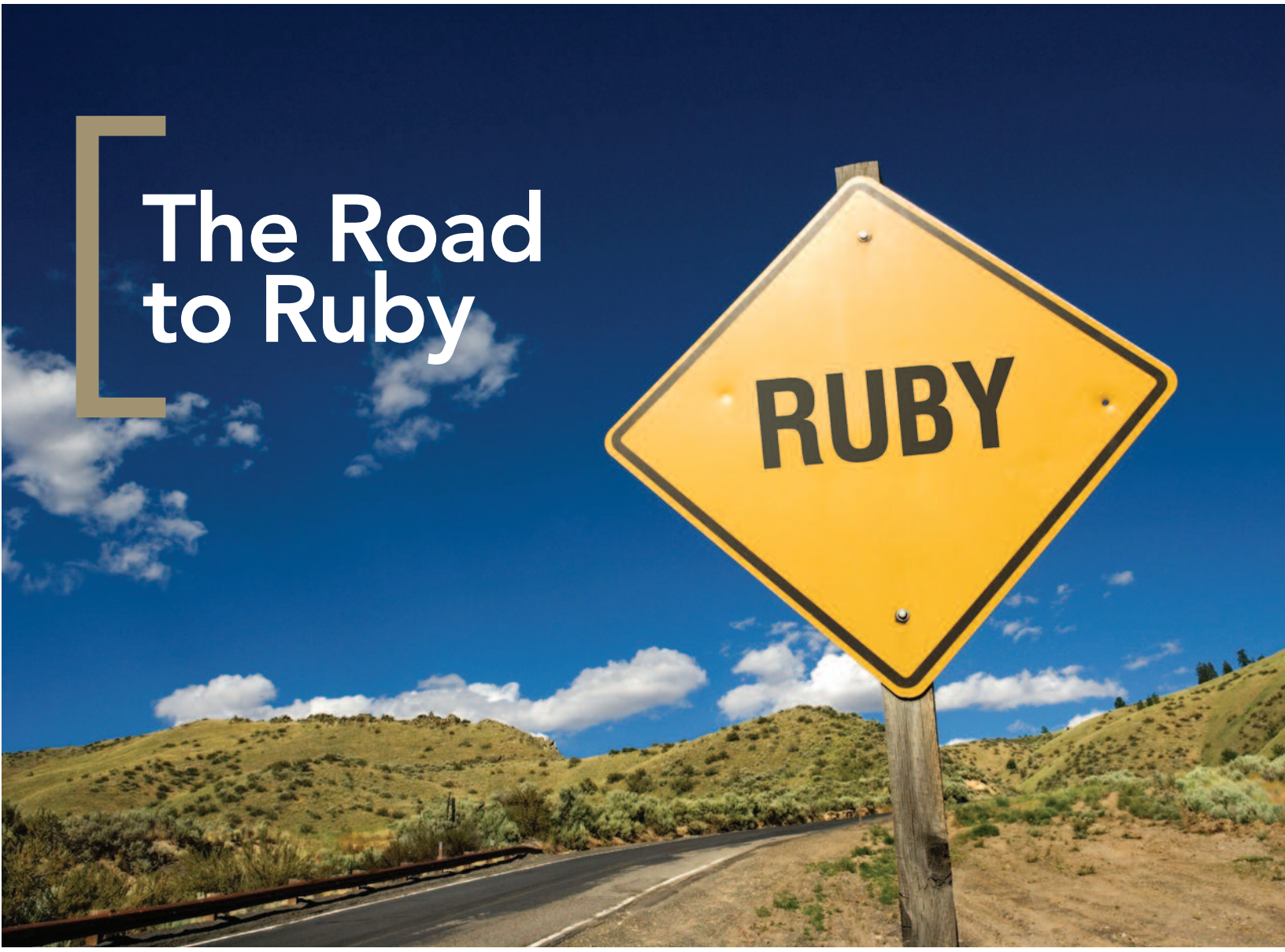
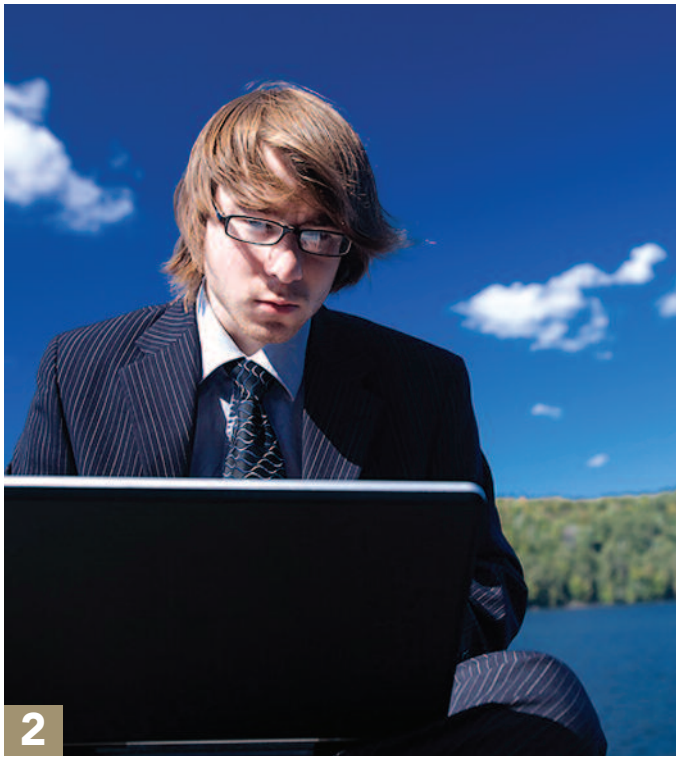
The Road to Ruby



an internet.com Developer eBook

contents

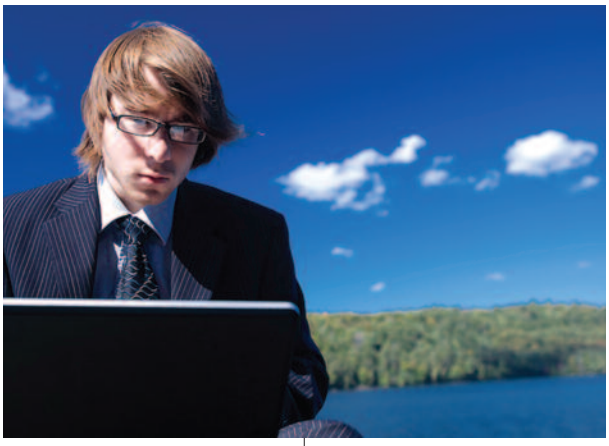
The Road to Ruby



A Java Developer's Guide to Ruby

By Mark Watson

As a Java developer, why should you learn Ruby? Because Ruby's versatility and flexibility complement Java well, and you will be a more effective and efficient developer if you use both languages. In fact, I use Java, Ruby, and Common Lisp for all my development and Ruby has become a core part of my work life. Specifically, the following reasons make Ruby compelling for Java developers:



Jupiterimages

- As a scripting language, Ruby is an effective tool for small projects. When I need to write utilities for data conversion and text processing quickly, I almost always use Ruby.
- Ruby is a dynamic and terse language.
- Using Ruby will often offer a different perspective on problem solving.
- JRuby is still a work-in-progress but I believe it eventually will provide an excellent Ruby deployment platform using the Java VM. Currently, IntelliJ, NetBeans, and Eclipse all provide excellent Ruby support.
- As the cost of software maintenance is roughly proportional to the number of lines of code, and Ruby programs are short and concise, they tend to be easier to read, understand, and maintain.
- The Ruby on Rails Web development framework is great for small and medium-sized database-backed web applications. You need to know Ruby if you want to use Ruby on Rails.

To demonstrate why Ruby is a good fit for Java developers, this article introduces the language features that will make you more efficient (see Table 1. Ruby and Java Feature Comparison) and then shows short program examples in both languages.

> Because Ruby's versatility and flexibility complement Java well, and you will be a more effective and efficient developer if you use both languages.

Table 1. Ruby and Java Feature Comparison

Language Features	Ruby	Java
Extending All Classes	Yes	Non Final Classes Only
Duck Typing	Yes	No
Code Blocks	Yes	No
Regular Expressions	Native	Standard Library Support
Supports Using External Programs	Yes	Yes, but not as easily as Ruby
Network Programming	Standard Library Support	Standard Library Support
Typing Dynamic	Static	
Class Inheritance	Support mix-ins from multiple classes	Single
String Handling	Yes	Yes

What You Need

To follow along with the rest of the article, you need to install external Ruby libraries. The RubyGems library system makes this easy. Download it from RubyForge and follow the installation instructions for your operating system. (If you already have Ruby set up, you can verify that your setup includes RubyGems—many Ruby install packages do—by typing gem in a command shell to check for installation.) Having a central repository for libraries and a standard tool like RubyGems will save you a lot of time: no searching for the libraries you need, installing them, and using them in multiple projects.

Use the following commands to install the required gems:

```
gem query --remote # if you want to see all available remotely installable gems
sudo gem install activerecord
sudo gem install mysql # if you want to use MySQL
sudo gem install postgres-pr # optional: install "pure ruby" PostgreSQL interface
sudo gem install postgres # optional: install native PostgreSQL interface
sudo gem install ferret # a search library like Lucene (same API)
sudo gem install stemmer # a word stemming library for demonstrating extending a
class
gem query # to show gems locally installed
gem specification activerecord # info on gem (ActiveRecord in this example)
```

Under Mac OS X and Linux, you will need to run the gem installs using sudo; if you are a Windows user, remove "sudo" from the previous commands.

This article also assumes that you will open a Ruby irb shell as follows and keep it open while you're reading:

```
markw$ irb
>> s = "a b c"
=> "a b c"
>>
```

The example programs and code snippets are short enough to copy and paste into an irb interactive session.

