MR. NEIGHBORLY'S

HUMBLE LITTLE RUBY BOOK

MR. NEIGHBORLY'S

HUMBLE LITTLE RUBY BOOK

Jeremy McAnally

For my wife, friends, and family,
thank you for the support and food.

Mostly the food.

TABLE OF CONTENTS

A Note

In the following book, I will be using `Ruby 1.8.5` to test all the Ruby code. Each example can be copied and pasted directly into `irb/fxri` and it should work famously. I have done so with each one to make sure they run.

Each time I am showing output from `irb`, you will see a → character followed by the output. Any method or variable name or code/system related text is typeset in `this font` for easy discernment from other text.

Any time I have found it necessary to differentiate a class object from an instance object, I have erred on the side of standard notation and went with the form of `Class#Object` (even though it's very ugly and is not what the rest of the civilized world uses).

"I gotta go. There's a dude next to me and he's watching me type, which is sort of starting to creep me out. Yes dude next to me, I mean you."

Night-hen-gayle (bash.org)

0

What'chu talkin' 'bout, Mister?

Yes there is a Chapter 0. There is a little bit of introductory stuff we need to talk about before we set you loose on Ruby. You wouldn't want to get psyched about a new gadget, get it home, and then figure out you need batteries, a grapefruit, and the ability to speak three languages to even open the box would you? You would? Well then answer me this: How would Ruby react to large, elastic monsters taking over an estuary? You don't know the answer!? Well, plainly we need to take a good, hard look at a few things before we turn you loose.

WHAT IS RUBY ANYHOW?

Ruby is an open-source, multi-paradigm, interpreted programming language (a bit of a mouthful I know! I'm going to explain it, I promise!). Ruby was created by Yukihiro "Matz" Matsumoto, a very fine Japanese gentleman who currently resides in Shimane Prefecture, Japan; Matz's work on the language was started on February 24, 1993 (commonly considered the birthday of the language; I hear that over in Japan they roll out a two-story cake and sing) and released to the public in 1995. Ruby is often hailed as one the most expressive and concise languages available to developers today. In that spirit of expressiveness, let's look at exactly what it all means. Let us now eviscerate these verbal furbelows with conviction!

Open Source The official implementation of the language is free software distributed under the GPL and the Ruby open source license. If you're unaware of what "open source" means, then look at it this way: Matz programmed the

entire Ruby interpreter and library, then gave away the code he used to do it. Since the source code is available, people can now take it and improve it. Many people take the code, improve it, and Matz (and his crack team of maintainers) integrate their changes back into the main source code. The benefit of open source is chiefly that you get a lot more minds working on a project than a proprietary project (and typically for free to boot!).

Multi-Paradigm Like to write code in a functional style a la Haskell or Scheme? That's cool; Ruby does that. Really dig object orientation like Java or Smalltalk? No problem; Ruby handles that, too. Prefer to use a procedural (a.k.a. imperative) style like Forth or Ada? Ruby can "get its procedure on" as good as any other language! Don't know what any of those mean but just really want to program? Excellent! Ruby is a multi-paradigm language because it doesn't constrain to a single programming mindset; you can use any of the aforementioned programming paradigms with no problems in Ruby. You can pick the one you prefer (or the easiest for you to learn) and go with it: Ruby doesn't mind. Unlike some other languages, it doesn't get jealous and give you "errors" if you break it off with objects and decide to go steady with closures instead.

Interpreted If you've used something like Assembly, Pascal, Fortran, or C/C++, you've used a compiled language. "Compiled" means that you've had to run your code through a little compiler and it spits out some sort of native code that will run without any sort of interpretation by the computer other than by the operating system itself. This can become time consuming as your project grows larger and larger, and sometimes can even be a severe hindrance to productivity. Oh, but there is another way! Ruby is an interpreted language, which means that there is an interpreter that reads your code and then emits native code to the operating system. Maybe this diagram will make more sense...

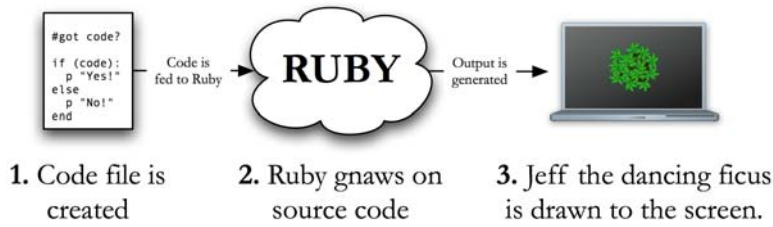


Figure 1: An overview of how Ruby handles programs.

There is a bit more to it than that (e.g. you have to coax Jeff out of his box), but that's the general concept. Code goes in, Ruby plays around with it, program comes out. When running Ruby code, you have a few options. The first option you have is to create a file with your source code in it, and then tell Ruby to execute that file by giving it as a command line option to the `ruby` command. Your next option is that you can use your editor's Ruby runner (if it has one) by using their shortcut; for example, SciTE has this feature that you can use by pressing F5. The last option is to use an interactive shell like `irb` or `fxri`; these shells give you a "Ruby prompt" at which you can type in lines of Ruby code and have them immediately executed. I used `irb`

extensively in writing this book because it allows speedier feedback than running from a file. These prompts are possible because Ruby is a dynamic language ran in an interpreter.

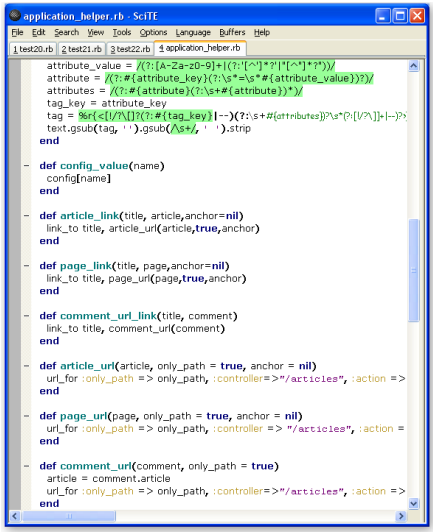
The interpreted nature of Ruby allows it to have many of the features that make it so great. Compiled programs are not nearly as dynamic as interpreted ones because they don't (usually) allow for runtime change to code or the application itself. Because interpreted programs are simply, well, interpreted from their original source (or a slightly compiled bytecode), they can allow for more far-reaching runtime interaction. We'll discuss all of this a lot more later as these features reveal themselves; now we need to make sure you actually have Ruby. Otherwise, these features will be like chasing the wind when it's running faster than you: meaningless!

INSTALLING RUBY

Sometimes installing a programming environment can be a pain; if you've ever tried to install a GNU compiler on Windows you know what I mean. Fortunately, Ruby is relatively easy to install on most platforms.

Windows

Installing on Windows is a snap; simply navigate over to the web site at http://www.ruby-lang.org and click on the "Ruby" link under the "Download" sidebar. Then click on the "Install Ruby under Windows" link. That page has a link to the installer that you need along with instructions on how to install it (basically, download, run, and you're done).



To edit Ruby files, you simply need a text editor. This could be something as simple as Notepad or as fancy as UltraEdit. Ruby comes with a fine editor named SciTE which will properly highlight, open, and save Ruby files for you; it also has the nice feature of running your programs for you so you don't have to poke around the command line to get them going. There are other Ruby specific development environments (e.g. FreeRIDE, Arachno, Mondrian, etc.), but these environments are not necessary to do development (i.e. I don't use them).

If you decide to simply use Notepad or something that doesn't have a feature that allows you to run your application from

within it, then you need to find your programs using the command line and issue the Ruby command to run them. For example:

```
ruby mycodefile.rb
```

I suggest simply using SciTE to avoid this, but sometimes this is a necessary evil (especially if you're already very comfortable with another editor).

Mac OS X

If you're rolling on Jaguar (10.2) or later, then you should already have some variant of Ruby on there. To find out which version you have (and to make sure it's compatible with this book which is based on the 1.8 versions), type the following in Terminal:

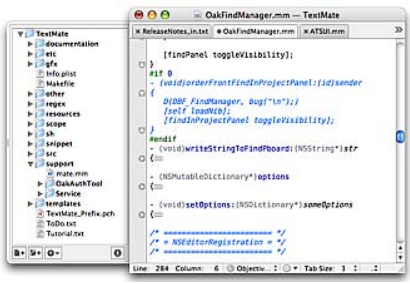
```
ruby -v
```

That should give you a short message telling you which version you have installed. If you have a 1.8 version installed, great! If not, let's install one.

The easiest way to install Ruby is going to be using DarwinPorts (http://www.darwinports.org/). Go to their website, then to their download page and download the proper .dmg file for your version (e.g., for Tiger/10.4 you would download something like DarwinPorts-1.2-**10.4**.dmg). Install that as you normally would. Then open Terminal and enter the following:

```
sudo port sync
sudo port install ruby
```

It will download some files, compile some things, calculate the airspeed velocity of an unladen swallow, and then finally you'll have a working Ruby distribution! Just run the above `ruby -v` command to make sure everything is in order. If that doesn't work, then go to the Darwin ports website and check their support and mailing list.



Editing Ruby files on Mac OSX can be done using something like Text Editor if you like to keep things simple. If you require a little more out of your environment, you can splurge on something like TextMate (my Mac IDE of choice). If you're a hardcore, UNIX-or-die, console-only kind of person, then vi or emacs works perfectly fine, and a lot of work has been done with these editors to actually make them pretty usable Ruby development environments if you are so inclined to use them.

Linux

To assess whether or not you have Ruby already installed (and you very well may), type `ruby -v` on the command line. If Linux can't find Ruby, then type `which ruby` on the command line. If you are again confronted with the horrible fact that Ruby is not on your system, prepare to install it. Fortunately, Ruby has widespread support on Linux, and depending on your variation of Linux/your environment, you will have to do one of the following.

Install from Administrator If you are not the administrator of your machine, you may have to throw yourself at the mercy of your systems administrator. Beg him or her to install it for you. Offer them pastries. Tell them that you will urinate on their desk if they don't install it. Whatever it takes, get Ruby installed!

Install from Package To install from a package, you will need to consult your distribution's documentation. Each distribution handles these sorts of things differently, but if your distribution simply doesn't have a package you have options. First, check unofficial repositories such as Dag's, Mind's, Shadoi's, or Debian-Unofficial (or the tons of others that Google will turn up if you ask it nicely). If you don't have any luck there...

Install from Source Installing from source is some people's first instinct. You do get a smidge better performance, for sure, but I'm hasty and like to get things done as quickly as possible. I'd rather just drop a package in and go. If you're a masochist or simply can't find a package, you can install from source. First, go to http://www.ruby-lang.org and download the source archive. Then, extract it and enter the source directory:

```
tar zxvf ruby-1.8.4.tar.gz
cd ruby-1.8.4
```

Poke around in there a bit; you might want to read the license or README to make sure that there aren't any gotchas for your distribution of Linux. Then, you need to configure, build, and install it:

```
./configure
make
make install
```

You should be good to go at this point. Type `ruby -v` to make sure that it's installed properly.

Editing Ruby files on Linux is as simple as using a plain text editor like gEdit or your favorite console editor, such as vi, emacs, or nano, or one of their X Windows counterparts like xemacs. There are also more robust environments, such as jEdit and Arachno Ruby that you can acquire, but they are not required.

LET'S TRY HER OUT!

Let's give this whole Ruby thing a try. You can either use irb or fxri to get instant feedback or type the source code in a file and use Ruby to execute it. If you want to use the former, either type irb on the command line or find fxri in the program group for Ruby. If you want to simply type it in and execute it, then open your favorite editor and let's get cracking.

```
puts "Hello, world."
```

This is, of course, the prequisite for any programming book. You should've seen "Hello, world." if you're using one of the interactive shells; if you're placing this in a file, save the file as something like hello.rb and then type `ruby hello.rb` to execute it. Now, let's make this a little more interesting.

```
puts "Hello, world.  What is your name?"
myname = gets()
puts "Well, hello there " + myname + "."
```

Save the file again and run this (or type it in your little interpreter); you should see a greeting, be asked for your name, and then greeted by name. If you didn't figure it out, `puts` makes text come up on the console and `gets` gets text from the user. Now that you've got a little Ruby under your belt, you're good to go on Chapter 1.

This Chapter

You learned a little about Ruby and how to install it. You learned...

- the history of Ruby and where it came from.
- the gist of what Ruby is and how it works.
- how to install Ruby.
- a little bit of Ruby.

1

Welcome to Ruby.

This section aims to introduce the syntactic sugar and linguistic misfortunes of Ruby in the quickest manner that will still allow for a full education on the subject. If you rate yourself a Ruby guru, hate language tutorials for one reason or another, or if you stayed at a Holiday Inn Express last night (or thought about it but decided their overpriced accommodations weren't for you), then you may merrily proceed on to the next section.

BASIC CONCEPTS OF RUBY

Ruby is an object-oriented language, but before you skip this section because you think you know what this is all about because you have used C++ or some other unfortunate excuse for an object-oriented language, then please pause and at least read the following sentence. In Ruby, everything you manipulate will be an object. Everything. Even the results of operations on said objects are objects; this approach differs from C++ or Java where primitive types exist or some statements do not return a value.

