and `to_s` methods. In both of the new methods we use the `super` keyword to invoke

the equivalent method in our parent class. In Ruby, `super` is not a reference to a parent class; instead it is an executable statement that reinvokes the current method, skipping any definition in the class of the current object.

Attributes, Instance Variables, and Bertrand Meyer

The `initialize` method in class `Song` contains the line

```
@title = title
```

Names that start with single "at" signs (@) are *instance variables*—variables that are specific to a particular instance or object of a class. In our case, each `Song` object has its own title, so it makes sense to have that title be an instance variable. Unlike languages such as Java and C++, you don't have to declare your instance variables in Ruby; they spring into existence the first time to reference them.

Another difference between Ruby and Java/C++ is that you may not export an object's instance variables; they are available to subclasses, but are otherwise inaccessible. (This is roughly equivalent to Java's `protected` concept). Instead, Ruby has attributes: methods that get and set the state of an object. You can either write these attribute methods yourself, as shown in example 3, or use the handy-dandy Ruby shortcuts in example 4.

It's interesting to note the method called `title=` in example 3. The equals sign tells Ruby that this method can be assigned to—it can appear on the left-hand side of an assignment statement. If you were to write

```
aSong.title = "Chicago"
```

Ruby translates it into a call to the `title=` method, passing "Chicago" as a parameter.

This may seem like some trivial syntactic sugar, but it's actually a fairly profound feature. You can now write classes whose attributes act as if they were variables, but are actually method calls. This decouples the users of your class from its implementation: you're free to change an attribute back and forth between some algorithmic implementation and a simple instance variable. In his landmark *Object-Oriented Software Construction*, Bertrand Meyer calls this the "Uniform Access Principle."

Blocks and Iterators

Have you ever wanted to write your own control structures, or package up lumps of code within objects? Ruby's block construct lets you do just that.

A block is simply a chunk of code between braces, or between `do` and `end` keywords. When Ruby comes across a block, it stores the block's code away for later; the block

is not executed. In this way, a block is similar to an anonymous method. Blocks can only appear in Ruby source alongside method calls.

A block associated with a method call can be invoked from within that method. This sounds innocuous, but this single facility lets you write callbacks and adaptors, handle transactions, and implement your own iterators. Blocks are also true closures, remembering the context in which they were defined, even if that context has gone out of scope. Let's just look at blocks as iterators for now.

The method in example 5 implements an iterator that returns successive Fibonacci numbers (the series that starts with two 1's, and where each term is the sum of the two preceding terms). The main body of the method is a loop that calculates the terms of the series. The first line in the loop contains the keyword `yield`, which invokes the block associated with the method, in this case passing as a parameter the next Fibonacci number. When the block returns, the method containing the `yield` resumes. Thus in our Fibonacci example, the block will be invoked once for each number in the series until some maximum is reached.

Example 6 shows this in action. The call to `fibUpTo` has a block associated with it (the code between the braces). This block takes a single parameter—the name between the vertical bars at the start of the block is like a method's parameter list. The body of the block simply prints this value.

If you write your own collection classes (or any classes that contain a stream of values) you can benefit from the real beauty of Ruby's iterators. Say you've produced a class that stores objects in a singly-linked list. The method `each` shown in example 7 traverses this list, invoking a block for each node. This is a Visitor Pattern in a three lines of code.

The choice of the name, `each`, was not arbitrary. If your class implements an `each` method, then you can get a whole set of other collection-oriented methods for free, thanks to the `Enumerable` mixin.

Modules, Mixins, and Multiple Inheritance

Modules are classes that you can't instantiate: you can't use `new` to create objects from them, and they can't have superclasses. At first, they might seem pointless, but in reality modules have two major uses.

- Modules provide namespaces. Constants and class methods may be placed in a module without worrying about their names conflicting with constants and methods in other modules. This is similar to the idea of putting static methods and variables in a Java class. In both Java and Ruby you can write `Math.PI` to access the value of π (although in Ruby, `PI` is a constant, rather than a final variable, and you're more likely to see the notation `Math::PI`).

Blocks and Closures

A Ruby block can be converted into an object of class `Proc`. These `Proc` objects can be stored in variables and passed between methods just like any other object. The code in the corresponding block can be executed at any time by sending the `Proc` object the message `call`.

Ruby `Proc` objects remember the context in which they were created: the local variables, the current object and so on. When called, they recreate this context for the duration of their execution, even if that context has gone out of scope. Other languages call `proc` objects *closures*.

The following method returns a `Proc` object.

```
def times(n)
  return Proc.new { |val| n * val }
end
```

The block multiplies the method's parameter, "n", by another value, which is passed to the block as a parameter. The following code shows this in action.

```
double = times(2)
double.call(4)  →  8
santa = times("Ho! ")
santa.call(3)   →  "Ho! Ho! Ho! "
```

Even though the parameter "n" is out of scope when the `double` and `santa` objects are called, its value is still available to the closures.

- Modules also are the basis for *mixins*, a mechanism by which you add canned behavior to your classes.