Ruby String Handling

The Ruby String class provides a large set of string-processing methods that are more flexible than Java's string handling capabilities. This section shows a useful subset of Ruby's string processing. This code snippet shows how to combine strings, take them apart with slices, and then search for substrings (the examples to follow use the # character to make the rest of a line a program comment):

```
require 'pp'  # use the "pretty print" library. Defines the function 'pp'

# define some strings to use in our examples:

s1 = "The dog chased the cat down the street"
s2 = "quickly"

puts s1
puts s1[ 0..6]   # a substring slice up to and including character at index==6
puts s1[ 0...6]  # a substring slice up to (but not including) the character at
index==6

puts "He is a #{ s2}  dog #{ 1 + 6}  days a week." # expressions inside #{ }  are
inserted into a double quote string

puts "   test  ".strip  # create a copy of the string: the new copy has white
space removed

puts s1 + ' ' + s2 # string literals can also be formed with single quotes
puts s2 * 4

puts s1.index("chased") # find index (zero based) of a substring

s1[ 4..6] = 'giant lizard'  # replace a substring (/dog/ -> /giant lizard/)
puts s1

s2 = s2 << " now"  # the << operator, which also works for arrays and other col-
lections, copies to then end
puts s2

puts "All String class methods:"
pp s1.methods # the method "methods" returns all methods for any object
```

The output would be:

```
The dog chased the cat down the street
The dog
The do
He is a quickly dog 7 days a week.
test
The dog chased the cat down the street quickly
quicklyquicklyquicklyquickly
8
The giant lizard chased the cat down the street
quickly now
All String class methods:
```

```
[ "send",
  "%",
  "index",
  "collect",
  "[]=",
  "inspect", ......]     # most methods not shown for brevity--try this in irb
```

The << operator in the above example is really a method call. When evaluating expressions, Ruby translates infix operators into method calls. For example, the << operator in the following code adds the value of the expression on its right side to the value on the left side:

```
>> "123" << "456"
=> "123456"
>> "123".<<("456")
=> "123456"
>> 1 + 2
=> 3
>> 1.+(2)
=> 3
```

In the above example, using the form ".<<" is a standard method call.
Many classes use the << operator to add objects to a class-specific collection. For example, you will later see how the Ferret search library (a Ruby gem you have installed) defines the << operator to add documents to an index.

Modifying an Existing Class

The key to Ruby's versatility is the ability to extend all its classes by adding methods and data. I frequently extend core Ruby classes in my application, not in the original class source code. This likely seems strange to Java or even C++ developers, but this technique lets you keep resources for a project in one place and enables many developers to add application-specific functionality without "bloating" the original class. As a Java programmer, think how the limitations of Java constrain you: if you want to add functionality and data to an existing class, you must subclass.

The following listing shows how to add the method stem to the String class:

```
begin
  puts "The trips will be longer in the future".downcase.stem # stem is undefined
at this point
rescue
    puts 'Error:' + $!
end

require "rubygems"
require_gem 'stemmer'

class String # you will extend the String class
    include Stemmable # add methods and data defined in module Stemmable
end

puts "The trips will be longer in the future".downcase.stem
```

You will also find it useful to add methods and perhaps new class instance variables to existing classes in your application.

The next section looks at "duck typing," another example of the extreme flexibility that Ruby offers.

Ruby Duck Typing

In Java, you can call a method only on an object that is defined (with public, package, etc. visibility) in the object's class hierarchy. Suppose that you have a collection of objects and you want to iterate over each element in the collection, calling one or more methods. In Java, the objects would need to be part of the same class hierarchy or implement interfaces defining the methods that you want to call.

As you have probably already guessed, Ruby is much more flexible. Specific data types and classes are not required in Ruby's runtime method-calling scheme. Suppose you call method foo on an object obj, and then call method bar on the resulting object of this first method call as follows (the example shows two equivalent calls; when there are no method arguments, you can leave off the ()):

```
obj.foo.bar
obj.foo().bar()
```

The result of calling obj.foo will be some object, and whatever the class of this new object is, you would attempt to call method bar on it.

As another example, suppose you want to call the method name on each object in a collection. One element in this collection happens to be of an instance of class MyClass2 that does not have a method name defined. You will get a runtime error when you first try applying method name to this object. You can fix this by dynamically adding the method as follows:

```
class MyClass2
     def name
        "MyClass2: #{this}"
     end
end
```

Developers who are used to a strongly type checked language like Java likely will expect this "unsafe" flexibility to make their programs less reliable because the compiler or interpreter is not statically checking all type uses. However, any program bugs due to runtime type checking will be found quickly in testing, so there is no decrease in software reliability. Yet you get the benefits of a more flexible language: shorter programs and shorter development time.

Dealing with Missing Methods

Still skeptical about duck typing? Hang on, because now you are going to see another Ruby trick: how to handle missing methods for any Ruby class, starting with this simple example that applies two methods to a string object, one that is defined (length) and one that is undefined (foobar):

```
markw$ irb
>> s = "this is a string"
=> "this is a string"
>> s.length
=> 16
>> s.foobar
NoMethodError: undefined method `foobar' for "this is a string":String
         from (irb):3
```

You'll see an error thrown for the undefined method. So "patch" the String class by writing your own method_missing method:

```
>> class String
>>   def method_missing(method_name, *arguments)
>>     puts "Missing #{method_name}  (#{arguments.join(', ')})"
>>   end
>> end
=> nil
>> s.foobar
Missing foobar ()
=> nil
>> s.foobar(1, "cat")
Missing foobar (1, cat)
=> nil
>>
```

Whenever the Ruby runtime system cannot find a method for an object, it calls the method method_missing that is initially inherited and simply raises a NoMethodError exception. This example overrode this inherited method with one that does not throw an error, and it prints out the name and arguments of the method call. Now, redefine this method again, this time checking to see if the method name (after converting it to a string with to_s) is equal to foobar:

```
>> class String
>>   def method_missing(method_name, *arguments)
>>     if method_name.to_s=='foobar'
>>       arguments.to_s.reverse  # return a value
>>     else
?>       raise NoMethodError, "You need to define #{method_name}"
>>     end
>>   end
>> end
=> nil
>> s.foobar(1, "cat")
=> "tac1"
>> s.foobar_it(1, "cat")
NoMethodError: You need to define foobar_it
        from (irb):38:in `method_missing'
        from (irb):43
        from :0
>>
```

If the method name is equal to foobar, this example calculates a return value. Otherwise, it throws an error.

Ruby Code Blocks

Ruby uses code blocks as an additional way to iterate over data. These blocks offer more flexibility and power than the limited iteration functionality built into the Java language. The previous example showing basic string functionality used the stemmer gem to find the word stems of a string containing English words. The following example uses the String split method to tokenize a string using the space character as a word delimiter and then passes a code block defined using the { and } characters to mark the beginning and end of a code block (you also can use begin and end). Local variables in a block are listed between two | characters:

```
puts "longs trips study studying banking".split(' ')
puts "longs trips study studying banking".split(' ').each { |token| puts
"#{ token}  : #{ token.stem} "
```

This code snippet produces the following:

```
longs
trips
study
studying
banking

longs : long
trips : trip
study : studi
studying : studi
banking : bank
```

You can see another good use of code blocks in the following example, which uses the Array collect method. The collect method processes each array element and then passes it to a code block:

```
require 'pp'
pp [ "the", "cat", "ran", "away"] .collect { |x| x.upcase}
pp [ "the", "cat", "ran", "away"] .collect { |x| x.upcase} .join(' ')
```

In this example, the code block assumes that the elements are strings and calls the upcase method on each element. The collect method returns the collected results in a new array. It also uses the method join to combine all the resulting array elements into a string, separating the elements with the space character. This is the output:

```
[ "THE", "CAT", "RAN", "AWAY"]
"THE CAT RAN AWAY"
```

Writing Methods That Use Code Blocks

You can use the yield method to call a code block passed to a method or function call. The following example uses the method block_given? to call yield conditionally if a code block is supplied. The method yield returns a value that is printed:

```
def cb_test name
      puts "Code block test: argument: #{ name} "
      s = yield(name) if block_given?
      puts "After executing an optional code block, =#{ s} "
end
```

This example calls function cb_test, first without a code block and then with one:

```
>> puts cb_test("Mark")
Code block test: argument: Mark
After executing an optional code block, =
nil
=> nil
>> puts cb_test("Mark") { |x| x + x}
```

```
Code block test: argument: Mark
After executing an optional code block, =MarkMark
nil
=> nil
>>
```

The string value Mark is passed as an argument to yield, and inside the code block the local variable x is assigned the value Mark. The return value from the code block is MarkMark.

Ruby Regular Expressions

Ruby has built-in support for handling regular expressions using the class Regexp. Java's java.util.regex APIs offer similar functionality but regular expression support in Ruby definitely has a more native feel to it. You can create a regular expression object by either directly using a method call like Regexp.new("[a-e]og") or enclosing a regular expression between slash characters like /[a-e]og/. You can find good tutorials on both regular expressions and on Ruby's regular expression support on the web; this simple example shows only using the =~ operator:

```
=> 4
>> "the dog ran" =~ /[a-e]og/
=> 4
>> "the zebra ran" =~ /[a-e]og/
=> nil
```

Ruby Network Programming

Ruby has a great standard library for network programming as well. I frequently use Ruby for collecting data from the Internet, parsing it, and then storing it in XML or a database.

Ruby Document Indexing and Search Using the Ferret Library

By now, you have installed the Ruby gem called ferret. Ferret is the fastest indexing and search library based on Java Lucene (even faster than the Common Lisp version, Montezuma). One interesting fact about the Ferret library is that during development the author David Balmain eventually wrote most of it in C with a Ruby wrapper. The lesson is that if you start to use Ruby and have performance problems, you can always recode the time-critical parts in C or C++. Ferret defines a few classes that you will use in your own applications once you adopt Ruby:

- Document represents anything that you want to search for: a local file, a web URL, or (as you will see in the next section) text data in a relational database.
- Field represents data elements stored in a document. Fields can be indexed or non-indexed. Typically, I use a single indexed (and thereby searchable) text field and then several "meta data" fields that are not indexed. Original file paths, web URLs, etc. can be stored in non-indexed fields.
- Index represents the disk files that store an index.
- Query provides APIs for search.

Indexing and Searching Microsoft Word Documents

The following is the Ruby class I use for reading Microsoft Word documents and extracting the plain text, which is an example of using external programs in Ruby:

```
class ReadWordDoc
  attr_reader :text
  def initialize file_path
```

```
    @text = `antiword #{file_path}`    # back quotes to run external program
  end
end
```

The "trick" here is that I use the open source antiword utility to actually process Word document files. You can run any external program and capture its output to a string by wrapping the external command in back quotes. Try the following under Linux or OS X (for Windows try `dir`):

```
puts `ls -l`
```

This example prints the result of executing the external ls (Unix list directory) command.
The following Ruby script enters a Word document into an index (plain text files are easier—try that as an exercise):

```
require 'rubygems'
require 'ferret'
include Ferret
include Ferret::Document
require 'read_word_doc' # read_word_doc.rb defines class ReadWordDoc

index = Index::Index.new(:path => './my_index_dir')  # any path to a directory

doc_path = 'test.doc'                      # path to a Microsoft Word
doc_text = ReadWord.new(doc_path).text  # get the plain text from the Word file

doc = Document.new
doc << Field.new("doc_path", doc_path, Field::Store::YES, Field::Index::NO)
doc << Field.new("text", doc_text, Field::Store::YES, Field::Index::TOKENIZED)
index << doc

index.search_each('text:"Ruby"') do |doc, score|  # a test search
  puts "result: #{index[doc]['doc_path']} : #{score}"    # print doc_path meta
data
  puts "Original text: #{index[doc]['text']}"             # print original text
end

index.close  # close the index when you are done with it
```

Notice how short this example is. In 24 lines (including the class to use antiword for extracting text from Word documents), you have seen an example that extracts text from Word, creates an index, performs a search, and then closes the index when you are done with it. Using Ruby enabled you to get complex tasks done with very few lines of code. Had you coded this example in Java using the very good Lucene library (which I've done!), the Java program would be much longer. Shorter programs are also easier and less expensive to maintain.