If you have never delved into object-oriented programming (or programming at all), then that is a different story altogether. When writing Ruby code, or object-oriented code in general, the idea is to create models in your code that render the process you are trying to go through in code. For example, if you were creating a cookbook application, you would probably want to create a list of recipes (my skills of deduction are amazing, I know). To model that in a not-so-object-oriented way, you would most likely use a series of list structures of some sort to hold the various sorts of data with a synchronized way to track the position of each list or some such nonsense. Object-oriented programming simplifies this and allows you to create classes and objects to model the needed components. Using our example, you could create a `Recipe` class with string attributes `name` and `author` and a hash or array attribute of `ingredients`. A class's purpose is to *model* some *thing* in your application; classes create the "prototype" for the nouns in your programs: objects. Class instances, or objects (the terms are interchangeable), then take that prototype and put it into

action. In our example, objects could be created for each recipe in the list that would be instances of the class `Recipe`, which would in turn could hold data and do things related to being a recipe (i.e., hold a list of ingredients, add ingredients to that list, and so on) and enforce constraints that would be enforced on a normal recipe (i.e., only allow numeric values for ingredient amounts, make sure there is a name for the recipe, and so on).

TYPES IN RUBY

Just because everything is an object in Ruby does not mean that everything is generic (in the sense that specialized functionality does not exist) or that there are no "built-in" classes. Ruby provides a number of built-in classes which act as building blocks for the all of the components of your application. These types differ from those in many other languages in that they all originate from the same class originally: the `Object` class. This hierarchy means that there is only one "base" type rather than a number of primitives like there are in languages such as C. What follows is a walk-through of how these types differ and what they can offer you as a developer.

Strings

The first of these types that we will look at are strings, which are simply sequences of bytes that represent a sequence of characters. Strings can be formed a number of ways, but the most common is likely to be using a string literal. A string literal is a constant string that is created by enclosing it in single or double quotes. For example:

```
puts 'Hello, Darling.' → Hello, Darling.
puts 'What\'s  up?'    → What's up?
puts "A\tTab."         → A          tab.
```

Wait a minute! What are those backslashes? Those are escape sequences, a backslash followed by a character to create normally unprintable characters (i.e. in this example I used `\t` to create a tab character but you can also use others to create things like new lines and vertical tabs). I said unprintable because in the other example, I used `\'` to create a single quote character; this character would normally be unprintable because it is contained in a set of single quotes and would effectively close the set of quotes causing an error.

Now, if you noticed, I used single quotes for some of the strings and double quotes for others. There is a difference between the two notations. Single quoted strings are quite silly and have a very limited set of escape sequences they can use (as a matter of fact, only single quote and backslash are allowed) and are typically useless unless performance is a concern for you (and turning double quoted strings to single quoted strings should probably be the last thing you try when improving performance); double quoted strings, on the other hand, offer far more functionality in the way of interpolation. Firstly, they offer far more escape sequences. As noted above, you can use `\n` to create a newline character, `\t` to create a tab character, and so

on; below is a table of all the available escape sequences you can use with double quoted strings (there are quite a few).

ESCAPE SEQUENCES

`\a`	Bell alarm	`\f`	Form feed
`\???`	Octal value	`\n`	New line
`\x??`	Hex value	`\r`	Return
`#{???}`	Value of ???, where ??? is a Ruby expression	`\s`	Space
`\e`	Escape	`\t`	Tab
`\c?` `\C-?`	Control-?	`\v`	Vertical tab
`\M-?`	Meta-?	`\b`	Backspace
`\M-\C-?`	Meta-Control-?		

Looking at that table, you may have noticed that double quoted strings also offer another interesting feature: expression interpolation. As fancy as that sounds, it simply means this: you can insert the value of pieces of Ruby code into strings directly. Remember that everything in Ruby is an object, even the results of expressions. That means you can do this:

```
"Inches/yard: #{12*3}"        → Inches/yard:  36
"#{"Tora! "*3}"               →  Tora! Tora! Tora!
```

The second example is confusing, unless you remember that everything is an object in Ruby (yes, even string literals! They are of class `String`.). Since the string literal creates a String object, you can act on it just like any other object. In this case, multiplying a string by 3 simply does what you would think: makes three copies of the string.

Another, less awesome method of creating strings is using a special delimiter: `%Q` or `%q`. The way this constructor works is to follow `%Q` or `%q` with any non-alphanumeric, non-multibyte character. For example:

```
%q{Hoagies & grinders!} → Hoagies and grinders!
%Q;#{"Navy beans! "*3}; → Navy beans! Navy beans! Navy beans!
```

Note that `%q` acts like a single quoted string and `%Q` acts like a double quoted string. Just associate them by size: little q, one quote but big Q, two quotes.

Yet *another* way strings can be created in Ruby is the use of the verbose eyewart known as here documents (Perl programmers rejoice!), also known as

"heredocs." These unfortunate language constructs create a string by specifying a delimiter after a set of << characters to start the string and putting the delimiter on a line of its own to end it. For example:

```
my_string = <<MY_STRING
       This is a simple string that is
       pre-formatted, which means that the
       way  it is formatted here including
       tabs and newlines will be duplicated
       when I print it out.
MY_STRING
```

The final method that can be used to create a string instance is to simply use the `to_s` method of an object. Many objects simply output the standard results for this method (i.e. their class name and instance id or something similar), but others provide better faculties. For instance, `Fixnum` will actually return a string of the number value rather than simply a big blob of Ruby data.

Numbers

The second type we will look at is Ruby's built-in classes for numbers: `Fixnum` and `Bignum`. When creating a numeric object, any integer that is between (-2^{30}) and (2^{30} - 1) is assigned to an instance of `Fixnum` and anything else outside that range is assigned to an instance of `Bignum`; Ruby does this assignment transparently so there is no need to worry which one to use if you create a bookkeeping application for yourself and your bank balance (like mine) sits below -2^{30} constantly.

Integers are created by entering the number you wish to use without quotes (lest it become a string). The particular format depends on which numerical base you plan on using. Ruby supports standard decimal (base-10) operations but it also support operations on octal (base-8), hexadecimal (base-16), and binary (base-2) numbers. For example:

```
-123456789                  ⟶  -123456789        # Fixnum
0d123456789                 ⟶  1234567890        # Fixnum
1234323424231               ⟶  1234323424231     # Bignum
0x5C1                       ⟶  1473              # Hex
01411                       ⟶  777               # Octal
1_90_33                     ⟶  19033             # Fixnum
```

Notice that Ruby ignores underscores in numbers (some people choose to use them in place of commas for larger numbers to enhance readability). The examples also illustrate the various base notations. To create a binary number (base-2), prefix the number with 0b; to create an octal number (base-8), prefix the number with 0; to create a hexadecimal number (base-16), prefix the number with 0x. To create a standard, base-10 integer, either simply type the number as normal (i.e. 1678) or prefix it with 0d (i.e. 0d1678).

In addition to integer types, Ruby also has support for a Float type. Float numbers hold numbers that are fractional (i.e. they have a partial value that is expressed in decimal form). For example:

```
1.5                    ⟶  1.5
1.0e5                  ⟶  100000.0
1.e5                   ⟶  !NoMethodError
```

Each side of the decimal point must contain a number. When notating floats using scientific (or condensed) notation, you must place a 0 next to the decimal point or Ruby in its silliness will try to execute a method named (for example) e5 on class `Fixnum`.

Since numbers are objects (i.e. since everything is an object in Ruby) they also contain methods that can act on them. You can get a number's size with the size method, convert a number to a string using the `to_s` method, and many others:

```
-4.abs                 ⟶  4
6.zero?                ⟶  false
```

The above methods are obviously named (the abs method gets the absolute value and the `zero?` returns `true` if the number is zero), but they are not the only methods that are offered. Check the Ruby API Documentation for more information.

Numbers also offer methods that may not seem like methods at first glance: the arithmetic operators. Here are some examples:

```
2 + 2                  ⟶  4
6 / 3                  ⟶  2
-4 * 2                 ⟶  -8
```

A full listing of these operators and their function is available below. A quick tip: if you've ever programmed in another language, chances are they are the same (unless you've been programming in some sort of willy nilly non-mathological language).

ARITHMETIC OPERATORS

+	Addition
-	Subtraction
/	Division
*	Multiplication
( )	Order of operations (i.e. group expressions to force a certain order of operations)
%	Modulus (i.e. the remainder for those not in the know)

COLLECTIONS

It is a great thing to be able to push data around in its singular form, but everyone knows that collections are where the party is at (at least that's what MTV says). I think God once said that it's not good for data to be alone, and Ruby provides a few ways to facilitate this.

A collection (sometimes called a container) is an object that holds a group of related objects; this relation could be by type (i.e. all of them are strings), purpose (i.e. all of them are names), or by favorite cheeses (mine is provolone). A collection can be used to house a number of data items, keep them organized, and perform operations across all its members; each member (or element) of a collection is also a separate, visible object that can be operated on (i.e. it can still call methods, be added to and subtracted from, etc.).

The Range

The first and most primitive is the range. Ranges hold a sequential collection of values, such as all numbers between 1 and 9 or the letters from A to Z. A range is created by placing a series of dots (or periods or decimals or whatever it is you kids call them nowadays) between the lower and upper limit of the range. For example, if you were creating a roleplaying game and wanted to set the possible ranges for the height of each race (in inches), you could type:

```
human = 48..81
elf = 40...68
grotesquely_huge_guy = 120..132
```

Ranges can use either two dots, which indicates an inclusive range of all values including the beginning value and the end value, or three dots, which excludes the last value. This seems backwards at first glance, but in truth that third dot is so fat that it pushes the last element out of the range. I am not kidding; crack open a debugger and find out for yourself. For example, the range 1...7 would produce a range like this:

On the other hand, the range 1..7 would produce this:

Now that you can get the right values in a range, you may want to actually do something with them. Ranges offer a number of ways to test and compare them. Firstly, you can compare ranges to one another using the == operator (more on this

operator and others later) or the `eql?` method. If you were to write software to manage bake sales (which I hear that's a booming market in the software industry right now) then you may write some test code to test the probability of the range of good and bad cookies you can expect from a batch:

```
good_cookies = 1...3
bad_cookies = 1..3
burnt_cookies = 1..3

puts(good_cookies == bad_cookies)           →  false
puts(good_cookies.eql?(burnt_cookies))      →  false
puts(bad_cookies == burnt_cookies)          →  true
```

Ranges are considered equal if their beginning and end values are the same, but note that even though the `good_cookies` and `bad_cookies` shared the same beginning and end value in code, the values differed. The values were changed by the value of the inclusive flag (remember the two dot-three dot thing?). The values for `good_cookies` are `[1,2]` while `bad_cookies` holds `[1,2,3]`.

Ranges also offer a way to test whether or not a value is contained within a range using `===` or the `include?` method. For example, if you and your co-worker guessed a number of good cookies, but wanted to see if it was within the probable range of good cookies, you could do this:

```
my_guess = 2
his_guess = 19

puts(good_cookies === my_guess)             →  true
puts(good_cookies.include?(my_guess))       →  true
puts(good_cookies === his_guess)            →  false
```

The `include?` method will return any value that is contained with the range of values in the range (i.e. it would return `true` if you tested 2.44564 against `bad_cookies`); if you're feeling a little alternative, you can also try `include?`'s alias `member?`.

The Array

The second built-in collection is the array, an integer indexed and ordered collection of elements. If you have had any introductory computer science course, this concept of an array should not be foreign to you but Ruby's implementation may seem slightly unfamiliar to you. While the indexing is zero based like C/C++ and Java (i.e. the first element is referenced at index `0`, the second element `1`, and so on), unlike these languages, the elements in a Ruby array do not have to be the same type; nor does the type of the array have to specified before it is initialized for use. So, without thought to types, you could end up with an array that's something like this:

Figure 2: Look, ma! No types!

In Ruby, literal arrays can be created and stuffed with values in a variety of fun and interesting ways:

```
its_so_empty = []
oh_so_empty = Array.new
hello = ['ni hao', 'bonjour', 'hi', 'howdy']
random_types = [13, 'napkin', (1336 + 1).to_s]
```

An array can be initialized with values of any type, even variables, values returned from methods, literals such as quoted strings, or nothing (to create an empty array). This is handy mostly for literal values, but Ruby offers a few more methods for creating arrays that are more convenient and certainly more *Rubyrific*. Strings offer a special way to create arrays from their contents. Let's say you were writing haikus and wanted to make sure each line (which is conveniently filled with one syllable words) matches the ol' "5-7-5" paradigm by splitting the line into an array so you can count the elements:

```
my_haiku = %w( my dog digs it here\n )
⟶  ["my", "dog", "digs", "it", "here" ]

my_haiku = %w( he is nice to me & cats\n )
⟶  ["he", "is", "nice", "to", "me", "&", "cats"]

my_haiku  = %W( but he ate #{(2*3)/6} once )
⟶  ["but", "he", "ate", "1", "once"]

my_haiku  = %w( but he ate #{(2*3)/6} once )
⟶  ["but", "he", "ate", "#{(2*3)/6}", "once"]
```

Oops! A string wrapped in the `%W` delimiter acts like a double quoted string: it performs string interpolation and extended escape sequence substitution, but `%w` delimiter acts just like a single quoted string: it only allows a subset of the escape sequences to be used and does not facilitate interpolation. Some are confused by all of this poppycock, but it's very easy to remember: Bigger is better (unless you don't need all the fancy features or you have some sort of religious convictions against double quotes and/or capital W's).