Perhaps the easiest way to think about mixins is to imagine that you could write code in a Java interface. Any class that implemented such an interface would receive not just a type signature; it would receive the code that implemented that signature as well. We can investigate this by looking at the `Enumerable` module, which adds collection-based methods to classes that implement the method `each`.

`Enumerable` implements the method `find` (among others). `find` returns the first member of a collection that meets some criteria. This example shows `find` in action, looking for the first element in an array that is greater that four.

```
[1,3,5,7,9].find {|i| i > 4}  →  5
```

Class `Array` does not implement the `find` method. Instead, it mixes in `Enumerable`, which implements `find` in terms of `Array`'s `each` method. Example 8 shows

how this might be done. Contrast this approach with both Java and C#, where it is up to the class implementing the collection to also provide a considerable amount of supporting scaffolding.

Although a class may have only one parent class (the single inheritance model), it may mix in any number of modules. This effectively gives Ruby the power of multiple inheritance without some of the ambiguities that can arise. (And in cases where mixing in modules would cause a name clash, Ruby supports Eiffel-style method renaming.)

Other good stuff

This article is too short to do justice to all of Ruby. However, let's briefly touch on some additional highlights.

- *Classes and modules are never closed*. You can add to and alter all classes and modules (including those built in to Ruby itself).
- *Reflection*. As well as supporting reflection into both classes and individual objects, Ruby lets you traverse the list of currently active objects.
- *Marshalling*. Ruby objects can be serialized and deserialized, allowing them to be saved externally and transmitted across networks. A full distributed object system, DRb, is written in about 200 lines of Ruby code.
- *Libraries*. Ruby has a large (and growing) collection of libraries. All major Internet protocols are supported, as are most major databases. Extending Ruby is simple compared with (say) Perl.
- *Threads*. Ruby has built-in support for threads, and doesn't rely on the underlying operating system for thread support.
- *Object specialization*. You can add methods to individual objects, not just classes. This can be useful when defining specialized behavior for objects (for example, determining how they respond to GUI events).
- *Exceptions*. Ruby has a fully object-oriented, extensible exception model.
- *Garbage collection*. Ruby objects are automatically garbage collected using a mark-and-sweep algorithm. As well as simplifying programming, the choice or mark and sweep makes writing extensions easier (no reference counting problems).
- *Active developer community*. The Ruby development community is still a bazaar, small, intimate, and bustling. Changes are discussed openly and are made efficiently. You can have an impact on Ruby.

Some Real Examples

We finish off by looking at two larger Ruby programs. The first is a basic web server that echoes back the headers it receives. It's written as two classes, shown in listing 1. `WebSession` is a convenience class which provides two methods for writing to a TCP connection. The `standardPage` method is interesting. At a minimum it writes a standard page header and footer. If called with a block, however, it inserts the value returned by that block as the page body. This kind of wrapping functionality is a natural use for Ruby's blocks.

The `WebServer` class uses Ruby's TCP library to accept incoming connections on a given port. For each connection it spawns a Ruby thread, which reads the header and writes the contents back to the client. The code in the thread is wrapped in a `begin`/`end` block, used in Ruby to handle exceptions. In this case, we use an `ensure` clause to make sure that the connection to the client is closed even if we encounter errors while handling the request.

The second example packs a number of features into a small space. At its core, it represents the list of songs in an MP3 collection as an array, providing all the existing array functionality plus the ability to shuffle the entries randomly. If the array is sorted, then the entries will be ordered by song title.

Each entry in the array is an object of class `Song`. As well as providing a container for the song title, album, and artist, this class implements the general comparison operator, `<=>`. This operator is used when sorting containers containing songs: in this case we arrange to be sorted on the song title.

There are two common approaches to making our `MP3List` act as if it were an array: delegation or subclassing. Listing 2 shows the approach using delegation. The library module `delegate` provides a class `SimpleDelegator`, which handles all the details of forwarding method calls from class `MP3List` to the delegate. We create the array containing the songs, then invoke `SimpleDelegator`'s initialize method (using `super(songlist)`) to set up the delegation. From that point on, our `MP3List` will act as if it were an array. When the user shuffles a songlist, we create a new array containing the entries of the original in a random order, and use `SimpleDelegator`'s `__setobj__` method to delegate to that new array.

Listing 3 shows an alternative implementation of `MP3List` in which we subclass the builtin `Array` class and add our own `shuffle!` method. Why the exclamation mark? Ruby convention is to append a "!" to methods that change their object (or are otherwise dangerous), and to append a question mark to predicate method names.

Conclusion

Programming in Ruby is an immensely satisfying experience—the language seems to be able to represent high-level concepts concisely, efficiently, and readably. It's easy

Resources

Web sites. The official Ruby Home Page is `http://www.ruby-lang.org`. You can also find Ruby information at `http://www.rubycentral.com`. In particular, you'll find complete online references to Ruby's built-in classes and modules at `www.rubycentral.com/ref/`, and to the Ruby FAQ at `www.rubycentral.com/faq/`.