This example uses Word documents, but OpenOffice.org documents are simple enough to be read. With about 30 lines of pure Ruby code, you can unzip a document and extract the text from the content.xml element in the unzipped XML data stream. (XML processing is simple in Ruby, but it is beyond the scope of this article.)

Ruby Complements Java

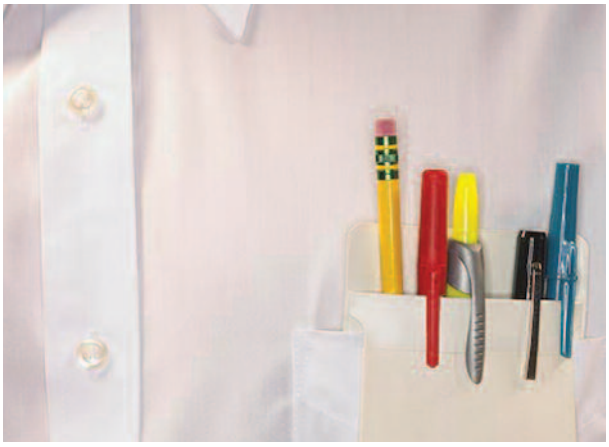
The cost of software development and maintenance is usually the largest expense for a company's IT budget—much more expensive than servers, Internet connectivity, etc. The use of Ruby can greatly reduce the cost of building and maintaining systems, mostly because programs tend to be a lot shorter (For me, the time spent per line of code is similar for most programming languages I use).

OK, so when should you use Java? I have used the Java platform for building systems for my consulting customers for over 10 years, and I certainly will continue using Java. A well-built, Java-based web application will run forever—or at least until servers fail or have to be rebooted for hardware maintenance. My confidence comes from seeing systems run unattended for months on end with no problems. My advice is to continue using Java on the server side for large systems and to start using Ruby for small utility programs. For my work, I view Java and Ruby as complementary, and not as competitors. Use the best tool for each task.

Ruby for C# Geeks

By Dave Dolan

Attention C# developers: C# is good for a great many things, but it's not the best language for everything. In fact, there is no such thing as "the best language." It all depends on what you're trying to do, and your personal preference and familiarity with the languages you have available. Although the Church-Turing Thesis suggests that anything you can do in one complete language, you can also do in another, the truth is not all languages are created equal. You can accomplish simple tasks with complex code in some languages and complex tasks with simple code in others. C# sometimes falls into both categories, but more often than not, it requires a bit more than its fair share of effort. Strangely enough, a language that allows you to accomplish simple tasks with simple code is rare. Enter Ruby, an interpreted, dynamically typed language that enables just that.



Jupiterimages

Many would say that the main difference between Ruby and C# is that Ruby is a dynamic language whereas C# isn't. However, referring to C# as a static language really isn't right because you wouldn't apply that term to an entire language as you would to one of the dynamic variety. Ruby really differs from C# in that its code is not actually compiled into an intermediate executable form before it is run. Instead, Ruby has at its heart a text-driven interpreter. This means that the expressions and statements in a Ruby program are evaluated as the interpreter passes over them. In C#, you must first compile the code to an .exe or .dll file to be able to run it.

In requiring compilation, C# encapsulates an opportunity to check syntax and optimize the runtime efficiency of the code before it's ever run. On the other hand, all of the declaration and specification that goes into setting up your code with all of the necessary information to allow the compiler to perform these tricks will slow you down when these features aren't necessary or desired. You might use a language like Ruby, with looser guidelines, to test your algorithmic theories or rapidly prototype an application. Sometimes you just need to format a couple of text files, and C# isn't exactly friendly in cases where you just want to automate something as simple as a command line.

This article offers a brief introduction to the Ruby language from the perspective of a C# developer. You'll learn the differences between the languages' features through line-by-line examinations of identical programs built in each one.

Same Results, Simpler Code

Every programming language tutorial I've ever read has what is (often sardonically) known as the "obligatory Hello World! example." I'm not very fond of that terminology, so I've spruced up the sample application for this article to the "slightly interesting Hello Foo! example." The snippets to follow show two programs (one in C#, the other in Ruby) that produce exactly the same output for my Hello Foo! example.

First, here's the C# version:

```
// Hello Foo! in C#


using System ;

namespace TestCSharp
{
    class MainClass
    {

        public static void Main(string[] args)
        {
            Console.WriteLine("Hello Foo!"); //say hello
        }
    }
}
```

As you can see, C# requires you to specify a lot of structure and actually tell the compiler that you're going to write a class. Specifically, here you tell it you're going to write a class with a well-known method called Main that a console application will use as the first method to call (known as an entry point) when the program executes. The required verbosity is somewhat mitigated by Visual Studio or any other syntax-helping editor that allows for templates and IntelliSense; however, the level of code complexity is still there regardless of whether or not you have to actually type every letter.

First, compile the C# program (If you're using Visual Studio or MonoDevelop, you can simply press F5.), and then you can run it:

```
# Hello Foo! in Ruby

puts "Hello Foo!" # say hello
```

Running this little snippet requires that you simply invoke the interpreter and then provide the name of the script file (hellofoo.rb):

```
C:\>ruby hellofoo.rb
Hello Foo!

C:\>
```

Since the Ruby interpreter assumes all of the structural information is there, you don't have to write it like you would in C#. The interpreter assumes that the first bunch of code you write without an enclosing class or module declaration is analogous to it appearing within the Main method of the entry point class. Very handy.

Note the differences in syntax between the languages as well:

- Semi-colons aren't used to delimit the end of a statement in Ruby.
- Ruby's built-in puts method is actually like C#'s Console.Writeline in that it will display the result of the .to_s method of the parameter object, which in C# you would write as .ToString().
- Parentheses are optional for method calls in Ruby. Most of the time, you simply omit them—particularly when you don't pass any parameters.
- The // style of commenting in C# is replaced by the # notation in Ruby. Authors Note: Because its syntax is regarded as self-explanatory, a common belief is that Ruby rarely requires comments. I'm a little skeptical of this idea, but I will admit that it's often notably easier to understand Ruby code as a human reader than it is the C# equivalent.

Dynamically Typed Means Faster Coding

Not only is Ruby an interpreted (i.e., dynamically evaluated) language , it is also dynamically typed. So the variables in Ruby do not require the specification of their types before you use them. The interpreter will infer variable types as they are assigned values. To add another twist, you don't have to declare variables at all! This is a common feature of many interpreted languages, and even a few compiled languages. To see what I'm talking about, consider the following Ruby snippet:

```
#Ruby

abc = 1   #abc is a Fixnum (integer)
puts abc

abc = "Rubber Baby Buggy Bumpers"   # now it's a string!
puts abc
```

You can see that you have to declare neither the variables nor their types in Ruby, and you can change the type by assigning it a different value—right in the middle of the running program. Very dynamic! If you tried to do something like that in C#, the program wouldn't even compile. C# requires statically defining types for variables, which allows the compiler to catch any errors that may arise when types don't match the ways in which they are used. However, it's faster and easier to throw together a script that doesn't go to all of this trouble in Ruby. Both approaches may have a time and a place, but you can choose the one that best suits the problem you're trying to solve.

The Ruby vs. C# Feature Rundown

The following is a simple example program in Ruby that demonstrates a variety of the features often seen in C# programs. (It's another "classic" example program used in many programming tutorials.) It is chock full of new stuff I haven't discussed yet, but I'll explain the entire program snippet by snippet afterwards. If it's still opaque after that, fear not because I provide a structurally similar C# version so you can compare lines for yourself:

```
#gessnum.rb
class Game
   def initialize(maxNum)
      @num = rand(maxNum)
      puts [ "\n\nMagic Number Game!\n",
           "-----------------\n\n",
             "I'm thinking of a magic number between 0 and #{maxNum}\n",
             "Would you like to guess my number?  (y/n) [n]:"]
      playNow = gets or "n"
      if playNow.chop == "y"
```

```
            play
        else
            puts "OK, Bye!"
        end

    end

    def play
        loop do # infinite loop!

            puts "\nNumber?!?"
            attempt = gets or break

            case attempt.to_i <=> @num
                when 1 # attempt > @num
                    puts "Guess Lower!"
                when -1 # attempt < @num
                    puts "Think Bigger, Mac."
                else # only have == left... so...
                    puts "Spot on! That's it! Bye!"
                    break # hop out of our loop!
            end
        end
    end
end

Game.new(100)
```

Here's a sample run:

```
C:\proj\ruby>ruby sample.rb


Magic Number Game!
------------------

I'm thinking of a magic number between 0 and 100
Would you like to guess my number? (y/n) [n]:
y

Number?!?
50
Guess Lower!

Number?!?
25
Think Bigger, Mac.

Number?!?
37
Spot on! That's it! Bye!
```

At the very beginning of the code, I define a class. Notice how it's just the word "class" followed by the class name—no curly anything, and I haven't set a namespace.

The Game class contains two methods: initialize, the constructor, and play, the main body of the game. Methods are designated simply by the def keyword, and are terminated by the end keyword. In Ruby, all constructors are called initialize, and they are called when an object is instantiated (more on this in a minute).

The following snippet designated a variable called @num in the constructor. The variable was set to a random number between 0 and maxNum:

```
@num = rand(maxNum)
```

Any variable inside a class that starts with a @ sign is known as an instance variable (like a field in C#). Just like in C#, the default for an instance variable is to be private, meaning that folks outside the class cannot see it or read its value. Why would I bother putting it in a field instead of a local variable? Well, for the same reasons that I'd do so in C#; I need to access it from other methods, namely the play method.

This command printed an array as a string:

```
      puts [ "\n\nMagic Number Game!\n",
           "------------------\n\n",
             "I'm thinking of a magic number between 0 and #{maxNum}\n",
             "Would you like to guess my number? (y/n) [n]:"]
```

It's much easier to just declare an array of strings and issue a single puts command for it to make a puts for each one.

The #{maxNum} portion is a Ruby trick known as string interpolation. The value of maxNum will be substituted for the occurrence of #{maxNum} in the string. This is roughly analogous to the String.Format() idiom in C#.

I set the value of a local variable 'playNow' to the result of the gets function, which reads a string from the input stream:

```
playNow = gets
   if playNow.chop == "y"
      play
   else
      puts "OK, Bye!"
   end
```

I had to compare playNow to "y" (for yes) to make sure the user actually wants to play the game. But wait, you say, what's that .chop business? That extension will drop the last character from the value, which would be the newline character, because gets records the newline generated by the enter key when it reads the input stream. So, if the program gets a "y" it invokes the play method, otherwise it kindly says goodbye.