The last way to form arrays that I would like to mention is the `to_a` method of some objects. This method converts an object or (rarely) one of its members to an array. For example, ranges support this method:

```
my_range = 1..10
→  1..10

my_dazzling_array = my_range.to_a
→  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Many objects implement this method; it is a convenient way to get an easily manipulatable data structure from some silly classes that are difficult to work with. You may consider peeking in the Ruby API documentation to see if the object you wish to use this method with does indeed implement it.

Now that you have an array, maybe you want to add to it. Elements can easily be added to an array by simply assigning a value to a non-existent index; for example:

```
my_dazzling_array[10] = 11
→  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

my_dazzling_array[12] = 12
→  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, nil, 12]
```

If a gap exists between indexes of the last element in the array and the newest added element, Ruby places `nil` (i.e. the equivalent of `null` in other programming languages; it represents a complete lack of value) in the gap elements (look at `my_dazzling_array[11]` above). If you simply want to add an element to end of an array, then you can use the << operator, the `push` method or certain forms of the `insert` method. For example:

```
my_dazzling_array.push(15, 16)
→  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 16]

my_dazzling_array.insert(-1, 17)
→  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 17]

my_dazzling_array << 14
→  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14]
```

The `push` method allows you to push one or more elements onto the end of an array; each element should be provided as a parameter to the method. The `insert` method allows you to insert elements at (the specified index + 1); in the example, I used -1 to add elements to the end of the array (i.e. -1 moves from the end of the array back one element to the last element. Adding an element after the last element would effectively add it to the end.). This method probably is not the best, but it can be used when the same method needs to insert elements at various places in the array (including the end). The << operator allows you to push specified elements on to the end of an existing array; I pluralized element because several of these "appends" can be chained together to add numerous elements to the end of an array. For example:

```
my_dazzling_array << 20 << 21 << 22
→  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 21, 22]
```

Now that you have data in your array, what are you to do with it? I personally like to admire my data, pride myself on my ability to harness all that is the array, and thrust my fist into the air yelling, "I AM SPARTACUS, LORD OF THE ARRAY!" Then my wife looks at me like I am crazy, it gets all weird, and I get back to work. I suppose that may or may not work for you; most people opt to simply use their array to hold stuff and call it when they need it (which is not nearly as fun). To make use of an array element's value, you simply reference the desired element by index in the form `array_name[index]`. For example:

```
puts my_dazzling_array[0]
→  1

index = 2

my_dazzling_array[index]
→  3

my_dazzling_array[0..2]
→  [1, 2, 3]
```

Notice that when referencing array elements, you can reference a single element with an integer or use a range to reference a number of elements. Remember, because array indexing is zero based, referencing index `0` is actually referencing the first element. Also, be sure that when you pass an index, that it is an integer or range; if you do not, Ruby will throw a `TypeError` (which will in turn crush your soul). This seems like a silly problem (and an even sillier consequence!), but it could show up if you were (for some reason) reading indexes from sockets or files (which reads everything in as strings so you would have to convert them which you'll learn how to do later). This method for referencing array elements operates just like the `at` method:

```
puts my_dazzling_array.at(0)
→  1
```

Another method, `fetch`, can also operate in this manner, but `fetch` can also specify a default value to return if the specified index is not found.

```
puts my_dazzling_array.fetch(999, "Not found!!")
→  Not found!!
```

Yet another method, `values_at`, can also operate just like `at`, `fetch`, and the `[]` operator, except this method can take a number of indexes to fetch and return as an array.

```
puts my_dazzling_array.values_at(0, 1, 2)
→  [1, 2, 3]
```

The last ways that I would like to share to retrieve elements are the methods `pop` and shift. The `pop` method grabs the last element in the array and removes it from the array; the `shift` method grabs the first element from the array and removes it shifting all other elements back one index.

```
my_dazzling_array.pop
→  [1, 2, 3, 4, 5, 6, 7, 8, 9]

my_dazzling_array.shift
→  [2, 3, 4, 5, 6, 7, 8, 9, 10]
```

So now you have an array, you have data in it, but maybe you are sick of that third element. He just keeps giving you crazy looks or eying your wife and you just want to take him out or possibly *extirpate* him. Well, fortunately Ruby can take care of that little problem, and let's just say it ain't pretty. The `delete_at` method deletes the element at the index specified as a parameter and returns the value of that element. For example:

```
puts my_dazzling_array.delete_at(1)
→  2

my_dazzling_array
→  [1, 3, 4, 5, 6, 7, 8, 9, 10]
```

Another method that arrays offer to delete items is the `delete` method (big surprise, huh?). This method deletes and returns the value that is referenced as a parameter rather than the index like `delete_at`. For example:

```
puts my_dazzling_array.delete(4)
→  4

my_dazzling_array
→  [1, 2, 3, 5, 6, 7, 8, 9, 10]

puts my_dazzling_array.delete(1337) { "Wrong!" }
→  Wrong!
```

Note that the item with the value of 4 was deleted rather than the index 4. Also note that the `delete` method offers the option for a "default" value to return if the specified value does not exist in the array; the last example is a demonstration of this feature. The value returned by the block of code between the braces will be the value returned if the item is not found (I talk more about these kinds of code blocks later on; if you're confused, curious, and impatient, look at page 31).

The Hash

The last collection type that Ruby offers is the hash (also sometimes known as an associative array or dictionary); hashes are collections of values of any type indexed by other values of (almost) any type rather than solely numbers like arrays (though you can use numbers for hashes also). For example, they can be indexed by strings; if you had a hash called thathash, you could call out its keys by name.

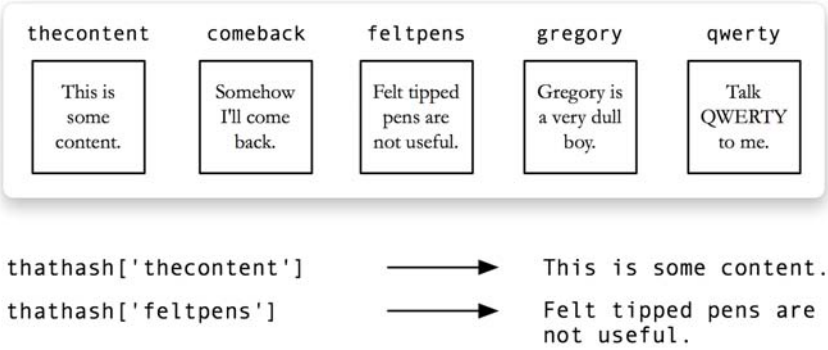


Figure 3: The hash, illustrated. Those darn felt tipped pens.

I say they can be keyed by almost any type because indexes (called keys in hashes) have two requirements: they must implement the `.eql?` method and they must have a constant (or constantly updated using the `rehash` method) hash code. These requirements stem from the way hashes handle indexing and looking up values by key. You can find a fine, technical explanation of this process of looking up and hashing and what have you in other volumes, but let us suffice now to say that the hash code is the hash's method for comparing and finding keys and the like, so it would not be wise to let that get thrown all willy nilly by a rogue or inaccurate hash code (strings are exceptions since Ruby makes a copy of them rather then references them; this way the value cannot be altered and the hash code changed without the hash knowing about it).

To create a new hash, you simply bracket nothing or a set of key value pairs (indicated by the "=>" combination) between a set of braces. If you place nothing between the braces (or you call the `Hash.new` method), then an empty hash will be created, but let's say you wanted to create a giant hash of the names of everyone that you know who is a wombat and where they live. You could do so like this:

```
my_wombats = { 'Wally Wombat' => 'The Jungle St.',
       'Wilma Wombat' => 'The House on the Corner',
       'Sam' => 'Notawombat Way', 'Mump' => 13 }
```

Okay, so maybe it's not a giant hash (yet). Anyhow, Ruby doesn't mind whitespace (spaces, tabs, and the like) or newlines (or the lack thereof) in the hash definition, so

long as you put a comma between each entry. I used string keys in my example, but any object can be used as long as it meets the requirements for keys (listed above).

Ruby also offers the `new` method for creating hashes; the `new` method for hashes varies slightly than what would be expected. One would expect that parameters provided to `new` would become the values of a hash, but in this case it takes a single parameter which becomes a default value if a nonexistent key is referenced (or if the `default` method is called).

```
new_hash = Hash.new("Not here!")

new_hash['non-existent key']
→  Not here!
```

Keys can be added to a hash by simply defining them; continuing from the above example, let us assume that you met a new wombat at the supermarket:

```
my_wombats['Wombie McWombat'] = '123 That Street'
```

Note you don't redefine the hash or enclose anything in brackets or braces; you don't have to call any silly methods (you *can* use the `store` method, but why would you?) or use any operators that seem foreign; you simply assign a value to the key.

So now that you have this fine hash o' wombats, what can you do with it? To reference a hash value, you can use the `fetch` method with the key as a parameter (lame!) or simply reference it similar in an array-like form: `hash_name[key]`. For example, if you were throwing a wombat party and wanted to invite all your wombat friends, but you couldn't remember Wally Wombat's address, you could print the value of his address like this:

```
puts my_wombats['Wally Wombat']
→  The Jungle St.
```

Hashes also offer the `values_at` method, which allows you to provide numerous keys and receive their values in an array. Values in hashes have to be called using these methods (`fetch`, `[key]`, or `values_at`); hashes do not offer methods like `pop` in arrays. Hashes offer `shift`, but it simply returns the first element as an array which contains the key in one element and the value in another; this method and a couple of others are not very useful if you simply want the value of elements. Oh sure, hashes offer a myriad of methods which allow you to do different things with the elements of a hash (i.e. the `to_a` method which changes the hash to an array, `merge` to merge two hashes, `replace` to replace one hash's values with another's values, etc.), but there are not a whole of options when it comes to grabbing elements from a hash. Hashes lack a lot of *bling* to be honest.

Ruby also offers a few methods that test elements in a hash; these are helpful to grab information about the hash without traversing the whole hash. For example:

```
my_wombats.has_key?('Wilma Wombat')
→  true

my_wombats.has_value?('Lamps and pandas')
→  false

my_wombats.empty?
→  false
```

The method names obviously explain what they do, but I will explain them a little bit anyhow. The `empty?` method checks whether or not any elements exist in the hash; this does not check the values of the hash, so if there is an element which is empty it returns false. The `has_key?` method checks the hash to see if the key passed as a parameter exists; this is probably more safe than checking they key for the default value or nil. The `has_value?` method checks the values of the hash to see if it exists in one of the elements. This method is not particularly useful (since it does tell you which key has the value), but it can be useful if you want to make sure that any key has this value. There are a lot of synonyms for `has_key?` (`member?`, `key?`, etc.) and `has_value?` (`value?`, etc.); check the Ruby documentation for a full list of these methods; maybe one of the synonyms will be easier for you to remember.

Hashes also, of course, offer methods to delete elements. To delete an element, simply call the `delete` method and provide the key you wish to delete as a parameter. Let's say that you had a falling out with Wilma Wombat (she got a little tipsy at the Wombat New Year party and vomited on your new eel skin Prada shoes) and you now want to delete her from your list of friendly wombats; you could do so like this:

```
my_wombats.delete['Wilma Wombat'] ⟵ This should be ()
→  The House on the Corner
```

When an element is deleted, its value is returned. This works great if you know exactly which keys you want to delete, but let's say that you get tired of how wombats smell (the smell of stale gin and guava juice can be rather unwelcome in the morning). You want to completely blow away the whole hash, but it seems silly to go through each key and call `delete`. Well, fortunately Ruby delivers:

```
my_wombats.clear              →  {}
my_wombats.empty?             →  true
```

When the `clear` method is called, the newly emptied hash is returned; aren't you glad that you don't have to deal with those darn wombats anymore?

VARIABLES AND THE LIKE

Now that we've gone over all the basic objects you have to work with (at least the important ones), we should probably talk about how to do something with them. It

would be silly to learn about what you have to work with without working with it (unless of course you were learning about poisonous snakes or Malaysian sloths); to do anything practical with an object you probably need to store a reference to it somewhere like a variable or a constant. We did this wantonly in our discussion of types, but now may be a good time to go over the finer points of assignment, expressions, and other fun party games with objects.

In the examples for working with the standard Ruby types, I often demonstrated variable assignment without really explaining exactly what was going on. It seems obvious what was going on: left value (a.k.a. lvalue) is set equal to the value of the right value (a.k.a. rvalue). Seems like second grade math, right? But notice that I said reference in the previous paragraph. Variables are, in elementary terms, names for values that live in the memory of your computer. In Ruby, variables point to a location in memory. If you point a pointer to another pointer, your point to the same location in memory.

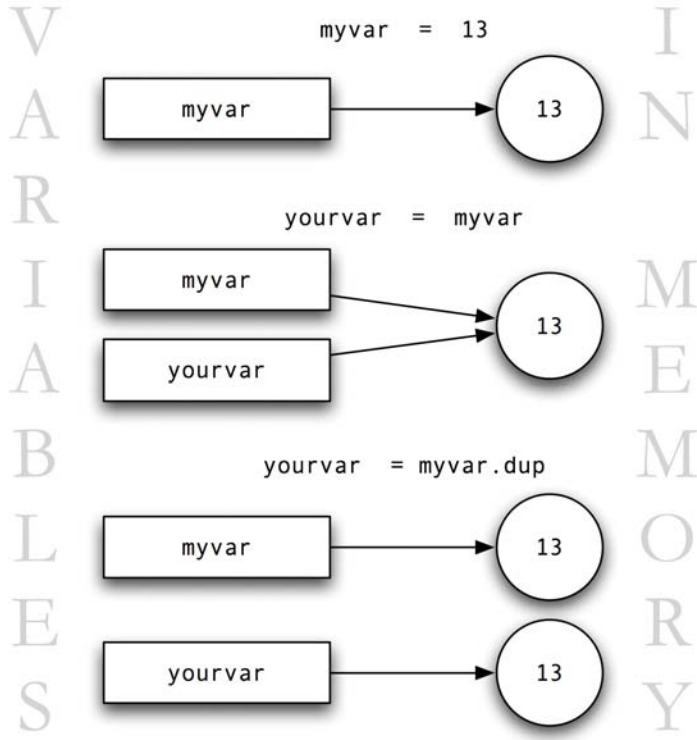


Figure 4: Variables as references. Your computer's memory looks suspiciously like a group of circles.

If you don't really get it, perhaps an example will work a little better to illustrate what references mean in practice:

```
first_var = "i hold a reference"
→  i hold a reference

second_var = first_var
→  i hold a reference

second_var.chop!  # Chops off the last character of the string
→  i hold a referenc

first_var
→  i hold a referenc
```

Wait a second! I modified `second_var`! Why is `first_var` different now? This the where the idea of a reference comes into play: variables are not objects but references (or pointers) to objects which live in the magical ether beyond (or the heap; whatever you kids call it nowadays). References (and, in turn, variables) merely point to objects; they do not hold the actual objects themselves. When you reference a reference, it does not duplicate the object: both references point to the same object. If you want to duplicate the object (i.e. create another copy of the object in another object rather than simply referencing it), you can use the `.clone` or `.dup` method on objects which offer it.

While assignment of the rvalue-into-lvalue sort is simple enough to understand due to its readability, there are other forms of assignment and all manners of bit twiddling and binary bruhaha you can pull on variables. One such ruckus you can stir up is chaining assignment. In a normal assignment, there is one lvalue and rvalue; after the assignment the lvalue equals the rvalue. When you chain assignments though, magic happens (well, not really). For example:

```
left = 5                    →  5
left = middle = 7           →  7
left                        →  7
middle                      →  7
```

In this example, the chaining results in two lvalues and one rvalue. This seems tame and practical enough; if you need to assign two variables the same value, you just place them as lvalues to the desired rvalue. Where it can get crazy is using something like this statement:

```
t = h = i = s = i = s = c = r = a = z = y = 100
→  100
```

Now every variable to the left of the final rvalue (`t, h, i, s, i, s, c, r, a, z, & y`) is set to the final rvalue (`100`). Though it seems like all the variables are being set in parallel, in actuality Ruby assigns them working from right to left (i.e. `y` is set to 100, `z` is set to `y`, and so on). Ruby does offer setting variables in parallel, but it is accomplished using a slightly different form:

```
p1, p2 = 1, 2                    →  [1, 2]
p1                               →  1
p2                               →  2
```

Note that Ruby returns an array of the assigned values. This form is a great method for swapping the values of variables (since they are actually set in parallel). You can also make use of this form with an array:

```
rvalue = 0                       →  0
a, b = rvalue                    →  0
a                                →  0
b                                →  nil
rvalue = [1, 2, 3, 4, 5]         →  [1, 2, 3, 4, 5]
a, b = rvalue                    →  [1, 2, 3, 4, 5]
a                                →  1
b                                →  2
```

Any array can be assigned to a list of variables; as shown in the example, Ruby ignores any extra elements in the array past the number of variables specified to assign to. This is useful if a method returns an array but you don't necessarily need anything past the first few elements. Notice that earlier in the example that any lvalues without corresponding rvalues are simply set to `nil`. Arrays can also be assigned in parallel in nested assignments ; Ruby is smart enough to pick apart your expressions into individual objects and try to assign them (which is a big step up from languages like C++ and C#). For example:

```
a, (b, c), d = 10, 11, 12, 13
→  a == 10, b == 11, c == nil, d == 12

a, (b, c), d = 10, [11, 12], 13
→  a == 10, b == 11, c == 12, d == 13
```

Much like the other form of parallel assignment, Ruby substitutes `nil` for lvalues which do not have a corresponding rvalue. The first example's `c` does not get assigned because it is in an array with `b` and the corresponding rvalue is not an array. In this case, Ruby assigns the first element the value.

Another form of ridiculous rvalue rigormoralitry is the additive assignment operator (`+=`) and the subtractive assignment operator (`-=`). These forms are somewhat similar to (but also replace) the `++` and `--` operators seen in many programming languages. The `-=` and `+=` operators are delightful pieces of syntactic sugar that make adding and subtracting objects and assigning the returned value to the initial object a breeze. For example:

```
lumps += 2            # lumps = lumps + 2
→  2
```

```
pie += lumps          # pie = pie + lumps
→  2

lumps -= pie          # lumps = lumps – pie
→  0
```

As you can see, these shortcuts allow you to accomplish the same thing in a whole lot less typing (every programmer's dream, right?). Also note that these operators work on more than numbers; anything that uses the + and - operators can use them since these syntactic sugar lumps merely wrap these operators. This means that anything that happens during normal use of these operators (i.e. certain objects perform extra work when adding or subtracting that is built in or that you specify) will still happen.

What if you don't want to be able to assign to an object? It's a rare case indeed unless you're trying to work a bug out or if you simply like to raise unhealthy amounts anger within yourself because you happen to be the Incredible Hulk. Freezing an object is useful if your program is acting wonky and spitting out an abnormal variable, but from what you can see, it should be working normally. So, you would simply `freeze` the object at the last line of code you see behaving normally:

```
# Lots of code here...
my_crazy_object = why_do_you_hate_me?
my_crazy_object.freeze

# Even more code...
my_crazy_object = abnormal_value
→  TypeError!  can't modify frozen object
```

This seems like a cool trick you'd use often, but it's really not. I suggest not using it unless you absolutely need to and you have permission from your mom and dad first.

Another crafty piece of syntax you may spot when looking at others source code or examples in books or the web is something that may look like this:

```
my_string =~ /\sstring\s/
```

What's that tilde for?! And what's with the slashes and the literal and the escape sequence outside of a string?! This is what's called a regular expression, a pattern that is used to match string or portions of strings in order to execute some manner of string manipulation. Ruby offers a very robust regular expression facility (which we will touch more on later), but right now let's suffice to say whatever is between the slashes will be matched and assigned to the lvalue when the =~ operator is used. For example:

```
my_string = "my string is looooong"

my_string =~ /\sstring\s/                    →  2

my_string =~ /\s/                            →  2

my_string =~ /my/                            →  0
```

The pattern enclosed in the slashes is matched to the string using the =~; using that pattern, the index of the first match (i.e. an occurrence of a string matching that pattern) is returned. I realize this is a rather cursory rundown of what regular expressions can do. I will discuss this more in detail later on in this chapter, but for now I thought it beneficial for you to be familiar with that if you see it somewhere before you get there.

This Chapter

You learned about Ruby's object system and built-in classes. You learned...

- that everything in Ruby is an object.
- that the basic built-in classes in Ruby are the number (`Fixnum` and `Bignum`), the `String`, the `Range`, the `Array`, and the `Hash`.
- that setting a Ruby variable is actually setting a reference rather than a value, but can be set to values if needed.

2

Break it down now!

Now that you are familiar with some basic objects and how to manipulate them a little bit, we should probably move on to segmenting your code; surely you didn't think that applications ran in one huge chunk of code! It would be silly if you had to repeat every piece of code you wanted to use again or worry constantly about if you were stomping on a variable you used 30,000 lines ago, so the language geniuses have created a few ways to segment your code.

The most fundamental of these is a block; blocks are just pieces of code segmented away from their context by an initiator and the end keyword. The initiator could be the `begin` keyword (which I will use as an example here) or something like `if` or `for` (which you will learn about in the next section). To create a block of code, you simply place the initiator on a line followed by any needed requirements (which are nothing for the `begin` keyword), followed by the code you wish to place in the block, and ended by the `end` keyword (I'm sure that one will be hard to remember...). Here's an example:

```
begin
  puts "I'm in a block."
  puts "Still there..."
  puts "Stillllll in here..."
  puts "OK, I'm done."
end
```

Using a `begin/end` block by itself really doesn't afford you anything except to show that it is separate from its context, but they can paired with and meshed into various constructs to achieve different results. I'll touch on a lot of these constructs in this section and later on when I talk about using blocks more effectively (it's a hootnanny trust me!).

METHODS

In the examples above, I've been rather liberal in my usage of methods without much explanation. If objects/variables are the nouns of programming, then we could

describe methods as the verbs; methods "do stuff." In more technical language, methods are pieces of code that are called from within other code using variables called parameters (also known as arguments or options) fed to them from the calling code. Think of them as `begin/end` blocks that can be called arbitrarily. When I say arbitrarily, I mean that they can be called anywhere, anytime. There isn't a set "method" block that all method calls live in. I realize this business with parameters and such sounds a little confusing, but we've already sent parameters to methods when we've sent text to `puts` or a string to `chop!`. We've already been doing it without ever saying we were! I slipped it right in there on you; I'm a smooth criminal, I know.

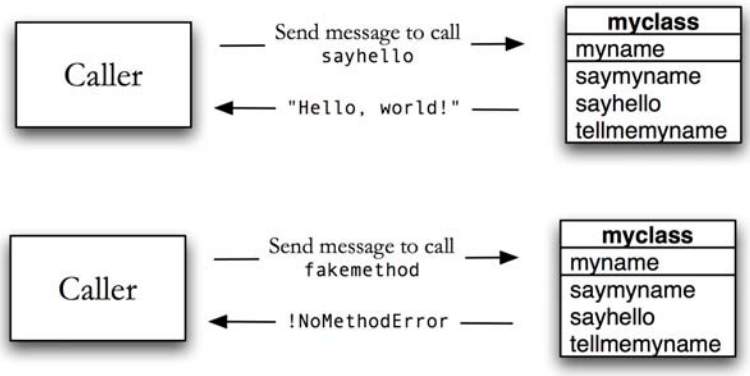


Figure 5: Calling of methods. It's much like pig calling, only electronic.

When methods are called in Ruby, you aren't technically "calling" a method (even though that terminology is often used to describe it). You are actually sending a message to an object saying, "Hey! Do you have this method?" If they do, the method is executed; if they do not, a NoMethodError is thrown and there is much weeping and gnashing of teeth. "Great, " you say. "But what are methods for? Sure, you can use them to 'do stuff,' but is there a 'bigger' purpose for them?" Of course.

Methods are used to, firstly, remove redundancy. For example, it would be silly to type the same 15 lines of code over and over again if you were going to be using them all through your application. You can just create a method and call it wherever you need it. Methods, secondly, allow you to segment your code better. Maintaining a 550 line piece of code is never fun for anyone (except extremely self-deprecating, abusive, masochistic crazy guys, but those guys work over in accounting right?); methods allow you to split up all the logic in that huge mess into smaller, more manageable chunks.

Defining Methods

I'm sure you are growing rather anxious in anticipation of being able to create your own shiny new methods. Well, wait no longer my impetuous friend! Here is that which you desire:

```
def my_new_method(name)
  puts "hey, " + name + ", this is my new method..."
end

my_new_method('magnus')
→  hey, magnus, this is my new method...
```

A method is defined by entering `def` followed by the method name and parameters (i.e. variables passed into a method to be used within that method, remember?); the following lines should contain the desired method code ended by the `end` keyword. That's it; simple enough, right?

Well, there's a bit more to methods than that. First, your method name should (by convention, of course) start with a lowercase letter (and preferably be all lowercase). The reason for this is that Ruby thinks that things that start with an uppercase letter are constants and classes rather than methods; this could cause some rather rascally behavior from your application. While we are on the subject of convention, there are other conventions that pertain to the name of methods. Firstly, if it is querying an attribute, it should end in a question mark; for example, if you were to write a method to get the number of French military victories, you could do something like `France.has_military_victories?`. This would, of course, return `false`. Another convention to follow is that if the method modifies its receiver in place (i.e. the method modifies the object that called it), then it should end in an exclamation point; for example, let's say you were replacing the existing cyborg software that all those robot celebrities run with a new snazzy Ruby-based system. To execute another facelift and make them look 40 years younger, you could do `Dolly_Parton.facelift!`, or to set their age to an arbitrary value, you could call `Bob_Barker.set_age!(30)`.

The next thing we should probably discuss about methods are the parameters (or arguments or whatever) that are passed in. These variables are passed into the method as local variables, which means they are local and usable only in the context of that block of code (the method). This means that the variables that created in that block of code and its parameters are not usable outside of that block of code. The language to explain it is a little dense, so let's look at an example:

```
def my_method(first, second)
  puts first
  third = second
  puts second
end

my_method("yes.", "no.")
→  yes.
→  no.