Download. The latest version of Ruby can be downloaded from: `http://www.ruby-lang.org/en/download.html`. Mirror sites are:

- `ftp://ftp.TokyoNet.AD.JP/pub/misc/ruby`
- `ftp://ftp.iij.ad.jp/pub/lang/ruby`
- `ftp://blade.nagaokaut.ac.jp/pub/lang/ruby`
- `ftp://ftp.krnet.ne.jp/pub/ruby`
- `ftp://mirror.nucba.ac.jp/mirror/ruby`
- `http://mirror.nucba.ac.jp/mirror/ruby`

Usenet. Ruby has its own newsgroup, `comp.lang.ruby`. Traffic on this group is archived and mirrored to the `ruby-talk` mailing list.

Mailing lists. For information on subscribing to `ruby-talk`, the English-language mailing list, see `http://www.ruby-lang.org/en/ml.html`.

to learn, and at the same time it is deep enough to excite even the most jaded language collector. Download a copy and try it for yourself. We think you'll like it.

```
class Song
  def initialize(title)
    @title = title
  end
  def to_s
    @title
  end
end
aSong = Song.new("My Way")
```

example 1: A simple class definition

```
class KaraokeSong < Song
  def initialize(title, lyric)
    super(title)
    @lyric = lyric
  end
  def to_s
    super + " [#@lyric]"
  end
end
```

example 2: A subclass of class Song

```
class Song
  # ...
  def title          # attribute reader
    @title           # returns instance variable
  end
  def title=(title)  # attribute setter
    @title = title
  end
end
```

example 3: Writing your own attribute methods

```
class Song
  # ...
  attr_accessor :title
end
```

example 4: Ruby shortcut for attribute methods

```
def fibUpTo(max)
  n1, n2 = 1, 1
  while n1 <= max
    yield n1            # invoke block value
    n1, n2 = n2, n1+n2  # and calculate next
  end
end
```

example 5: Iterator for Fibonacci numbers

```
fibUpTo(1000) { |term|  print term, " " }
```
produces:
```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

example 6: Using the `fibUpTo` iterator

```
class LinkedList
  # ...
  def each
    node = head
    while node
      yield node
      node = node.next
    end
  end
end
```

example 7: Iterator for a linked list

```
module Enumerable
  def find
    each {|val| return val if yield(val) }
  end
end

class Array
  include Enumerable
end
```

example 8: Adding functionality with a mixin

```
require "socket"
class WebSession
  def initialize(connection)
    @connection = connection
  end
  def write(string)
    @connection.write string
  end
  def standardPage(title)
    write "HTTP/1.1 200 OK\r\n"
    write "Content-Type: text/html\r\n\r\n"
    write "<html><head> <title>#{title}</title> </head>\n"
    write yield if block_given?
    write "</body></html>"
  end
end
class WebServer
  def initialize(port)
    @listen = TCPServer.new('localhost', port || 8080);
  end
  def run
    loop do
      Thread.start(@listen.accept) do |aConnection|
        begin
          session = WebSession.new(aConnection)
          request = []
          loop do
            line = aConnection.gets.chomp("\r\n")
            break if line.length == 0
            request << line
          end
          session.standardPage("Your Request") {
            "<h1>Your request was:</h1>\n" +
            request.join('<br>') +
            "<p>Thank you for testing our system."
          }
        ensure
          aConnection.close
        end  # begin
      end    # Thread
    end      # loop
  end
end
WebServer.new(ARGV[0]).run
```

listing 1: Simple Web Server

```
require 'delegate'
require 'find'
class Song
  attr_reader :title, :album, :artist
  def initialize(filename)
    @artist, @album, @title = filename.split("/")[-3..-1]
    @title.chomp!(".mp3")
  end
  def <=>(anOther)
    title <=> anOther.title
  end
  def to_s
    "'#{@title}'  #{@artist}--'#{@album}'\n"
  end
end
class MP3List < SimpleDelegator
  def initialize(base)
    songlist = Array.new
    Find.find(base) do |entry|
      if entry =~ /\.mp3$/
        if !block_given? or yield entry
            songlist << Song.new(entry)
        end
      end
    end
    super(songlist)
  end
  def shuffle!
    newlist = Array.new
    length.times do
      newlist << delete_at(rand(length))
    end
    __setobj__(newlist)
    self
  end
end
base = ARGV[0] || "/mp3"
list = MP3List.new(base + "/Hatfield And The North")
puts "Original:  ", list.sort
puts "Shuffled:  ", list.shuffle!
puts "5 entries: ", list[0..4]
puts "Filtered: "
list = MP3List.new(base) { |x| x =~ /Woke Up This Morning/ }
puts list
```

listing 2: MP3 File Lister With Delegation

```
class MP3List < Array
  def initialize(base)
    super()
    Find.find(base) do |entry|
      if entry =~ /\.mp3$/
        if !block_given? or yield entry
            self << Song.new(entry)
        end
      end
    end
  end
  def shuffle!
    newlist = Array.new
    length.times do
      newlist << delete_at(rand(length))
    end
    replace(newlist)
  end
end
```

listing 3: MP3 File Lister as Array Subclass