Normally, trying to run code from within the constructor may not be such a great idea or most objects, but for the sake of this paltry example game, it's no big deal:

```
def play
      loop do # infinite loop!

         puts "\nNumber?!?"
```

```
        attempt = gets or break

        case attempt.to_i <=> @num
           when 1 # attempt > @num
              puts "Guess Lower!"
           when -1 # attempt < @num
              puts "Think Bigger, Mac."
           else # only have == left... so...
              puts "Spot on! That's it! Bye!"
              break # hop out of our loop!
        end
     end
  end
```

The play method is an infinite loop, meaning that it will continue to execute until the process is terminated or something inside the loop issues a break statement. I prompt the user for a number and store it in the local variable attempt.

Listing 1. The C# Version of the Guess the Number Game

```
using System;

namespace GuessMe
{
 class Game
 {
 int num;

 public Game(int maxNum)
 {
 Random r = new System.Random();
 num = r.Next(maxNum);

 Console.Out .WriteLine(
 String.Join("\n", new string[] {
 "\n\nMagic Number Game!",
 "------------------\n",
 String.Format("I'm thinking of a magic number between 0 and {0}", maxNum),
 "Would you like to guess my number? (y/n) [n]:"}
 )
 );

 string playNow = Console.In.ReadLine();

 if (playNow == "y")
 {
 play();
 }
 else
 {
 Console.Out.WriteLine("OK, Bye!");
 }
```

continued

The next bit is a little strange for a C# fan, but it actually is not all that different from a switch statement. The expression it is case-ing on (switching on) is attempt.to_i <=> @num. The first part, attempt.to_I, converts the string value attempt to an integer. (The Ruby class of objects holding small integer values is actually called Fixnum.) It's a built-in method of the %(String) class, which itself is a built-in type. The <=> operator is analogous to the C# idiom of CompareTo(). If the values are equal, <=> returns the integer 0. If the left is less than the right, an integer value of -1 is returned, and a 1 is returned if the left side of the expression is greater than the right. Basically, this is a switch for the three possible values, but instead of the C# way (switch… case), it's the Ruby way (case…when):

The very last line is the actual body of the main program:

```
Game.new(100)
```

The only thing that happens here is that an instance of the Game class is created by calling the .new method with the parameter of 100. In Ruby, .new is a special method that will invoke the initialize method, the constructor of the object. Notice the object isn't assigned. Nobody needs to see the object, so it's not stored.

Listing 1. The C# Version of the Guess the Number Game

```
}

void play()
{
string attempt = null;
while(true)
{

Console.Out.WriteLine("\nNumber?!?");
attempt = Console.In .ReadLine ();

switch(Convert.ToInt32(attempt).CompareTo(num))
{
case -1:
Console.Out.WriteLine("Think Bigger, Mac.");
break;
case 1:
Console.Out .WriteLine("Guess Lower!");
break;
default:
Console.Out .WriteLine("Spot on! That's it! Bye!");
return;
}
}
}

}
class GuessGame
{
public static void Main(string[] args)
{
new Game(100);
```

The Ruby Mixin Feature

One trick I haven't covered is the ability of Ruby to change an existing class that has already been defined. This is called a mixin because it allows you to mix in your code with code that already exists. You can even create mixins that alter the built-in types in Ruby, effectively altering the way the entire language operates. To add even more variability, mixins allow you to import classes, methods, or extend classes with features of a Module (like a static class in C# 2.0) by modifying either the classes themselves or just particular instances of the classes!

You also can re-open an existing class definition and inject your own methods in, or override ones that are already there. For example, I could redefine the .to_s method of the Fixnum class, which is the class that all integers take on by default, to return something like "I, for one, welcome our robot overlords" every time. (The wisdom of doing something like this is of course questionable.) For the sake of demonstrating how very open the entire structure of Ruby is, here's how you can do this:

```
Fixnum.class_eval do
  def to_s
    "I, for one, welcome our robot overlords."
  end
end

q = 123
puts q.to_s
```

Of course, don't try this at home, or at least don't do it in production code.

For Further Study

The Ruby language offers many more features and goodies than I could cover in one article, but absorbing them a few at a time from here on will be no harder than what this tutorial has shown. For a quick perusal of how Ruby generally compares to C# and Java, see Table 2.

Language Feature	Ruby	C#	Java
Object Oriented	Yes	Yes	Yes
Compiled/Interpreted	Interpreted	Compiled	Both (usually compiled)
Variable Typing	Dynamic	Static	Static
Native Mixins	Yes	No	No (some add-on libraries offer limited support)
Closures	Yes	In version 2.0 +	No
Reflection	Yes	Yes	Yes
Multi-Threading	Yes	Yes	Yes
Regular Expressions	Native	Standard library support	Standard library support
Static and Instance Methods	Yes	Yes	Yes
Type Inference	Yes	No	No
Strong Typing	Yes*	Yes	Yes

** Ruby is still strongly typed, but the type can be inferred and easily changed at run-time. Note that this is not the same as statically typed.*

One of the particularly nice things about Ruby is the voluminous mass of documentation generated by the ever-expanding Ruby user community. If you would like to learn some more about Ruby, you should check out the industry standard tongue-in-cheek guide known as Why's (Poignant) Guide to Ruby. If humor isn't your ideal way to learn a language, then you might want to see the more formal Programming Ruby, which is actually quite authoritative on the subject. No matter what your preferences are, you can get wherever you want to go with Ruby from the official Ruby Language site (http://www.ruby-lang.org/).

The Road to Ruby from C++

By Michael Voss

Get the code for this article at: http://assets.devx.com/sourcecode/19155.zip

If you're a C++ developer who's curious about all of the hype surrounding the Ruby programming language, this article is for you. It provides a high-level overview of the key differences between C++ and Ruby, and then presents a small, complete example application implemented with each language.

Be forewarned, however: learning Ruby can be a very frustrating experience! Because once you become familiar with this powerfully concise language, you might find returning to C++ a bitter pill to swallow.

Jupiterimages

A High-Level Language Comparison and a Running Example

C++ is a statically typed, compiled language that has hybrid object orientation. Its static typing means that the type of every expression and variable is known at compile-time, allowing significant correctness checking before the program executes. Its hybrid object orientation means that it defines non-object primitive types such as int and float, and functions can exist outside of objects. The Ruby programming language is designed to let you write code quickly and concisely. Unlike C++, it is a very dynamic, interpreted language that includes a powerful set of libraries. While it is often referred to as a scripting language, it is a pure objected-oriented language that has sufficient expressiveness for general-purpose applications.

In Ruby, variables do not need to be declared and are free to change type from statement to statement. So the following code, where the variable x changes from a FixNum (an integer that fits within a native machine word) to a String to an Array, is a perfectly legal sequence of Ruby code:

```
x = 10
x += 4
x = "My String"
x = [ 1, "My String",
Hash.new ]
```

A significant downside of Ruby's dynamism is its use of an interpreter. Ruby's runtime performance just can't compare to a compiled language like C++. So even if you find yourself in love with the features of Ruby, you're likely better off sticking with C++ if you really need runtime efficiency.

Having digested some of the key differences between C++ and Ruby, you're now ready to examine the small, complete example application implemented with each language. The application calculates the total number of occurrences for each word found in a set of files with a given directory, and generates an XML file that summarizes these occurrences as output. Listing 1 shows the C++ implementation, and Listing 2 shows the Ruby implementation.

The Basics of Classes and Variables

Both versions define the following three classes:

1. A class that represents the total number of occurrences of a word across all files
2. A class that is derived from this class and extended to also maintain the occurrences of each word by file
3. A counter class that reads and parses the files, creates and updates the word counts, and outputs the XML file

The first class is defined in Listing 1 at line 22 and in Listing 2 at line 4. Both implementations maintain a string word and a total_count variable. In Ruby, instance variables are preceded by a "@" symbol, and therefore Listing 2 has @word and a @total_count. Local variables have no prefix and global variables have a "$" symbol as a prefix.

The C++ code uses a struct to declare this class. Therefore, the variables word and a total_count are public by default. Ruby, however, does not allow access to instance variables from outside of the object; all instance variables are private. You'll learn more about access control later, but for now focus on adding the needed accessor methods. Luckily, as the statement at line 7 of Listing 1 demonstrates, adding these methods is no chore. You can automatically generate the needed get accessor methods by listing the variables after attr_reader.

Both implementations of this class also define a constructor that takes the word string, a method add that incre-

Listing 1. The C++ Implementation

This C++ application calculates the total number of occurrences for each word found in a set of files with a given directory, and generates an XML file that summarizes these occurrences as output.

```
// for directory manipulation
#include <unistd.h>
#include <sys/types.h>
#include <dirent.h>

// streams and strings
#include <iostream>
#include <fstream>
#include <string>

// STL collections
#include <map>
#include <set>

struct my_compare {
  bool operator() (const std::pair<std::string, int> &f1,
                   const std::pair<std::string, int> &f2) {
    return f1.second < f2.second;
  }
};

struct word_count {
  std::string word;
  int total_count;
  typedef std::multiset< std::pair<std::string, int>, my_compare >
          file_set_t;

  word_count(std::string _word) : word(_word), total_count(0) {}

  virtual void add(std::string filename) {
```

continued

ments the counters, and a method file_occurrences that returns a data structure that holds per-file information. As shown on line 9 of Listing 2, the class constructors in Ruby are named initialize.
If you ignore the "include Comparable" in the Ruby code until later, the remainder of the implementation for this base class is then fairly straightforward for both languages.

Inheritance and Polymorphism

The next class defined in both files inherits from this simple base class, extending it to also track the total number of occurrence for the word in each file. The C++ implementation uses ": public word_count" to indicate the inheritance. The Ruby implementation uses "< WordCount". In both cases, the extended class adds a hash map to store the occurrence count associated with each processed file. The method add is extended to update the hash map, and the method file_occurrences returns this information.

There are few key differences between inheritance in C++ and inheritance in Ruby. Ruby, unlike C++, does not support multiple inheritance but does support mixins. The "include Comparable" found on line 16 of Listing 2 is an example. A Module is a set of function definitions. You can't create an instance of Module; you can only include it into class definitions. In this case, Module Comparable defines the comparison operators (<, >, ==) in terms of

Listing 1. The C++ Implementation

```
    total_count += 1;
  }

  virtual file_set_t file_occurrences() {
     return file_set_t();
  }
};

bool operator< (const std::pair<std::string, word_count *> &wc1,
                const std::pair<std::string, word_count *> &wc2) {
  return wc1.second->total_count < wc2.second->total_count;
}

bool operator< (const word_count &wc1, const word_count &wc2) {
  return wc1.total_count < wc2.total_count;
}

struct word_and_file_count : public word_count {
  std::map<std::string,int> file_count;

  word_and_file_count(std::string _word)
    : word_count(_word) {}

  void add(std::string filename) {
    if (file_count.find(filename) == file_count.end())
      file_count[ filename]  = 0;
    file_count[ filename]  += 1;
    total_count += 1;
  }

  word_count::file_set_t file_occurrences() {
     return word_count::file_set_t ( file_count.begin(),
                                     file_count.end() );
  }
```

continued

the <=> operator. So by defining <=> and including Module Comparable, you get the other comparison operators for free.