puts first
→  ! NameError: undefined local variable or method

puts third
→  ! NameError: undefined local variable or method
```

Notice that neither the parameters nor the created local variable are accessible outside the method unless they are returned or passed outside of it otherwise. This concept is known as scoping and will come up many, many times while programming in Ruby; I will highlight when scoping will be an issue with a new concept. Variables can be scoped globally, locally, class scoped, etc.; any block (including conditional blocks like `if`'s and loops) can have variables that are local to it. There will be more coverage of this as we progress, but it's important to remember scoping when working with methods; it can cause severe headaches if you aren't careful.

So now that you understand what parameters are and do, let's get fancy. What if you don't want to require a certain parameter? Or maybe you want a parameter to be able to take many parameters. Well, Ruby can deliver on both of those. Ruby offers optional parameters for methods; they aren't really optional so much as you can assign a default value to them. For example:

```
def new_method(a = "This", b = "is", c = "fun")
  puts a + ' ' + b + ' ' + c + '.'
end

new_method('Rails')
→  Rails is fun.
```

This technique is helpful if 99% of the time you'll be using the method with a certain value (either a parameter or local variable) but you want to be able to change that value every once in a while. You could pass in `nil` to the method every time you wanted to use the default value and filter it through a check or some hogwash like that, but that wouldn't save any typing nor would it make any sense. This language feature allows you simply specify the parameters you need and leave the rest as they as are; do note, though, when using this feature that the parameters must be in the same order and you can not skip any parameters in the list (i.e. it's best to place the ones you won't be explicitly defining that often at the end of the list).

Parameter lists can also be variable length; let's say that you wanted to create a rather contrived method that outputs your relations based on parameters you provide. The method could look like this:

```
def print_relation(relation, *names)
  puts "My #{relation} include: #{names.join(', ')}."
end

print_relation("cousins", "Morgan", "Miles", "Lindsey")
→  My cousins include: Morgan, Miles, Lindsey.
```

I could have provided any number of names in the list; by placing an asterisk before the identifier for the last parameter, you can turn it into a variable length list (which is actually just an `Array` created from the objects you provide, which is why we can use the `join` method with it). This technique is helpful when dealing with lists of objects or maximizing the flexibility of the method (i.e. using a method to process one or more objects in one fell swoop rather than calling the method several times).

Using Methods

Now that you know how to create methods, you would probably like to know how to use them more effectively (or simply at all). As you have seen in previous code examples, calling a method is as simple as putting the method name followed by the required parameters (if there are any). There are many ways to format a method call; sometimes a method may not require any parameters so the parentheses and parameters are not needed. Many times you can call a method without using the parentheses, but this is generally not good practice (it's silly to sacrifice readability just to save two keystrokes unless you're only passing one parameter). Let's look at some examples:

```
puts "Look ma!  No parentheses!"
puts("Look ma!  Parentheses!")
puts
puts()
```

All of the above examples are valid calls of `puts`. The first two examples demonstrate the optional parentheses usage; the second set merely demonstrates that not all methods need parameters. Do note that most methods do require parameters and will throw an `ArgumentError` if they don't get the correct number.

So methods are great, right? But how do we do anything with them? What good are they if the variables used in them are useless outside of them? This is where a method return value comes into play; a method allows you to return one or more values from within the method to be used outside of it. Methods always return a value, but if no value is explicitly specified, the value returned is `nil` (e.g. when a method is defined, Ruby returns `nil`) or the last value used inside the method (if that exists). For example:

```
def return_me(value)
  scoped_value = value
end

def echo_me(value)
  value
end

def multi_return
  return 'more', 'than', 'one'
end

my_value = return_me('this is fun!')
puts my_value
→  this is fun!

one, two, three = multi_return
puts one + three
→  more one
```

If no return statement is placed inside the method, the last value used in the method is returned; this value can be either a variable that has been assigned (as in the first example), an object that is created (e.g. placing a string literal on a line by itself because that creates a `String` object), or any other object that is referenced in the last line of the method (as in the second example). This means that a `return` command or

final reference isn't required if the last value used is the value you would like to return (as in the first example); if this is not the case, the second example demonstrates the usage of the final reference method of returning a value and the last example demonstrates usage of the return statement. The last example demonstrates using return and how you can assign variables in parallel (like discussed in the section on variable assignment) with method returns; since it is simply populating an array that is collected from the lvalues you specify, you can also use this method to populate arrays.

BLOCKS AND Proc OBJECTS

I mentioned blocks early in this chapter, but I'd like to cover them more in depth now. Blocks are a very powerful concept in Ruby, but very confusing for the newcomer, so some discussion is in order. In Ruby, a block is an object that contains some Ruby code along with the context neccesary to execute it. It doesn't make sense to say that a code block is an object, but remember that everything in Ruby is an object.

Block Basics

I said earlier that blocks are simply code wrapped in a do/end construct, but they go a little further than that. Blocks can be constructed in a number of ways, and in doing so, create an object that holds code that can be passed to methods or held in variables. Put simply, a Ruby code block is much like a method without a name tagged on it. Perhaps this will make a little more sense if you think of them as being very similar to C's function pointers, C++'s function objects, Python's lambdas and list comprehensions, Perl's anonymous functions, Java's anonymous inner classes, or even closer, Smalltalk's or Lisp's blocks. If you've used any of these languages and none of those sound familiar to you, this isn't very uncommon: typically they are shunned by all but experts in the language. Fortunately for you, I'm going to make you learn about them (they're an important concept in any language!), and even if you don't want to learn about them, too bad: you can't write Ruby without them.

Let's take a look at a simple usage of blocks: method parameters. A lot of methods take blocks as parameters, so let's look at one of those now.

```
myarray = %w{one two three four}
myarray.each {|element| print "[" + element + "]... " }

→  [one]... [two]... [three]... [four]...
```

This snippet simply iterates an array using the `each` method and passes in each element to the code block; the code block can then treat that element as an "argument" and operate it much like you would a method argument. The code block in this case is formed using braces; this is another way other than the `do/end` combination that you can form a code block. Although it looks like you're using the `each` method to "open" a code block, you're actually passing that block of code in as a parameter to the `each` method. If you're completely lost, perhaps breaking down this example will clarify

this example a little more. If you get the concept, skip the next paragraph; it'll just be redundant.

Let's take the following line of code apart and look at each part of this call separately.

```
myarray.each {|element| print "[" + element + "]... " }
```

We first call the `each` method on the array object `myarray`. This method takes a block as a parameter; that is to say that it takes a parameter which is a block of code that it will execute. This block is very similar to the `begin/end` blocks we saw earlier; we could rewrite the above code as follows if we wanted to.

```
myarray.each do |element|
  print "[" + element + "]... "
end
```

Notice that the braces are simply replaced by `do/end`. Both notations do the same thing, but the brace notation (i.e., `{ }`) is more concise and makes more sense if you only have a line or two of code. At a certain point in this method (which will be discussed later when we talk about how to use blocks in your own methods), the code tells Ruby to pass a parameter to the block and run the block. Ruby does so and returns the value of the block code (if there is one) much like it returns the value of a method. Let's visualize this flow of control just to drive the concept home.

If you still don't get it, you need to. Go visit some of the links in Appendix A under the Documentation section; search on Google; visit some of the blogs on the aggregators under the Ruby Language section of Appendix A. Someone, somewhere has explained this concept in a way that you can understand if I haven't; I wouldn't drive this concept home as much, except that it's a very cool, useful, powerful, and essential concept in Ruby. If you do grasp blocks, then let's move on to how to use them in your own code.

Procs and Blocks

Think of `Proc` objects as blocks that are pushed into variables. The difference between them is there, but not important enough to worry about until you need to (and you'll know when you do). The primary difference is performance, but that will be discussed when we reach the other end of the problem.

`Proc` objects are simply instances of the `Proc` class that hold a block of code that is executable.

```
myproc = Proc.new {|animal| puts "I love #{animal}!"}
myproc.call("pandas")

→  I love pandas!
```

As you can see, a `Proc` is created when the constructor is called and given a block as a parameter. The code in the block is then stashed away in a `Proc` instance and can be called at any time. `Proc` objects are especially useful when you want to create a block of code and pass it around or generate new blocks from that one. To call the code in the `Proc` object, simply use the obviously named `call` method and it will call the code inside the block you gave it. For example, let's say that The Big T.V. Network has commissioned you to write a Ruby script that will display the mastheads for their new lineup of shows (which includes Banjo Bill's Bridge Brigade, Cucina Italiana with Rafael Ramirez Rodriguez de Jesus, and PANDA!monium). You simply need to display the text for the show on the console and their fancy graphics backend will do the rest (yeah, their technology is *that* awesome). The only problem is that because their hosts change so often (i.e. Mr. Rafael just replaced Ho Chi Minh as the chef on Cucina Italiana just a minute and a half after he started), there needs to be a way to specify a show name separately from the host name and be able to change the host name on the fly. You say, "Hey! Blocks could possibly do that!"

```
def make_show_name(show)
  Proc.new {|host| show + " with " + host}
end

show1 = make_show_name("Practical Cannibalism")
show2 = make_show_name("Cotillions in the Amazon")

show1.call("H. Annabellector")
→ Practical Cannibalism with H. Annabellector

show2.call("Jack Hannah")
→ Cotillions in the Amazon with Jack Hannah

show1.call("Kirstie Alley")
→ Practical Cannibalism with Kirstie Alley
```

This looks like a typical `Proc` call like we looked at before, but notice something that's going on here. We fed it the show name when the `Proc` was created, but we never mentioned it after that. How is that possible? When the `show` parameter for the `make_show_name` method passed out of scope (i.e. the method exited), it should have been destroyed. Ah, but this is one of the beauties of a `Proc` object: it preserves the context in which it was created and can access that context at any time. This is why our show name was preserved without any further effort on our part.

Another way to create a Proc object is to bind a block of code using the lambda method; calling this method is essentially equivalent to calling `Proc.new`.

```
myproc = lambda {|x| puts "Argument: #{x}"}

myproc.call("Texas forever!")
→ Argument: Texas forever!
```

As you can see, the lambda function will take a block of code and bind it to a Proc, just like Proc.new. What can't be seen from this example are some of the differences that exist. First of all, Proc objects created with lambda have stricter argument checking than those created with Proc.new.

```
lproc = lambda {|a,b| puts "#{a + b} <- the sum"}
nproc = Proc.new {|a,b| puts "#{a + b} <- the sum"}
```

```
nproc.call(1, 2, 3)
→ 3

lproc.call(1, 2, 3)
→ !ArgumentError (wrong number of arguments (3 for 2))
```

The `Proc` object created with `Proc.new` functioned fine when given too many arguments, but the `lamba Proc` with its Nazi-like argument checking threw an `ArgumentError`. What a jerk...jeez. Crashing the whole application just because he got *too many* arguments? Lame. So, anyhow, another distinction between the two is how they control the flow of your application. Objects created with `lambda` don't affect the flow of the application outside of the block when returning a value; `Proc` objects created with `Proc.new`, on the other hand, will exit their enclosing method when returning.

```
def procnew
  new_proc = Proc.new { return "I got here..." }
  new_proc.call

  return "...but not here."
end

def lambdaproc
  new_proc = lambda { return "You get here..." }
  new_proc.call

  return "And I got here!"
end

puts lambdaproc
→ And I got here!

puts procnew
→ I got here...
```

Note that in the case of `procnew`, the value returned is the value returned from the block. The `lamba`-created `Proc` object simply returns its value to its parent method, which can then stash the value in a variable or return it if it wants to. This is an important distinction to remember, because it can cause you a lot of headache if you are using `Proc` objects in a method and you can't figure out why the method keeps breaking (I speak from experience!). Now that you understand how to work blocks into your code using `Proc` objects, let's look at how to integrate them in tighter with your methods.

Building Blocks

There are a few ways to get blocks to work for you in your methods; the first way is that you can pass a `Proc` object in as a parameter just like you would any other object. This can get tedious, however, and, from what I hear, it also hits your performance pretty hard (I would put the hit on a level somewhere between being slapped with a greasy piece of bacon and the rapture). Fortunately, Ruby gives you a few ways you can put blocks to work with minimal fuss and performance degradation. Integrating blocks into your everyday code usage is quite simple; just combine in a sprinkle of

determination, a dash of `yield`, and a liberal application of closures in a small integrated development dish and bake at 400˚ for 15 minutes or until crisp.

Implicit Block Usage Outside of taking a Proc parameter, Ruby offers only one other way to use blocks as parameters, but this way is not only more intuitive, it performs better. I call this implicit block usage because you don't tell your method, "Hey, I'm using this block here," and then call it in the method. You simply yield control of the code over to the block; this won't really make sense without an example, so let me just show you a simple snippet.

```
def yieldme
  print "1. Enter method. "
  yield
  print "3. Exit method."
end

yieldme { print "2. Enter block. "}
→ 1. Enter method. 2. Enter block. 3. Exit method.
```

Notice what happens here. First, we enter the method and print out our first statement. Second, the yield method is called, and our block is executed. The thread yields control over to the block temporarily; when the block exits, the control is restored to the block's caller. Lastly, the rest of the method is executed. This is how the `each` method on arrays that we used earlier works. Let's say we wanted to rewrite that method for some reason (perhaps your pet raccoon who fancies bowlers convinced you to rewrite it); you could use `yield` to execute the block.

```
def myeach(myarray)
  iter = 0
  while (iter < myarray.length):
    yield(myarray[iter])
    iter += 1
  end
end

testarray = [1,2,3,4,5]

myeach(testarray) {|item| print "#{item}:"}
→ 1:2:3:4:5:
```

I realize this snippet might be a little over your head (especially that while line!), but bear with me because this is a simple enough snippet to understand. The while block creates a loop, which means that we execute the code inside the block a number of times (learn more about loops on page 53). Even though it may look complicated, the same concept applies here as before: the method yields control to the block. The difference here is that we passed a parameter to the block each time we looped over the code; this allows us to use variables from within the calling method within the block for processing. Using yield is an excellent way to implement an iterator like this for your own collections.

The Ampersand (&) Parameter If you prepend the name of the last parameter of a method with an ampersand, then the block that is passed to the method will become a `Proc` just as if you had passed it as a parameter normally. Well, not completely normally; it does a few tricks.

```
def ampersand(&block)
  block.call
  yield
end

ampersand { print "I'm gettin' called! " }
→ I'm gettin' called! I'm gettin' called!
```

I said it would become a `Proc`, so you can use `call` on it, but notice also that `yield` works. This is an interesting and helpful trick, since you may want to use `call` or `yield` in different cases.

YOUR OBJECTS LACK CLASS!

As stated (many times) before, everything in Ruby is an object; Ruby, of course, allows you to create your own objects through the creation of classes to describe them. If you've programmed in an object oriented language before (like C#, C++, Python, or Java), then the concepts of classes and objects should not be foreign to you, but there are some distinctions between those languages' implementation of object orientation and Ruby's implementation.

One thing that may seem rather foreign is the way that Ruby handles typing of objects. Languages like C++ or Java operate solely on static (or explicit) typing; this sort of typing requires that each object have its type explicitly defined at compile time (or the compiler will throw an error). I realize that most modern languages that use static typing also implement a sort of reflection or some such module that allows you to dynamically load types, but Ruby uses a completely different approach to typing. If you've used Python, you're familiar with the concept of dynamic typing; Ruby uses this same idea but calls it "duck typing" (which makes it much easier to explain).

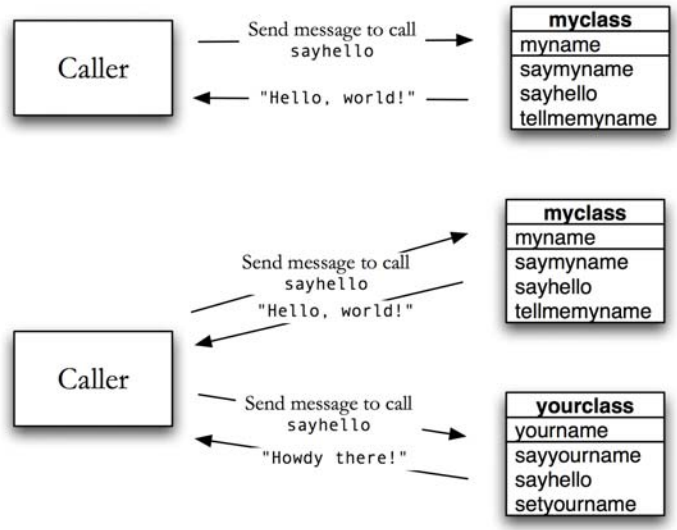


Figure 6: Both classes implement a method named sayhello, so they can both respond to the message. I would like to add that yourclass smells.

Another concept that may seem foreign to C++ or PHP programmers (but not Java or C# programmers) is the concept of a language-wide object hierarchy. In Ruby, every class is actually an object which is an instance of the `Class` class which is in turn derived from the `Object` class. We can demonstrate this with code:

```
puts File.class
→  Class
puts File.superclass
→  Object
puts Object.superclass
→  nil
```

The super class of a class is the class which it is derived from; in Ruby, we can say that classes can "inherit" from another class all of its methods and variables. As a result, if a class is derived from another class (i.e. it inherits from another class), it has access to all of the super class's methods and variables. The catch in Ruby is that unlike some other languages, a class can only inherit from one class at a time (i.e. a class can inherit from a class that inherits from another class which inherits from another class, but a single class can not inherit from many classes at once). As we can see, the `File` class is just an instance of the `Class` class which inherits from the `Object` class which inherits from nothing. This means that the `File` class has access to all of `Object`'s methods and variables (just like every other class). This also means that the `Object` class is the origin of all other objects; it is the Adam and the Eve, the Creator, the Architect, the Mother of all Objects and Classes!

I'm sure this talk of classes being an instance of something is somewhat confusing, but keep in mind: everything in Ruby is an object. Everything. So when you define a class, you're really creating an instance of the `Class` class. When you create a new object from a class, you're calling `(class name).new` which is a method that returns a new object instance of the class it describes. Everything in Ruby is an object!

Defining Classes

So let's get to it. To define a class, you place the `class` keyword at the beginning of a line, followed by a `<` and the class it inherits from (if it inherits from a class). For example:

```
class MyFirstClass < Object
end
```

That's it; we've just defined our first class. Granted, that was the most contrived example ever put into print and it represents a completely useless class, we still defined a class. Notice that I indicated it inherits from `Object`, but this is completely unnecessary; Ruby will assume that if you define a class with no other inheritances that you wish to inherit from `Object`.

Methods and Variables

Classes can contain variables and methods; the first thing you would most likely want to add would be a method so you can make your class do some work. The first method we should add is the `initialize` method, which is the method that Ruby calls when you call `(class name).new`. When you call the `new` method, a new object is created and the `initialize` method (with parameters passed from `new`) is called to setup the object's state (i.e. variables, flags, etc.); this is very similar to (albeit identical to) other language's constructors. For example, let's say that the Boogeyman has decided to give up on trying to freak out every kid in the world (he's rather old, you know) and instead build a robot army (running software written in Ruby no less) to do his nefarious bidding. The initial class definition and initialize method might look like this:

```
class Boogeyman
  def initialize
    puts "Yes, master?"
  end
end

monster1 = Boogeyman.new
→  Yes, master?
```

This method, of course, does no real work other than to demonstrate that when you create a new object, the initialize method is called. Let's make it do some work now:

```
class Boogeyman
  def initialize(name, location)
    @name = name
    @location = location
    puts "Yes, master?"
  end
end

monster1 = Boogeyman.new("Mister Creepy", "New York, NY")
→  Yes, master?
```

This new initialize method sets some instance variables (i.e. variables that are used within an object to retain its state); to set an instance variable, prefix the lvalue with an at symbol (`@`). Unlike other languages, you don't have to include these variables inside the class definition.

Variables set this way are unique to that particular instance; we could say that they are instance scoped (i.e. they are not usable outside of that instance unless they are passed outside of it; remember scoping?). Notice that since `@name` and `name` are scoped differently, we can use duplicate names without ambiguity (though this is usually not a good idea). Let's create a few methods to work with an object's state:

```
class Boogeyman
  def change_location(newlocation)
    @location = newlocation
    puts "I moved to #{newlocation}!"
    self.get_info
  end
```

```
  def change_name(newname)
    @name = newname
    puts "I shall be called #{newname} from now on!"
    self.get_info
  end

  def get_info
    puts "I am #{@name} in #{@location}."
  end
end

monster1 = Boogeyman.new("Loopy Lou", "Albuquerque, NM")
→  Yes, master?

monster1.change_location("Wyoming")
→  I moved to Wyoming!
→  I am Loopy Lou in Wyoming.

monster1.change_name("Beezlebub")
→  I shall be called Beezlebub from now on!
→  I am Beezlebub in Wyoming.
```

This example demonstrates two important concepts. First, notice that I did not enter the entire class listing again. This wasn't laziness (well, not completely at least); in Ruby, classes are never closed. This means you can always add methods to or redefine (or in proper terminology, override) any method for a class simply by opening a class definition and adding or redefining a method. This can be dangerous at times, but overall it's one of the most useful aspects of Ruby's object implementation. Let's look at an example:

```
class String
  def writesize
    puts self.size
  end
end

size_writer = "Tell me my size!"
size_writer.writesize
→  16
```

As I said before, while it's possible to override a class's methods (even built-in classes!), it can be dangerous (i.e. modifying some of `Object`'s methods or modifying certain operators can make everything go nuts), but at the same time, it can also be useful. The Ruby web framework Rails makes extensive use of this concept, especially in its ActiveSupport package; if you're looking for something a little more complicated and interesting, I suggest looking at their extensions to various classes in that package.

The second concept shown in these examples is the use of the `self` object. The `self` object always points to the current instance; it allows you to call methods from within the current instance (like `size` in the `String` class or `get_info` in our class). Though `self` isn't required in most cases (i.e. if no receiver is specified for a method, Ruby assumes you meant `self`), it is important to be aware of it in case you are in a context where you will need it (e.g. you have implemented a method named `puts` in your class and you want to call it and not the built-in one).

Attributes

While instance variables are useful in their own way, they aren't visible to the outside world. It may seem like a dandy situation at first: all your objects' states are completely hidden and unchangeable by the outside world. But after a while, you might just want to retrieve or change a value within an object. How are we to do this?! Well, it's really quite simple:

```
class Boogeyman
  def scare_factor
    @scare_factor
  end

  def hiding_place
    @hiding_place
  end

  def scare_factor=(factor)
    @scare_factor = factor
  end

  def hiding_place=(place)
    @hiding_place = place
  end
end

monster1 = Boogeyman.new("Crazy Cal", "Nashville, TN")
monster1.scare_factor = 6000
puts monster1.scare_factor
→  6000
```

As the example shows, to create a readable attribute, you simply create a method and place the instance value to return in it (the last value used in a method is returned remember?). Attributes are simply methods that are used to retrieve or set values. To create a writable attribute (i.e. an attribute you can set), you simply append an equals sign (=) after the name of the attribute method; you can either do like I did and write straight to an instance variable or do some other work before you do so (i.e. make sure the value provided is the proper type/format, convert formatting to a more usable form, etc.). This seems like an awful lot of work just to write to a value in a class doesn't it? I mean, in C# or something similar all I have to do is put "`public`" before the variable in the class and it's visible to the outside! Well, since attributes are such a common construct, Ruby has a really simple facility for them:

```
class Boogeyman
  attr_reader :scare_factor, :hiding_place
  attr_writer :scare_factor, :hiding_place
end
```

Now you can read and write attributes just as before; these faculties are a pretty way to create methods that behave identically to the ones we created before. This technique is easier than writing out methods, but you lose the flexibility you may gain by making your readers or writers explicit methods that you write. For example, let's say that the scare factor is supposed to be displayed in Freak-o-grams (Fg); you could write a reader to display it as such:

```
class Boogeyman
  attr_writer :scare_factor
```

```
  def scare_factor
    return str(@scare_factor / 1000) + "Fg"
  end
end

monster1 = Boogeyman.new("Psycho Sally", "Los Angeles, CA")
monster1.scare_factor = 6000
puts monster1.scare_factor
→  6Fg
```

Some would call these virtual attributes, but I really think they need a special name. It really doesn't matter what you call them, but they are a great way to mask the implementation of your class. To the outside world, it looks like a normal attribute, but you know the truth! It's your little way of sticking it to the man.

Access Control

So far our methods and attributes have been wide open to the world, but now let's take a look at ways we can control access to parts of our class. Up until now, all of our methods (except `initialize`, which is always private) have been what we call public (i.e. accessible from within the class and the outside world). Since that is the default behavior of Ruby, let's add a method to our class as an example:

```
class Boogeyman
  def scare(who)
    puts "I just scared the bejezus out of #{who}!"
  end
end
```

The method we just created is public because we didn't specify any access controls; we could create protected methods (i.e. a method that accessible by any instance of a class or its derived classes) by placing `protected` on a line and then entering subsequent methods which will be protected. For example:

```
class Boogeyman
  protected
    def are_you_a_monster?(whosasking)
      puts "Yes, #{whosasking}, I am a monster!"
      return true
    end
end
```

Now the only objects which have access to this method are those that are instances of `Boogeyman` and any class that is derived from `Boogeyman`. This is useful if you have a method like the one above that needs to provide information to classes of the same or similar type, but no one else. On the other hand, if you have a method that only the current object instance should have access to, then you should declare it private. The difference between protected and private is only slight: protected allows any instance of the same or derived class to access it but private allows only the current instance to have access. Let's add a method to phone home to the Boogeyman himself and redefine our `scare` method to use it.