In C++, you sometimes rely on inheritance combined with virtual functions to enable polymorphism. A pointer x of type T * can point to an object of type T or any object with a type below T in the class hierarchy. A virtual method invoked through x is resolved by walking up the class hierarchy, starting from the type of the object pointed to by x.
Ruby on the other hand uses duck typing—if something looks like a duck, swims like a duck, and quacks like a duck, then it's a duck. Take the following code for example:

```
def my_method(x)
  x.print_hello
end
```

For Ruby, it doesn't matter what type x is. If the object x has a method print_hello, the code will work. So unlike C++, which would require the objects to inherit from a common base type, you can pass objects of unrelated types to my_method, as long as they all implement print_hello.

Listing 1. The C++ Implementation

```
};

template <typename W>
class word_counter {
private:
  std::map<std::string, word_count *> word_counts;
  const std::string directory_name;

  void count_words_in_file(const std::string &filename) {
    int i = 0;
    std::ifstream file(filename.c_str());

    if ( file.is_open() ) {
      while (file.good()) {
        std::string line;
        getline(file,line);
        char c = line[ i=0] ;
        while ( c != '\0' ) {
          std::string buffer;
          while ( c != '\0' && !isalpha(c) ) {
            c = line[ ++i] ;
          }
          while ( c != '\0' && isalpha(c) ) {
            buffer += c;
            c = line[ ++i] ;
          }
          if (buffer.length()) {
            if (word_counts.find(buffer) == word_counts.end())
              word_counts[ buffer] = new W(buffer);
            word_counts[ buffer] ->add(filename);
          }
        }
      }
    }
```

continued

Visibility and Access Control

The final class in the example applications is implemented starting at line 67 in Listing 1 and line 54 in Listing 2. This class iterates through all of the files in the provided directory, breaking the file into tokens and counting the occurrences of each word. It also defines a method to dump the XML output.

Both C++ and Ruby support public, protected, and private members. In the example, the method count_words_in_file is declared as private in both implementations.

In both C++ and Ruby, public methods can be called by anyone, and protected methods can be called only by objects of the same class or objects that inherit from the defining class. The semantics of private differ between C++ and Ruby, however. In C++, methods are private to the class, while in Ruby they are private to the instance. In other words, you can never explicitly specify the receiver for a private method call in Ruby.

Blocks and Closures

One feature of Ruby for which C++ has no good counterpart is its support of blocks. The most common use for blocks is iteration. You'll find examples of iteration with blocks in Listing 2's implementation of the WordCounter class at lines 65, 73, and 76.

Listing 1. The C++ Implementation

```
  }

public:
  word_counter(const std::string &_directory_name)
       : directory_name(_directory_name) {}

  void count_words() {
    char *cwd = getcwd(NULL,0);

    if (chdir(directory_name.c_str()))  {
       std::cerr << "Could not open directory" << std::endl;
       exit(1);
    }

    DIR *d = opendir(".");
    if (!d) {
       std::cerr << "Could not open directory" << std::endl;
       exit(1);
    }

    while (struct dirent *e = readdir(d)) {
      std::string filename = e->d_name;
      count_words_in_file(filename);
    }
    chdir(cwd);
    delete cwd;
  }

  void dump_results() {
    typedef std::multiset< std::pair< std::string, word_count * > >
            wc_set_t;
    std::cout << "<counts>" << std::endl;
    wc_set_t wc_set(word_counts.begin(), word_counts.end() );
    for (wc_set_t::const_reverse_iterator it = wc_set.rbegin();
```

continued

Consider the code at line 65 for example:

```
Dir.foreach(".") {  |filename|
    count_words_in_file filename
}
```

The class Dir is used to inspect directories. The method foreach is passed two arguments: the string "." and the block of code in parentheses, which in turn specifies an argument filename within the vertical bars. The method foreach iteratively invokes the block, passing in the names of each file found in the current working directory ".". This simple feature can save significant keystrokes and also leads to very readable code.

Blocks also are useful for more than just iteration. Line 48 uses a block to specify the comparison function to use for sorting an array of pairs:

```
def file_occurrences
  return @file_hash.sort {  |x,y| -(x[1]<=>y[1]) }
end
```

Listing 1. The C++ Implementation

```
         it != wc_set.rend(); it++) {
      std::cout << "<word occurences=\"" << it->second->total_count
                << "\">" << std::endl << it->second->word << std::endl;
      word_count::file_set_t file_set = it->second->file_occurrences();
      for (word_count::file_set_t::const_reverse_iterator
             fit = file_set.rbegin(); fit != file_set.rend(); fit++) {
        std::cout << "<file occurences=\"" << fit->second << "\">"
                  << fit->first <<  "</file>" << std::endl;
      }
      std::cout << "</word>" << std::endl;
      delete it->second;
    }
    std::cout << "</counts>" << std::endl;
  }

  void run() {
    count_words();
    dump_results();
  }
};


int main(int argc, char *argv[]) {
  char *dir = argv[1];

  if (!strcmp(argv[1],"--no-file-info")) {
    word_counter<word_count>(argv[2]).run();
  } else {
    word_counter<word_and_file_count>(argv[1]).run();
  }
  return 0;
}
```

The expression x <=> y is -1 if x < y, 0 if x == y, and 1 if x > y. The above code returns an array of pairs, where each pair consists of a String and a FixNum. The block specifies that the second element of each pair (the FixNum) should be compared using the negation of the <=> operator. This method therefore returns the word occurrences pairs in decreasing order of occurrences.

At line 15 of Listing 1, the C++ example code also specifies a comparison function to be used for ordering pairs. However, lacking support for anonymous functions, the comparison is implemented as a function object, later used to define the STL multiset at line 25.
The blocks in Ruby are so useful in part because they are closures. A closure captures the context in which it is defined, so it can refer to local variables found within the scope of the definition. Take for example the code at line 76 of Listing 2:

```
wc.file_occurrences.each {  |pair|
   f = e.add_element "file", {"occurrences"=>"#{pair[1]}"}
   f.add_text pair[0]
}
```

The expression wc.file_occurrences returns an array of pairs. The array's method each is then invoked with the subsequent block as an argument. It's important to note that the block will be invoked from within the method each. However, because the block is a closure, it can still access the object e (which represents an XML element) that was in the local scope of the method where the block was defined.

While you can use C++ function objects to implement much of the functionality described above, I believe that the elegance and readability of blocks speak for themselves.

Table 1. A Comparison of Features in C++, Java, and Ruby

Language Feature	C++	Java	Ruby
Type System	Static	Mostly static	Dynamic
Object Orientation	Hybrid	Pure (with primitive types)	Pure
Inheritance	Multiple	Single and interfaces	Single and mixins
Overloading	Method and Operator	Method	Operator
Polymorphism	Yes	Yes	Yes
Visibility/Access Controls	Public, protected, private, and friends	Public, protected, package, and private	Public, protected, private (instance)
Garbage Collection	No	Yes	Yes
Generics	Yes (templates)	Yes (generics)	No
Closures	No (function objects)	Yes, with some limitations (inner classes)	Yes (blocks)
Exceptions	Yes	Yes	Yes

A Wide Range of Libraries, Regular Expressions

Another clear advantage that Ruby has over C++ is the vast collection of libraries that come with the standard distribution, as well as its support for regular expressions. For example, compare the ad-hoc implementation of an XML generator in method dump_results in Listing 1 to the use of the REXML library in Listing 2. Next, note the use of the pseudo-standard dirent.h for working with directories in C++ to that of the class Dir in the Ruby implementation. Finally, compare the ad-hoc parsing of the files at line 77 of Listing 1 to the much more concise code at line 90 of Listing 2.

While many available C++ libraries, such as Boost, provide a wide range of utilities, Ruby provides many of these features as part of the standard distribution. So out-of-the-box, the Ruby programming language and its standard libraries simplify many of the more common programming tasks when compared to C++.
A Summary of Language Features

To summarize, Table 1 presents some of the key features of C++ and Ruby discussed in this article (with Java included for comparison).

Listing 2. The Ruby Implementation

This Ruby application calculates the total number of occurrences for each word found in a set of files with a given directory, and generates an XML file that summarizes these occurrences as output.

```
require 'rexml/document'
require 'set'

class WordCount

  # read-only attributes
  attr_reader :word, :total_count

  def initialize(word)
  @word = word
  @total_count = 0
  end

  # Use a mixin of Comparable module to get comparisons for free
  # we must define <=>
  include Comparable

  def <=>(other)
    return @total_count <=> other.total_count
  end

  def add(filename)
    @total_count += 1
  end

  def file_occurrences
    return []
  end
```

continued

Listing 2. The Ruby Implementation

```
end

class WordAndFileCount < WordCount

  # read-only attributes
  attr_reader :file_hash

  def initialize(word)
  super(word)
  @file_hash = Hash.new
  end

  def add(filename)
  @total_count += 1
  v = @file_hash[ filename]
  @file_hash[ filename]  = (v == nil) ? 1 : v + 1
  end

  def file_occurrences
    return @file_hash.sort {  |x,y| -(x[ 1]<=>y[ 1]) }
  end

end

class WordCounter

  def initialize(directory_name)
    @directory_name = directory_name
    @word_count = Hash.new
  end


  def count_words
    pwd = Dir.pwd
    Dir.chdir @directory_name
    Dir.foreach(".") {  |filename|
       count_words_in_file filename
    }
    Dir.chdir pwd
  end

  def dump_results
    root = REXML::Element.new "counts"
    @word_count.values.sort.reverse.each {  |wc|
      e = root.add_element "word", {"occurences"=>"#{wc.total_count}"}
      e.add_text wc.word
      wc.file_occurrences.each {  |pair|
        f = e.add_element "file", {"occurences"=>"#{pair[ 1]}"}
        f.add_text pair[ 0]
      }
```

continued

Listing 2. The Ruby Implementation

```
    }

    doc = REXML::Document.new
    doc << REXML::XMLDecl.new
    doc.add_element root
    doc.write $stdout
  end

  private

  def count_words_in_file(filename)
    return if File.directory? filename

    File.open(filename) {  |file|
      file.each_line {  |line|
        words = line.split(/[^a-zA-Z]/)
        words.each {  |w|
          next if w.size == 0
          @word_count[w] = $count_gen.call(w) if @word_count[w] == nil
          @word_count[w].add filename
        }
      }
    }
  end

end

if ARGV.include?("--no-file-info") then
ARGV.delete("--no-file-info")
$count_gen = lambda {  |word| WordCount.new word }
else
$count_gen = lambda {  |word| WordAndFileCount.new word }
end

counter = WordCounter.new ARGV[0]
counter.count_words
counter.dump_results
```

Five Essentials For Your Ruby Toolbox

By Peter Cooper

Ruby, the dynamic, interpreted programming language that adopts a pure object-oriented approach, has been gaining popularity among programmers. While many of Ruby's new converts are primarily developing Web applications, Ruby has a great history (much of it in Japan) as a separate language in its own right. Only recently are developers in the West beginning to understand the significant advantages Ruby has over other scripting languages such as Perl and Python, or even more established enterprise-level languages such as Java.