```
class Boogeyman
  private
    def phone_home(message)
```

```
      # TODO: Actually make this phone home
      # For now, we'll just write to the console
      puts message
    end

  public
    def scare(who)
      phone_home("I just scared the living poop out of #{who}!"
    end
end
```

Now only the current instance will have access to the `phone_home` method; we wouldn't want just anyone phoning home and making it look like this monster would you? Notice that you can use the `public` keyword in the same way to make methods explicitly public or in this case to change the mode back to public after defining a private method.

Class Scoped Objects

So far we've been working with instances: instance variables, instance methods, et cetera, but many times a class (rather than an instance of that class) needs to maintain a state or provide a method that is not tied to an instance. This is where class constants, variables, and methods enter the ball game.

Class constants are handy little mechanisms that allow you to place values into the class scope that will not be changed (unlike variables which may and probably will change). To create a class constant, you simply place the constant name and value into the class definition:

```
class Boogeyman
  MY_BOSS = 'Mr. Boogeyman'
end
```

Now every method in class `Boogeyman` (both instance and class scoped) has access to the value `MY_BOSS`.

To create class variables, place two at symbols (`@@`) before the name of a variable; they operate nearly identical to instance variables except their state lives in the class rather than a particular object. For example, the Boogeyman has asked that we have a way to get the name of the newest denizen he has released and where he is. We can provide this with a class variable:

```
class Boogeyman
  # We'll redefine initialize
  def initialize(name, location)
    @name = name
    @location = location

    @@latest = @name
    @@location = @location
    puts "Yes, master?"
  end
end
```

```
monster1 = Boogeyman.new("Macabre Mac", "Seattle, WA")
→  Yes, master?
monster2 = Boogeyman.new("Gory Gary", "Reston, WV")
→  Yes, master?
puts Boogeyman.latest
→  Gory Gary
```

As the example shows, you access a class variable by using the class name followed by a dot and the variable name; much like class constants, you can access this value from either class scoped or instance scoped methods. Notice that because `@@location`, `@location`, and `location` are all scoped differently, they can all use the same name without causing a problem. While this isn't recommended (having three variables with the same name is likely to drive you batty in less contrived, more real world situations), it is possible (and occasionally useful).

Class methods are methods that are provided by a class (not an instance) that may not particularly need to be to an instance. This feature is useful if, for example, you wanted to provide a prefabricated instance of a class (e.g. a method named `man` for a class named `Person` might provide an instance with the `gender` field set). The Boogeyman has requested that we have a class method to output the latest robot's name since he's way too lazy to delve into the code or use a Ruby console to find out for himself (he is retired, you know). So let's provide:

```
class Boogeyman
  def Boogeyman.latest_monster
    puts "The latest monster is #{@@latest}."
    puts "He is in #{@@location}."
  end
end

Boogeyman.latest_monster
→  The latest monster is Gory Gary.
→  He is in Reston, WV.
```

Because `latest_monster` is a class method, it only has access to variables within its scope (i.e. class variables); this means it cannot access instance variables at any time unless they are passed into the method as a parameter or referenced in a class variable. Unlike other class scoped variables, class methods are not visible to instance scoped objects and methods; this means that you must call a class method using its full call (i.e. you wouldn't be able to simply type `latest_monster` like you can `MY_BOSS` or `@@latest`; you would be required to call it using `Boogeyman.latest_monster`).

MODULES

Perhaps sometime you'll need to organize a lot of code. I mean a *lot*. Like the population of China a lot. Maybe that code isn't simply one class or it isn't necessarily all perfectly related; maybe it's got some issues; maybe it's still angry about that argument it had with grumpy_butt.rb last week; in any event, you have been charged to group it together for the sake of reuse and organization. Normally in something like C or PHP you might simply stick this code in a file and include it wherever you

need it, but what if you have two methods or constants that are named the same? Let's say you were creating a panda fighting game. You have a constant `DEATH` in `const_values.rb` to represent the amount of life that would constitute a death in the game, but you also have a constant `DEATH` in `fighter_values.rb` to hold an array of values dealing with the player character named Death. Both pieces need to be included, but there's a name conflict. You would rather maintain a simple naming scheme for constants rather than concocting some silly name like `DEATH_VALUES_BECAUSE_I_CANT_INCLUDE_IT_WITHOUT_A_HUGE_NAME`. This is where modules are rather handy; they allow you to group constants and methods together logically into groups called namespaces, groups which organize modules and the like in such a way as to avoid ambiguity and promote logical organization of code. Namespaces allow you to write larger groups of reusable code without the danger of stomping on other code outside of the namespace. This means there could be a namespace `FighterValues` and a namespace `ConstantValues` to maintain their respective values.

Creating Modules

The syntax for creating a module is very similar to the syntax for creating a class; you place the keyword `module` followed by the module name; then on the subsequent lines you enter the methods and classes which should resign in this module followed by the `end` keyword. Let's look at our example from before:

```
module FighterValues
  BAMBOO_HEAD = { 'life' => 120, 'hit' => 9 }
  DEATH = { 'life' => 90, 'hit' => 13 }
  KOALA = { 'life' => 100, 'hit' => 10 }
  CHUCK_NORRIS = { 'life' => 60000, 'hit' => 99999999 }

  def chuck_fact
    puts "Chuck Norris' tears can cure cancer..."
    puts "Too bad he never cries."
  end
end

module ConstantValues
  DEATH = -5   # Pandas can live PAST DEATH.
  EASY_HANDICAP = 10
  MEDIUM_HANDICAP = 25
  HARD_HANDICAP = 50
end

puts FighterValues::DEATH
→  {'life'=>90,'hit'=>13}

puts ConstantValues::DEATH
→  -5
```

Now both values can be used and co-exist in a friendly environment. I'm sure you're thinking, "Why not just use a class?" I asked myself that when I first saw this construct; the only reason I saw was that, for the sake of design and proper software engineering, you shouldn't put things in a class that don't really go together so the module was a good excuse to break that rule without breaking it.

But then I saw the coolest part.

Modules have a mechanism that allow for what's called a mixin, code that is "mixed into" a class as if it is part of its original code. Think of it as inheritance, except better. As noted earlier, a class in Ruby can only inherit from one class at a time. To inherit from another class you would have to create some sort of chain of inheritance that would allow you to do "multiple inheritance" (not really but that's as close as you can get in Ruby). Mixins eliminate the need for that. You could create a class, inherit from another class, and mix in as many modules as you need. This feature is especially great if the code that you need to mixin needs to only be mixed in (i.e. it won't ever be used by itself). Let's look at a contrived example:

```
module Movement
  def run
    puts "I'm running!"
  end

  def walk
    puts "I'm walking a bit briskly!"
  end

  def crawl
    puts "I'm so slowwww!"
  end
end

class Man
  include Movement

  def jump
    puts "I'm bipedal and I can jump like a fool!"
  end
end

class Sloth
  include Movement

  def flop
    puts "It's all a lie...all I can do is flop around."
  end
end

mister_man = Man.new
mister_man.run
→  I'm running!

mister_slothy = Sloth.new
mister_slothy.flop
→  It's all a lie...all I can do is flop around.
```

As you can see, this mechanism is very similar to inheritance in that you can use all of the mixin's code. To use a mixin, simply define a module and then use the `include` keyword followed by the module's name (note I said *module;* the include keyword has nothing to do with files or libraries like in PHP or C++); from then on the class has access to all of that module's constants and methods. It's obvious that this example doesn't do this mechanism justice (i.e. it doesn't demonstrate the usage of module constants in a mixin, it doesn't do much with the host class, etc.), but this is merely meant to be an introductory example in hopes that you will experiment and read further. As you will learn, the magic really happens when the class actually interacts

with the mixin (as with some of the Ruby built-in mixins such as `Singleton` or `Comparable` which greatly extend the functionality of your class or some of the Rails mixins), but that's for a more advanced look at the subject.

You may be thinking that this sounds great now, but do be careful. Mixins are awesome as long as they are written well, but if the developer doesn't pay attention and be careful about naming they can create havoc in your application. For example, let's say you have a constant called `PI` which holds Pi to the 72nd digit (which you typed out manually because that's the amount of precision you need), but you mix in a trigonometry library written by Billy McDoofus which has a constant named `PI` which is only Pi to the 5th digit. You need that precision, but since Billy McDoofus is an idiot, the mixed in library's constant will override your constant. It's best to be sure that your naming scheme is unique (possibly even including the module name or your name) so as to not stomp on others' code.

You may be thinking right now, "That jerk left out methods in modules! What about methods!? I'm going to kill his cat and take my money back! Are my methods simply going to suffer the same terrible fate?" You would be justified in your furor, but relent, good friend, for methods do not suffer the same fate. When you call a method, Ruby will first look to the host class (that is, the class being mixed into) and then to its mixins (and then to its superclasses and its mixins and so on); this behavior is the exact opposite of constants (and no I don't know why that is). This could be a blessing or a curse (i.e. you may *want* the method to override yours), but generally this is the safest functionality for it.

FILES

As your application gets bigger and bigger, you surely won't want all of your code living in one huge 5MB file. Chopping code up into files is one of the oldest and easiest ways to segment your code. I saved it for last because I don't want your answer to code segmentation to always be "Stick it in a file!" when Ruby offers more (and better suited) options than that. I think PHP programmers especially get stuck in this rut of including code all over the place, creating a jungle of files that only the machete of the delete command can navigate, but I digress. To include a file in Ruby, you have two options: `load` and and its more elegant cousin `require`. The difference is that the `load` keyword includes a source code file unconditionally while `require` will only include it once (which means no duplication of variables and methods):

```
load "libraries/myfile.rb"
require "/home/myaccount/code/libraries/myotherfile.rb"
```

Both keywords accept either relative or absolute paths; if Ruby identifies it as a relative path, it will search for the file in the current path (stored in `$:` for those who are curious). The require statement is also great in that you can use it in conditionals, loops, and other constructs or use variables in its paths (you can not with load). Keep in mind though that local variables in included files do not trickle into the context they are included in (this is contrary to PHP's and C/C++'s behavior); these variables are locked into the context that they are written in.

This Chapter

You learned how to break up your code into more logical and usable pieces. You learned...

- how to segment your code using blocks, methods, classes, modules, and files.
- how variable scoping works and how it can benefit you.
- how to make your own classes and objects and how to make changes to others.
- that modules allow you to mix in code to classes and extend them.

3

Hustle and flow (control).

Flow control is essential to every application (unless of course your application will always have one course of action and you don't mind copying and pasting a lot of code). Flow control constructs allow you to branch (i.e. execute a different flow of code) based on conditions, run the same branch numerous times, or bail when one of the conditions is wonky. There are two "general" flow control mechanisms: conditionals and loops. I'll address them separately lest they end up in some sort of extremely confusing mental *cesspool*

CONDITIONALS

Conditional constructs are those constructs that give you and your users choices; they allow your program to branch and execute different code based on conditions within the application. They compare values to determine truth and based on that truth they determine which (if any) branch of code should execute. Think of them as the traffic lights of the Ruby programming world.

The `if` statement

The most fundamental conditional statement available in Ruby is the `if` statement: it simply compares values to determine truth or untruth. If it is true, then the code contained within the main branch is executed; if it is not true, either the code in the `else` clause is executed or, if no `else` block is present, no code in the block is executed and the application continues on.

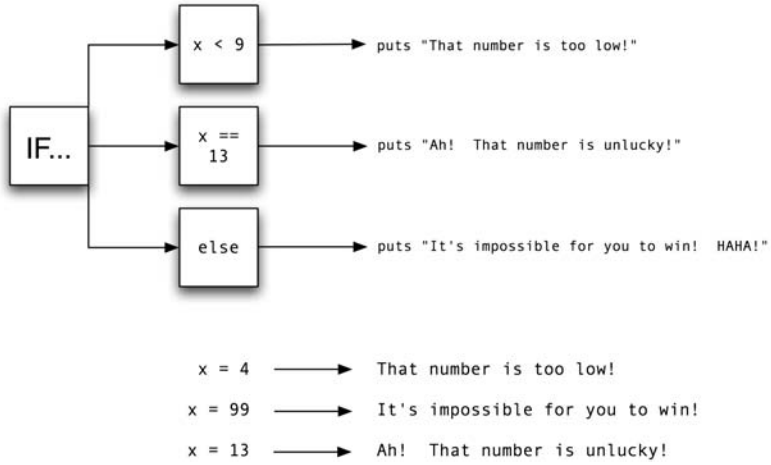


Figure 7: If I were a fish in the sea, I'd wiggle my tail and I'd giggle with glee...

The result of all `if` statements is either true or false regardless of how many conditions they test. Let's look at an example:

```
if not (true != false) && (true != true):
  puts "Whoa!  Alternate dimension!"
elsif (false === 'false') then
  puts "Type conversion?  I think not..."
elsif (true == 1) or (3 == false)
  puts "Booleans != Numbers"
else
  puts "I guess we're still OK!"
end

→ I guess we're still OK!
```

There's a lot to look at here, so let's take it a piece at a time. An `if` statement is constructed by placing the `if` keyword on a line followed by a conditional comparing values equality, inequality, etc. The `if` statement itself opens the conditional block (i.e. `if` is the initiator of a `begin/end` block); you can create an `if` block simply with the `if` statement, its block of code, and the `end` keyword if that's all you need. This example, though, demonstrates all of the possible constructs for an `if` statement. The `elsif` and `else` keywords allow you to branch if the conditions in the `if` statement are not met; `elsif` allows you to present another conditional test while `else` is simply a fail safe if none of the conditions are met. This means you don't need to create 10 `if` blocks when you can simply wrap them into 1 `if` block with a lot of `elsif` statements (with the safety net of the `else` statement to boot).

The second thing to notice are the operators used in the conditionals. To test for equality, you should use the == operator (that's two equals signs, not one; if you use one it will usually result in `true` because you would be assigning a value rather than checking for equality). To test for inequality, use the `!=` operator. Rather than entering these into a big fat paragraph, here is a table of operators usable by Ruby in conditionals:

CONDITIONAL OPERATORS IN RUBY	
`a == b`	Is true if the value of a to the value of b.
`a === b`	Is true if the value of a to the value of b and they are the same type.
`a != b` `a <> b`	Is true if a is not equal to b.
`a !== b`	Is true if a is not equal to b or they are not the same type.
`a < b`	Is true if a is less than b.
`a > b`	Is true if a is greater than b.
`a <= b`	Is true if a is less than or equal to b.
`a >= b`	Is true if a is greater than or equal to b.

The third thing to notice about the example are the logical operators; these are special operators used to chain conditionals together or to evaluate special cases. Ruby supports two forms of the operators: C-style and stringified. The C-style operators are like the one in the first example; `&&` is the same as "`and`," the `||` operator means "`or`," and so on. The stringified operators are the same as their symbolic brethren except that they are strings: `and`, `or`, etc. Here is a table with these operators in both forms and their usage:

CONDITIONAL LINK OPERATORS IN RUBY		
`&&`	`and`	Conditional is true if linked statements are both true.
`||`	`or`	Conditional is true if either of the linked statements are true.
`!`	`not`	Conditional is true if attached statement is false (i.e. it works like !=).

The Ternary Operator Now that you know the long hand way to do `if` statements, I would like to show you a bit of a shortcut. The ternary operator (Random piece of trivia: the ternary operator is named simply for having three parts) allows you to execute an if statement without creating an entire `if` block. For example, here is an `if` block followed by its ternary equivalent:

```
if ('Yes' == 'No') then
  puts "Wrong!"
else
  puts "Yeah, baby!"
end
→ Yeah baby!