Jupiterimages

If you are one of those who recently boarded the Ruby bandwagon, you may not be sure which of the many available Ruby tools and libraries are most helpful to your development. This article looks at five essential tools and libraries that Ruby developers should have in their arsenal to be able to get the most out of the language.

1. RubyGems

In general, RubyGems provides a standard way to publish, distribute, and install Ruby libraries. It allows library developers to package their products so that installation becomes a one-line process. The resulting packages are simply called "gems." Likewise, RubyGems makes it easy for developers to get up and running with a whole swath of libraries quickly.

Like many packaging and installation systems for other languages (and even operating systems), RubyGems will detect dependencies and install them before installing the desired library, making it a no-brainer process to get a certain library running.

RubyGems currently isn't a standard part of the Ruby installation, but it likely will be in the future. For now, you have to download it separately, but the process is extremely simple. You need only open an archive and run a single Ruby file inside.

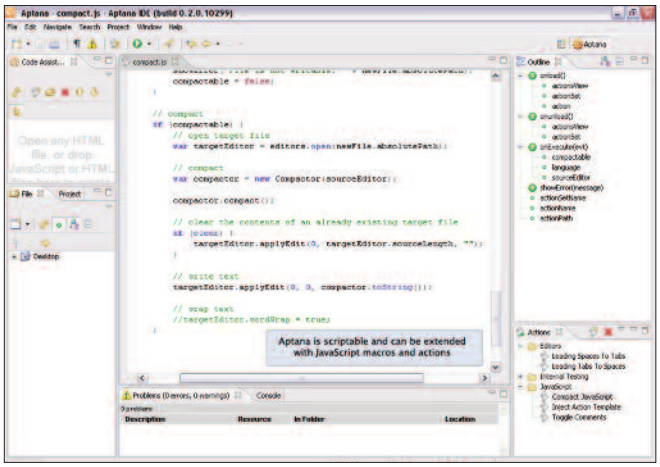
Beyond basic installation of most third-party Ruby libraries, RubyGems also makes managing the libraries installed on your computer simple. It provides a basic command line interface for uninstalling and upgrading libraries. You can even use this interface to install multiple versions of the same library on a single machine, enabling you then to address these separately (specifically by version, if necessary) by applications. This makes RubyGems even more powerful than, say, the popular CPAN system for Perl.

The primary reference and documentation site for RubyGems is rubygems.org.

2. A Good IDE or Text Editor

As with developing in other programming languages, Ruby developers rely on a myriad of different IDEs and text editors for development work. Because of the different platforms and preferences of each developer, it's

Figure 1



impossible to recommend a single option, so this article quickly covers a few alternatives.

RADRails

RADRails was one of the first serious Ruby-specific IDEs. Despite the name, RADRails is not only for Rails applications. It, in fact, is generally useful for developing Ruby applications (see Figure 1). Based upon the popular Eclipse IDE, RADRails is cross-platform (Windows, Linux, and OS X) and open source. Although other IDEs have now become popular, RADRails is still a good Ruby-specific choice.

jEdit

Like RADRails, jEdit is an open source, cross-platform IDE. Unlike RADRails, it isn't Ruby-specific at all. It is a general programmer's text editor. What earns jEdit a

Figure 3

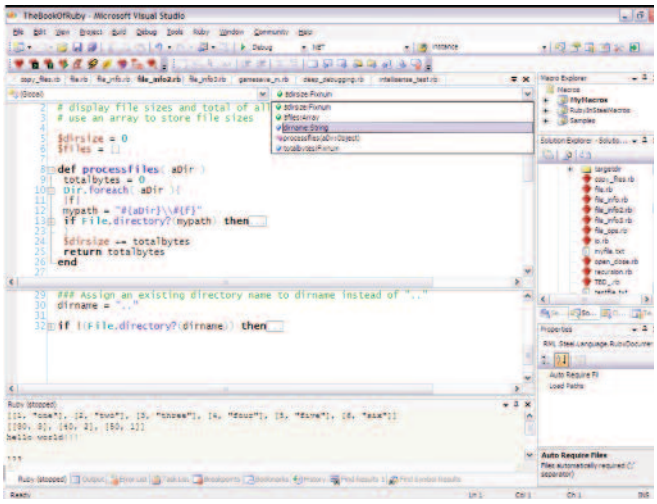


Figure 2

spot on this list is its "Ruby Editor Plugin," a plugin that adds a number of Ruby- (and Rails-) specific abilities to the editor, including syntax and error highlighting, integrated documentation, and auto-indentation (see Figure 2).

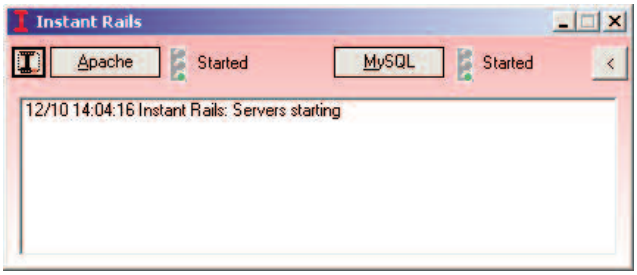
Ruby In Steel

Ruby In Steel is a professional-grade Ruby IDE for Microsoft Visual Studio (MSVS) 2005. It features not only code completion, but also full Microsoft-style IntelliSense features on Ruby code (see Figure 3). While it's not cheap ($199), a free limited-feature edition and a free thirty-day trial make Ruby In Steel appealing to new Ruby developers who particularly appreciate the MSVS IDE.

TextMate

TextMate is an editor available only on Mac OS X. Its use by most of the core team of Rails developers has led to its strong adoption among OS X-based Ruby developers. Like jEdit, TextMate is a general programmer's text editor with a significant number of available Ruby-specific extensions. Depending on the current exchange rate, TextMate costs approximately $50US. TextMate's developer, Allan Odgaard, has been helping another developer produce a Windows clone called E

Figure 4



(a.k.a. E-TextEditor).

3. Instant Rails
Most Ruby developers who have Ruby on Rails Web application framework installed went through the lengthy process of installing RubyGems, installing the Rails gems, and then setting up their environments. It doesn't need to be such a complex procedure, however. Two tools enable you to install a Rails application on a new computer quickly.

For Windows users, a single application called Instant Rails enables them to install and run an Apache Web server, Ruby, the MySQL database engine, and the Rails framework all at once (see Figure 4).

Instant Rails gets a Rails application up and running with only a couple of clicks. This can be ideal if you need to deploy a Rails application to a client or on a remote machine where installing Ruby and Rails is not appropriate.

There are plans to port Instant Rails to Linux, BSD, and Mac OS X in the future, but currently Mac OS X users have an alternative called Locomotive. Like Instant Rails, Locomotive provides an all-in-one Rails deployment system within a single application.

4. Mongrel – A HTTP Server Library
Mongrel is an HTTP server tool and library for Ruby. On the surface, it doesn't sound particularly exciting, but its benefits are compelling. Ruby already comes with a HTTP server library known as WEBrick, but it's extremely slow. Mongrel's speed and reliability are head and shoulders above WEBrick and other alternatives, so installing it allows your Rails applications to run much faster. In fact, Mongrel is now used in the majority of Rails deployments, so it's a useful tool to learn. Additionally, you can use Mongrel directly from your Ruby code to develop your own HTTP server programs.

Installing Mongrel takes only a minute with RubyGems (using a mere gem install mongrel command). It has separate builds for Windows- and UNIX-related platforms due to the need to compile some external C code.

Of the five tools this article covers, Mongrel is the only library. As such, it serves as a great example of how to package, market, and document a library. Mongrel's popularity rests not just on its performance, but also on the way creator Zed Shaw has engaged the community and thoroughly documented the library. If you view Mongrel's source code, you'll find almost as many comments as lines of code.

5. An RSS Feed Reader for Community Interaction
One of the interesting things about Ruby is its community. As a language that gained widespread popularity in the West in only the past few years, Ruby developers have taken advantage of new technologies like blogs to build the community and share their knowledge.
Most of the best Ruby developers have blogs and reading them can extend the knowledge of a novice Ruby developer substantially. Indeed, the strong, evangelical blogging community is one of the main reasons Rails has taken off in the past couple of years. So having a quick and easy way to keep up with several Ruby blogs is key. As nearly all blogs publish an RSS feed, an RSS application or access to an online RSS reader provides a lot of value for any Ruby developer.

The most popular online RSS reader is Google Reader, but client-side readers exist for all platforms. On Windows, a good, free RSS reader is SharpReader. On Mac OS X, NewsFire is a good one. A good place to find Ruby blogs worth reading regularly is Ruby Inside. Visit the site and subscribe to the blogs in the "Choice Blogs" sidebar.

Keeping up to date with Ruby blogs means you'll not only come across the best tutorials and Ruby posts as they're written, but you'll also begin to connect with the community and get a feel for how to extract value from it.

10 Minutes to Your First Ruby Application

By James Britt

This tutorial assumes that you already have a current version of Ruby installed, and you have a code editor handy. You don't need a fancy IDE to code in Ruby; Vim, Emacs, and TextMate are great choices. NetBeans and Eclipse work fine as well.

So you've discovered the grace and power of Ruby and you're ready to explore the subtle but important ideas behind its elegance. Follow this tutorial to create a small, useful Ruby application. As Ruby is primarily an object-oriented language with classes and objects, you can jump right in and create a class to encapsulate behavior. The instructions begin with a simple version of the application, and then expand it. Along the way, you will learn what makes Ruby tick.

Jupiterimages

The example application will serve two purposes:

1. Demonstrate some features of Ruby.
2. Do something useful in the process.

A word on the title: Were you to write this code yourself, assuming some moderate Ruby knowledge, it probably wouldn't take more than 10 minutes. Once you learn how Ruby works and understand what sort of code it enables, you'll find that you can whip up useful utilities in short order. Of course, a walkthrough of such code will take a bit more than 10 minutes if you're new to the language.

Target Problem: Simplifying File Launching

Ruby is primarily a text-based, command-line-oriented language. Some GUI libraries are available, as well as multiple Web application frameworks, but exploring GUI development with Ruby is beyond the scope this article. The goal here is to write something that works from the command line.

The example task is simplifying file launching. Given a text file (maybe a Ruby source code file), suppose you want to create a way to launch it in some associated application from the command line. And you want to launch it without having to keep track of file types and application associations. Yes, Windows already does this, but your application will have additional features that go beyond this simple behavior.

Version 0: The Launcher Code

First, create a sparse Ruby file. Ruby files end with .rb and have the pivotal line that defines the path to your Ruby interpreter up top. Call the file launcher.rb:

```
#!/usr/local/bin/ruby

# Example application to demonstrate some basic Ruby features
# This code loads a given file into an associated application

 class Launcher
 end
```

Notice you can use a pound sign (#) to start a line-level comment. Everything to the right of the # is hidden from the interpreter. Ruby has a means for commenting multiple lines of code, too. Class names begin with a capital letter; classes are constants, and all Ruby constants start with a capital letter. (For a more complete overview of Ruby syntax, please see "Ruby—A Diamond of a Programming Language?", Part 1 and Part 2.)

While this code seemingly does nothing, it is executable. If you're playing along at home, you should see that your copy of the code executes. A simple way to run a Ruby script is to simply call the ruby interpreter and pass the name of the file, like this (see Sidebar 1. Instructions for Executing launcher.rb in Unix and Windows):

```
$ ruby launcher.rb
```

When you run the file, you should see nothing—unless there's an error of some sort in the code. So, nothing is good. It doesn't mean nothing is happening; when the ruby interpreter parses your file, it encounters your class definition and makes it available for creating objects. The following code adds the class definition to your code:

```
#!/usr/local/bin/ruby

# Example application to demonstrate some basic Ruby features
# This code loads a given file into an associated application

 class Launcher
```

Instructions for Executing launcher.rb in Unix and Windows

On Unix systems you can set the file as executable and call it directly:

```
$ chmod u+x launcher.rb
$ ./launcher.rb
```

Windows users have a leg up here if they used the so-called One-Click Ruby Installer. It takes a few more clicks than one, but in the end it sets up an association for .rb files. So a Windows user should be able to execute the app straight off as follows:

```
C:\some\dir> launcher.rb
```

However, Windows users should also know that though they can launch a Ruby file by double clicking on it from the Windows file explorer, the results are fleeting: code will execute in a command shell, which will remain visible only so long as the application is running. For the sake of this demonstration, it's best to run the file from a command shell.

```
    end

launcher = Launcher.new
```

The code first creates a variable (launcher) that is assigned a reference to a new instance of the class Launcher. You do not have to declare the type of the variable. Ruby uses strong, dynamic typing, and variables can hold references to objects of any type. Pretty much everything in Ruby is an object, including strings, numbers, and regular expressions. Each of these has a formal creation method (e.g., String.new), but Ruby tries to make it easy and fluid to work with the common cases.

Secondly, Ruby creates the object instance by invoking new on your Launcher class. New is a class method; it's analogous to constructor methods in Java. Of course, an empty object won't get you far, so you must add some behavior.

Adding Behavior

The essence of your application takes a given file name and passes it to an associated application for processing of some sort. The launcher code will need to know how to do this mapping, so when you create an instance of a Launcher class, you must pass in some sort of mapping. You've seen that you can use the class method new to create an instance of a class. To create an instance that starts life with some set of data, you can pass in arguments to new. To handle this, you of course will have to add some code to Launcher:

```
def initialize( app_map )
  @app_map =  app_map
end
```

You define methods in Ruby using the def keyword, followed by the method name, and then the augment list, if any. The argument list is in parentheses for clarity, though Ruby will allow you to omit them when the meaning of the code is unambiguous (see Sidebar 2. Why You Add initialize Method When Passing Arguments to new Method).

It's worth noting then that Ruby objects begin life with assorted built-in behavior. You can use these as is, or opt to override them.

Instance Variables

Your initialize method takes one argument, app_map. Again, as with the earlier variable, you do not give the types of method arguments. You just say that the method takes one argument (app_map), and in the body of the method this argument gets assigned to the variable @app_map. The @ symbol indicates that the variable is an instance variable (i.e., it is available to all the code in this object). You create this instance variable when you create your object, and it will be available to any other methods you add to your code.

Why You Add initialize Method When Passing Arguments to 'new' Method

You're probably thinking, why am I adding a method named "initialize" when I want to pass arguments to a method named "new"? The reason has to do with how Ruby creates objects from classes. All classes (such as Launcher) inherit from the class Object, and part of the deal is that the objects they create have a default initialize method. When the class method new is called, it first allocates some resources for the desired object, and then invokes the fresh object's initialize method. If you wish to provide creation parameters via new, you must define your own initialize method to handle the arguments in the newly created instance.

To have your application execute a given file using the associated application, drop some more code into it:

```
class Launcher

  def initialize( app_map )
    @app_map =  app_map
  end

    # Execute the given file using the associate app
    def run( file_name )
        application = select_app( file_name )
    system( "#{application}  #{file_name}" )
    end

    # Given a file, look up the matching application
    def select_app( file_name )
        ftype = file_type( file_name )
        @app_map[  ftype ]
    end
```

A Few Words About Objects, Types, and Behavior

Ruby follows a message-passing model of object-oriented programming. When you see code like foo.bar, it means that the message "bar" is being passed to the object referenced by foo. Most of the time, that object will have a method bar, and when you see such code you may be tempted to think of it as calling foo's bar method. However, while that is convenient (and common), it is important to know what's happening under the hood.

When an object receives a message, it first looks for a corresponding method. The search will work its way up the inheritance hierarchy, starting with the object's own class, until it reaches the Object class. If no match is found, then the method method_missing is called. As you may have guessed, there's a default implementation of method_missing, and it does nothing more than raise an exception. But just as you were able to override the default definition of initialize, you also can alter method_missing. You are free to redefine it so your object might apply some smarts to handling arbitrary message requests, making it appear that the object implements many more methods than it actually does.

This flexibility is at the core of one of the most appealing aspects of Ruby, but it also points to an important aspect that may trouble some people. You've seen that you do not declare data types when creating variables or defining method argument lists. If you want to check data types, you can. Code can ask for an object's type, and act accordingly. For example, you may want to write a method that accepts either a file name (e.g., a String object) or a file handle (e.g., a File object). But Ruby code rarely checks an object's type simply for defensive measures, refusing to continue unless given an object that asserts itself to be a certain type. Because classes and objects are mutable at run-time, the notion of type in Ruby is essentially defined as the behavior of an object at any given time. Type is defined by which methods an object responds to, not which class it comes from. As Rubyist Logan Capaldo once said, "In Ruby, no one cares who your parents were. All they care about is if you know what you are talking about."

The general term for this is duck typing, from the phrase, "If it walks like a duck and quacks like a duck, then it's a duck."

```
    # Return the part of the file name string after the last '.'
    def file_type( file_name )
        File.extname( file_name ).gsub( /^\./, '' ).downcase
    end

end
```

The method run takes a file name as its argument, passes it to select_app to find out which application to execute, and then uses Ruby's system method to invoke that application, passing the file name. The system method simply kicks the given command into a sub-shell. While select_app takes the file name, calls file_type to get a 'normalized' file extension, and then uses that as a key into @app_map to see which application to run.

Finally, file_type takes the file name and uses a class method on Ruby's File class to get the extension. The string returned by extname includes the period (.) that precedes the file extension. You don't need that, so the code uses gsub (or global substitute) to strip it; it then converts what remains to all lowercase letters with downcase.

For compactness, all these method calls are chained together. The string returned from File.extname is the receiver of the gsub request; the string returned from gsub then becomes the receiver of the call to downcase.

The example code so far has used objects that you expect to be Strings and Hashes, but what you really care about is that these objects will respond to particular messages in an appropriate way. (Before delving into how to call your shiny new object, see Sidebar 3. A Few Words About Objects, Types, and Behavior.) For such a small application, the subtlety and power of an object system based on messages and run-time behavior may not be critical, but it is important to understand this as you go on to write larger Ruby applications.

The Smarts Behind Launching Logic

Where you were mapping a file extension to a particular application name, you now want to add associations with Ruby code. Specifically, you want Ruby classes custom-coded for each type of file you want to process.

First, create a new Ruby source file named launcherx.rb (the x is for extended) in the directory as launcher.rb:

```
#!/usr/local/bin/ruby
# File launcherx.rb
require 'launcher'
class Launcher
def handler( file )
get_handler(file) || build_handler(file)
end
def build_handler file
handler = Class.new
application = select_app(file)
eval "def handler.run
system( '#{application}  #{file}' )
end"
handler
end
def get_handler(file)
```

continued

Rounding Out Version 0

Finish up this first version by putting it to use. You can add the following code to the end of the file to create an instance of Launcher and use it to run an application:

```
def help
  print "
  You must pass in the path to the file to launch.

  Usage: #{ __FILE__} target_file
"
end

if ARGV.empty?
  help
  exit
else
  app_map = {
      'html' => 'firefox',
      'rb' => 'gvim',
      'jpg' => 'gimp'
  }

  l = Launcher.new( app_map )
  target = ARGV.join( ' ' )
  l.run( target )
end
```

The Smarts Behind Launching Logic

```
begin
here = File.expand_path( File.dirname(__FILE__ ))
ftype = file_type(file)
require "#{ here} /handlers/#{ ftype } "
Object.const_get( ftype.capitalize ).new
rescue Exception
nil
end
end
# Execute the given file using he associate app
def run( file, args = nil )
handler(file).run(file, args)
end
end
```

The first thing to note is that the code is calling require to load the existing definition of Launcher. Yet your new code also defines a class named Launcher. What gives? When Ruby encounters a class definition that uses the name of an existing class, it updates the existing class with the new class. The methods defined in your first version of Launcher are still there; new methods defined in the additional code get added. And, as in the case of run, when new code uses the same name as existing code, the new code replaces the old. The upshot of this is that you do not have to duplicate the code from your first version; you need only add to or modify it.

continued

The method help will render instructions if needed. ARGV is the argument vector; it is a built-in Ruby object that holds all the parameters passed to your program. If it's empty, then your program has nothing to work with, so it displays the help and exits. Otherwise, it creates a hash object and assigns it to the variable app_map.

The { ... } notation is Ruby's literal syntax for creating a Hash object. You could have used Hash.new, but it's verbose. Using the literal notation, you map hash keys to values using =>. The hash is used to populate your Launcher instance, while the command-line arguments are collected into a single string stored in the variable target, which is passed into run.

Before trying this code, you need to change the application values used in app_map so that they refer to the proper executable. Assuming you have "rb" mapped to a text editor, you can try the code like this:

```
$ ruby launcher.rb launcher.rb
```

This should open your source code in your editor.

Bulking Up to Version 1 with Dynamic Loading

So far, so good with Version 0, but you can do better. Rather than having a simple, direct mapping of file types to the application, you could map file types to execution handlers. That is, you can define code for your file types that can then decide which application to run, and with which arguments, depending on additional command-line arguments.

For example, if you are doing web development and have created an HTML file, you most often want to view it in a browser. So your application as it is works OK. But sometimes you want to view it using a particular browser. Right now, Launcher only allows a single application association. What you may want is the ability to launch

The Smarts Behind Launching Logic

Not so incidentally, this also works on core Ruby classes. For example, you can add to or alter the behavior of String by defining a String class with your own methods. If your code is frequently altering strings (perhaps to replace special characters), you can make your code clearer with something like this:

```
class String
def amp_escape
self.gsub( '&', '&' )
end
end
```

Which then enables your code to do this:

```
"This & that".amp_escape
```

Your new application file now needs to handle the new behavior. The run method changes because this new version will be invoking Ruby code instead of directly calling a shell command, and you want the option of passing in additional arguments. Therefore, this version expects a file name and an optional array of arguments. It's optional because in the methods argument list you're giving it a default value of nil. Arguments pre-assigned this way must always come last in the argument list.

Whereas your first version simply used the file extension to pull an application name from a hash, this code uses

continued

myfile.html in the Opera web browser:

```
$ ./launcher myfile.html opera
```

Or you my want to perform some syntax checking on the HTML:

```
$ ./launcher myfile.html syntax
```

In other words, you want to add some smarts (see Sidebar 4. The Smarts Behind Launching Logic).

Dynamic Loading

To add those smarts, you will change your program so that you can associate file types with Ruby code rather than associating a particular application. That Ruby code will handle the launching logic, allowing you to decide just how clever to be when launching an application for a given file type (see Sidebar 5. Dynamic Class Loading with Defined Custom Ruby Classes).

Before doing this, make one small change. Having all your code in one place is handy, but it's not a good practice for anything but the smallest apps. For the sake of better organization, split out the general class code from the code that interacts with the user. Do this by creating a file, go.rb, and moving all but the actual Launcher code into that file (i.e, that last chunk of code you just added):

```
#!/usr/local/bin/ruby

require 'launcher'
```

The Smarts Behind Launching Logic

handler to create a corresponding Ruby class to handle the given file name. The handler method is short; it first calls get_handler to see if it can locate a matching handler class. The || method is Ruby's logical OR. Should get_handler return false (or nil, which Ruby treats as false), then the code to the right of || is invoked. If there is no defined handler class, then the code makes one.

Recall that your new version of run will expect get_handler to return an object that responds to the run message. The build_handler method therefore needs to define a class with this behavior. There are a variety of ways you could do this. Here, you're going to first create a generic instance of the class Class and then dynamically add a run method that knows how to handle the particular file type in question.

Your new Launcher class retained the application map code from the original. This mapping serves as a fallback for handling files in the absence of any special Ruby code, meaning that your new version still does what the first version did. Your code can still call select_app to get the default application. The trick now is to get that into a method on your new class.

Perhaps the simplest way to do this is to build a string with the code you might write if you did know which application to invoke. You then have Ruby eval (i.e., evaluate) this string, making it part of the current process. (Note: capricious use of eval on arbitrary strings is not wise. It works well for the sample application and helps demonstrate an interesting feature of Ruby, but use it with care in more serious applications--especially any code that allows input from users.)

Just like that, build_handler can now return an object (albeit sparse) that knows how to do the one thing that matters: respond to a run request.

```
# Script to invoke launcher using command-line args
def help
  print "
  You must pass in the path to the file to launch.

  Usage: #{ __FILE__} target_file
"
end

unless ARGV.size > 0
  help
  exit
else
  app_map = {
      'html' => 'firefox',
      'txt' => 'gvim',
      'jpg' => 'gimp'
  }

  l = Launcher.new( app_map )
  target = ARGV.join( ' ' )
  l.run( target )
end
```

Note the extra line of code near the top:

```
require 'launcher'
```

You need this line to make your Launcher available to the current script. The require method looks for a file matching the given string. The file extension is omitted, so Ruby first will assume you want a .rb file but also will look for a compiled library (e.g., .so) if it doesn't find a Ruby file. (Ruby searches a pre-defined load-path, which includes

Dynamic Class Loading

The real fun is in defining custom Ruby classes that have more interesting implementations of run. First, assume all these classes will live in files named after the file extension they handle. For example, a handler class designed to process HTML files will go into a file named html.rb. Also, all such files will go into a relative subdirectory named handlers. Asserting these two conventions allows the get_handler code to know just what to look for and where to look for it, bypassing a need for lengthy configuration settings.

When get_handler is called, it:

1. Uses some built-in File methods to figure out the current file-path location (__FILE__ is a special Ruby variable that refers to the actual file containing the current code).
2. Appends your pre-defined handlers directory to the current path.
3. Uses the file extension of the target file name to derive the name of the file holding the handler class code.

All of this is passed to require with the expectation that such a file exists and that it will be loaded. If all goes well, Ruby will load and parse this file, making the desired class available to your code.

continued

the current directory, so if you keep launcher.rb in the same place as go.rb, you're good. If you move it, you have to be more explicit about were Ruby can find it.)

Writing a Handler Class

Now that you have a simple framework for routing file names to Ruby code, create a handler class for HTML files. The class needs to implement a run method that accepts at least one argument for the target file name, and an optional array of additional parameters. The class name must be Html in a file named html.rb, and placed in a handlers subdirectory:

```
class Html

  DEFAULT_BROWSER = 'firefox'

  def run file, args
    if args.empty?
      system( "#{ DEFAULT_BROWSER}  #{ file}" )
    else
      dispatch_on_parameters file, args
    end
  end

  def dispatch_on_parameters file, args
    cmd = args.shift
    send( "do_#{ cmd}", file, args )
  end

  def do_opera file, args=nil
    system( "opera #{ file}   #{ args}" )
  end

  def do_konq file, args=nil
```

Dynamic Class Loading

Now, without knowing the name of the class you want to instantiate in advance, you again need to do a bit of dynamic invocation. You could again use eval, but you can also reach into Ruby's list of constants (remember, classes are Ruby constants) and call new. Again, if all has gone well, Object.const_get will return the class desired, and new will then return an instance.

Should something go wrong (perhaps there is no such file to load, or the code in the file is malformed) and Ruby raises an exception, the code uses rescue to handle things. You could use rescue with more specific exceptions for more targeted error handling, but for the purposes of this example, you simply want to trap all exceptions and quietly return nil.

You may have noticed that get_handler does not explicitly specify which value to return. Indeed, none of your methods have done so. In Ruby, the return value of a method (with some exceptions) is the value of the last expression executed. get_handler has one top-level expression: begin/rescue/end. Its value will be either the value of the last expression in the begin/rescue section or the value created in rescue/end. Ruby does define return, which exits a method returning the provided value, but method flow control is sufficient to define an unambiguous return in most cases.

```
    system( "konqueror #{file}   #{args}" )
  end
end
```

The code defines a constant for a default browser. In the absence of any extra arguments, then, you can have the target file launched in Firefox. (Note that you may have to change this so that it defines an executable command. On my Ubuntu machine I can run firefox with no explicit path and have a browser come up. On Windows, for example, the full path to the executable may be needed.)

If there are additional arguments, run calls out to dispatch_on_parameters, which extracts the first item from the args array and uses it to dynamically construct a message string. The send method is built in to all Ruby objects. It allows you to explicitly send a message to an object. When used by itself (as you are doing here), the receiver object is assumed to be the current object. So the code is sending a message to itself.

You prepend do_ to the actual argument value as a safeguard against method name collision. (For example, if the first argument were exit, you probably would not want to invoke Ruby's exit method. You'd call do_exit, which would then decide what the correct behavior should be).

This handler code has some fairly trivial examples of possible parameter handling. As is, you can launch a target HTML file in either some default browser or specify a particular browser:

```
$ ./go index.html opera
$ ./go index.html konq
```

A Little Overtime for Coolness

You've received an educational and practical example, but can you push things a little further? Of course you can. Mind you, this will take you past the 10-minute mark, but it should be worth it.

The standard Ruby distribution includes a wealth of libraries for all sorts of tasks. One of the most interesting is REXML, an XML parser written in pure Ruby. Developer Sean Russell wrote REXML to allow the manipulation of XML using a Ruby-style API rather than the usual W3C DOM API. Before too long, Sean's work became part of the Ruby standard library.

For the sake of simplicity, your HTML files in this example must use XHTML because REXML handles only XML. (There are very good Ruby tools for processing near-arbitrary HTML, one being Hpricot. However, they require installing additional libraries, the explanation of which is beyond the scope of this article.) Trusting that you are working with well-formed XHTML source, you can have your HTML handler do some file analysis. Add this code to the end of your Html class and you'll be able to run some simple reports on your XHTML:

```
  def do_report( file, args=nil )
    require 'rexml/document'
    begin
      dom = REXML::Document.new( IO.read( file ) )
      if args.empty?
        puts basic_xhtml_report( dom )
      else
        puts report_on( dom, args.first )
      end
    rescue Exception
```

```
      warn "There was a problem reading '#{file}':\n#{$!}"
    end
  end

  def report_on dom, element
    els =    dom.root.elements.to_a( "//#{element}" )
    "The document has #{els.size}  '#{element}' elements"
  end

  def basic_xhtml_report( dom )
    report = []
    css = dom.root.elements.to_a( '//link[@rel="stylesheet"]' )
    unless css.empty?
      report << "The file references #{css.size}  stylesheets"
      css.each do |el|
        file_name = el.attributes['href']
        file_name.gsub!( /^\//, '')
        unless File.exist?(file_name)
          report << "*** Cannot find stylesheet file '#{file_name}'"
        end
      end
    end

    js = dom.root.elements.to_a( '//script' )
    unless js.empty?
      report << "The file references #{js.size}  JavaScript files"
      js.each do |el|
        file_name = el.attributes['src']
        file_name.gsub!( /^\//, '')
        unless File.exist?(file_name)
          report << "*** Cannot find JavaScript file '#{file_name}'"
        end
      end
    end

    report.join( "\n" )
  end
```

There's a lot going on here, but key method is do_report. The code creates a REXML Document object and assigns it to dom. If there are no extra arguments, you get back a basic report. Otherwise, the code does some cursory examination of a particular element.

The report_on method takes a document argument and an element name, and uses REXML's XPath features to find out how often that element is used. Although it's rudimentary, it certainly can serve as a demonstration and starting point for you to keep hacking.

The basic_xhtml_report method is similar, but focuses on a particular set of elements. It uses REXML to find all the CSS and JavaScript references, and then uses the File class to check that the referenced files exist. Again, not deep, but adding additional logic makes for a nice project.

Clean, Expressive Code with Minimal Scaffolding

You now should have a better understanding of some of the features that make Ruby so special, namely:

- Ruby is primarily an object-oriented language, where a key concept is objects responding to messages.
- Ruby uses strong, dynamic typing, where the notion of "type" is based on what an object can do more than on a particular class name or inheritance hierarchy. An object's behavior is not confined to a literal mapping of messages to methods, and behavior may be constructed dynamically at run time.
- Ruby classes are open; you are free to alter their behavior for what you deem appropriate for a given application.

This combination of open classes and dynamic behavior enables you to write clean, expressive code with a minimum of boilerplate scaffolding. Ruby gets out of the way and lets you get coding.