('Yes' == 'No') ? (puts "Wrong!") : puts ("Yeah, baby!")
→ Yeah baby!
```

The ternary operator consists of the question mark after a conditional (which uses the same operators as an `if` statement) and a colon between the two possible branches of code. The parentheses are not essential in every case, but do enhance readability and

allow you to span the branches multiple lines (i.e. open the parentheses on one line enter each line of code and close the parentheses on another line). This multi-line behavior is not recommended because it slightly mangles the code and really doesn't save you any typing at all compared to an `if` statement.

Unless In Ruby the `unless` conditional statement operates as a "reverse" `if`. I say reverse because in an `if`, the main branch executes only if the conditional is true; in an `unless` statement, the main branch only executes if the provided condition is false. For example:

```
unless ("I am true." == "I am true."):
  puts "Something wonky!"
else
  puts "All is alright!"
end

→ All is alright!
```

Since "I am true." does indeed equal "I am true.", the main branch is not executed but the `else` branch is. This structure doesn't provide anything over an `if` statement except to enhance readability of the code (e.g. it's much easier and casual to say "Do this unless this" rather than "Do this unless this is not this" and to avoid writing a lot of negative `if`'s; keeping positive chi in your code is very important to the emotional health of Ruby).

Statement Modifiers The `if` statement also offers a nifty piece of syntactic sugar called a modifier; this construct allows you to tag an `if` statement on the end of a statement and have it decide whether or not the statement is attached to should be executed. For example:

```
thelma_louise = 13
puts "She's less than 15 alright!" if thelma_louise < 15
puts "She ain't more than 12, though!" if thelma_louise < 12

→ She's less than 15 alright!
```

As you can see, if you simply tag on an `if` statement to end of a line, it will evaluate the `if` statement first and execute the code if it finds that the if statement is true. This construct replaces code like this:

```
if thelma_louise < 15:
  puts "She's less than 15 alright!"
end
```

with a far more concise and elegant solution. *Rubylicious*. This construct can be tagged on to `begin/end` blocks also. For example:

```
begin
  puts "It's so true!"
end if (true == true)
→ It's so true!
```

This is a very elegant solution for controlling big blocks of code rather than having a huge, open `if` statement that spans 500 lines (especially if there's that one end keyword you forgot to drop in).

The `case` Statement

If you haven't noticed, I'm all about having the neatest code in the least strokes (who isn't nowadays?). It gets really irritating really fast when you are testing the same value over and over again in a series of `if` and `elsif` statements that quickly add up to a lot of sloppy looking code. It's a good thing that there is a way to put all those together into one syntactically sugarlicious block called a `case` statement. A `case` statement comes in two forms in Ruby. The first form is the "standard" way `case` statements operate: you have a "target" variable and each case is a test of values against that variable. For example:

```
the_tao = 1234

case the_tao
  when 666: puts "That's such a lie!"
  when 1337
    puts "Liarrrr!"
  when 32767 then
    puts "Whatevurrrr!"
  else
    puts "You are harmonized with the Tao."
end

→ You are harmonized with the Tao.
```

This form of the `case` statement simply tests the target value against the values in the cases; cases are formed using the `when` keyword followed by the test which is followed either by the word `then` or a colon (both of which are non-essential if the case starts on the next line); the `when` keyword segments the cases so no "`end`" keyword is needed except to close the `case` block. Each case is tested starting from the top; when a match is found the proper branch is executed and the block exits. If nothing matches then the else clause is executed (just as in `if` blocks).

The second form of the case statement adds more flexibility by allowing you to use conditionals like `if` statements and removing the target. Even though the target is removed, you can still use them in the same way you would the first form:

```
enlightenment = 42

case
when enlightenment > 60:
  puts "You are too hasty, grasshopper."
when enlightenment < 40 or enlightenment == nil:
  puts "You are like the sloth, my friend.  Diligence is key!"
when enlightenment == 42:
  puts "Hello, Enlightened One."
else
  puts "Yeah, not quite, pal.  Maybe next time."
end

→ Hello, Enlightened One.
```

As you can see, this form can still be used to compare a single variable, but allows for more robust comparisons than the first form. Each comparison is checked, starting with the top case, first comparison, working to the right and down; when a case is matched (i.e. all conditions needed for a case to be true are met including chained conditions) that branch is executed and the block exits.

A commonality between the two forms is their lack of "fall through"; in many programming languages, `case` blocks require some sort of keyword (usually `break` or something similar) which tells the application to keep checking the other conditionals after that block is finished. For example, look at this C-style syntax snippet:

```
int my_number = 42;

switch(my_number) {

  case 41:
    printf("A little higher...");
    break;

  case 42:
    printf("You have found the answer!\n");

  default:
    printf("And have fallen into a hole in the C compiler!\n");
    break;

}

→ You have found the answer!
→ And have fallen into a hole in the C compiler!
```

If it's not evident from the code, the `break` keyword will exit the `switch` block if it is encountered. This keyword is not needed in Ruby since it doesn't have fall through, but since C-style languages do, the `default` block (like Ruby's `else` in case statements) gets executed because there was no break in the block that evaluated to true (i.e. case 42). As helpful as the fall through mechanism is, it can really be quite annoying when the lack of one `break` statement causes your entire application to come to a screeching halt.

LOOPS

Loops save your fingers and brain a lot of work by executing the same or slightly different code numerous times, responding to conditions as need be. Loops are essentially a sequence of statements which you enter once but which may be carried out several times in succession a finite number of times, maybe once for each member of a collection or until some condition is met. We'll cover both of these types in the coming sections.

Conditional Loops

Conditional loops execute based on a provided conditional statement (i.e. the same conditionals used in `case`, `if`, and `unless` blocks); these loops come in two forms: `while` and `until`. A `while` loop will execute only while a conditional is true:

```
x = 0
while(x < 10):
  print x.to_s # print is like puts without a \n at the end
  x += 1
end

⟶ 0123456789
```

Each iteration, the conditional is checked; if it evaluates to `true`, the branch executes; otherwise, the block exits.

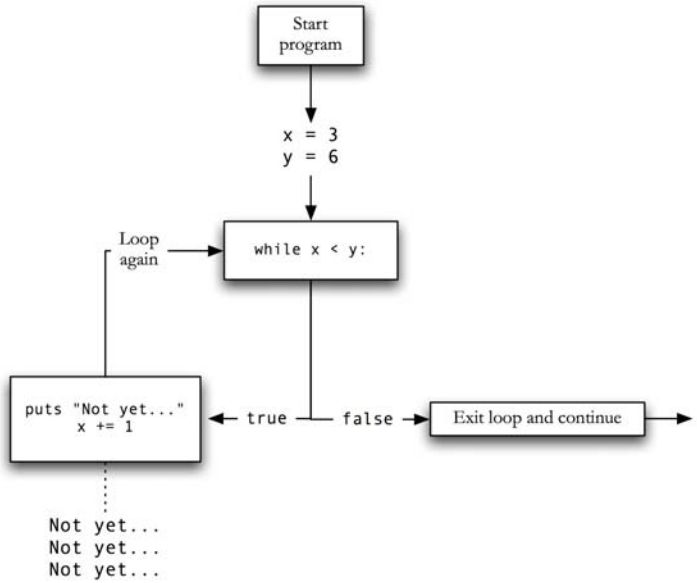


Figure 8: The while loop: setting the standard for looping since 1973.

In other words, it's as if you put an `if` statement at the top of the loop and the main branch is executed until the `if` statement becomes false. The `until` loop has the same relationship as that of the `unless` statement to the `if` statement:

```
x = 0
until(x >= 10)
  print x.to_s
  x += 1
end

⟶ 0123456789
```

Each iteration of the loop, the conditional is checked just like a `while` loop, except in an `until` loop, the branch is executed if the conditional evaluates to `false`.

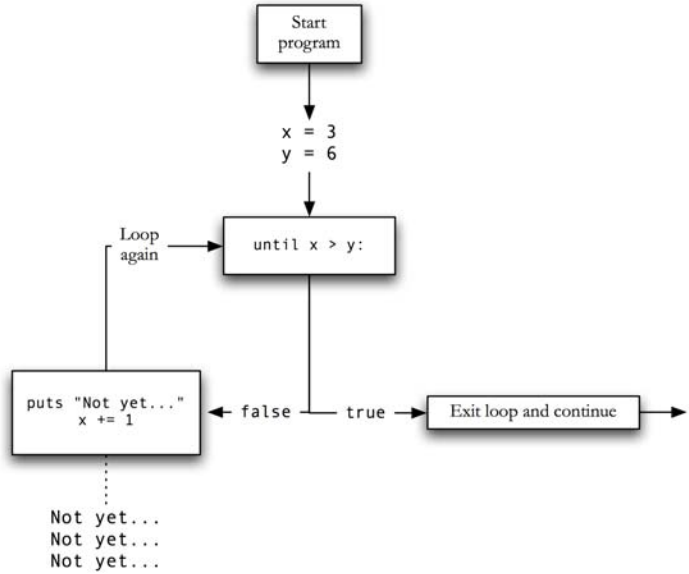


Figure 9: I think the until loop gets jealous of all the attention that people give while.

Much like the `unless` statement, if the conditional provided is true then the main branch is skipped, and, in the case of the `until` loop, the block exits.

These loops are great if you are doing some sequential operation (i.e. doing ten repetitions of the same task, outputting ordered data, or, like the examples, counting), but they have their pitfalls. If the conditional doesn't meet the requirements for the branch to be executed (i.e. true for `while` and false for `until`) before the first execution, the loop never runs; be aware of this possibility if you are manipulating variables in the loop that exist outside of the loop whose manipulated values are essential to your application's execution after the loop. This condition can and will cause problems, so make sure that you do a veritable assortment of verifications on your variables (more on this later).

Another pitfall that may rear its ugly yet somehow mildly bearable head is that the conditional may never reach a state where the loop will break (i.e. the condition in your while loop may never reach false) and thus put your program into an infinite loop and lead it and all of Creation into oblivion. It's been said that the universe actually implodes and compresses to the size of a single Pocky if this happens, but I can't verify that. So, how do you prevent that from happening? For starters, make sure that you use as flexible conditionals as possible (e.g. the >= operator is probably a better idea than == if you're counting up simply because you never if by some freak accident it might go over your intended value; this isn't the best idea in all scenarios, but it should be considered) and make sure you interact with the conditional value (make sure you actually do something to affect the conditional; you'd be surprised how many people forget that).

If you're a hardcore *über-1337* programmer, you may be saying to yourself, "Yo, fool! I ain't gots to be worrying about that because I be using those fo' loops fo' rizzle!" That may be true, and you'd be keeping it real in about any other language except Ruby. That's right: Ruby doesn't offer a "for loop" in the traditional sense, but it does offer one that uses the keyword `for`...

Iterating Loops and Blocks

In my life as a developer, I've seen some rather preposterous pieces of code; everything from text files used as high traffic databases all the way to people reinventing the wheel at least 17 times during the course of their application development (news flash to all developers, programmers, and hackers out there: most languages have date/time manipulation routines, which means you don't need to write your own for your application and/or module. Thank you.). These people are honestly just making it far more difficult for themselves and everyone else; that's sort of the way I feel when I see people using conditional loops to grok a collection in languages that plainly offer something better. For example,

```
my_array = [13, 1, 4, 5, 29]

i = 0

while (i < my_array.length)
  print my_array[i].to_s + " "
  i += 1
end

→ 13 1 4 5 29
```

would work, but is not the best answer (especially in Ruby). Ruby offers a couple of ways to iterate a collection safely, easily, and efficiently: iterating loops. The first of these is the `for` loop. The `for` loop in Ruby, as noted before, isn't like the `for` loop in, say, C or its derivatives, but I feel it's even infinitely more useful.

The `for` loop in Ruby behaves very similarly (almost identically) to the `for` loop in Python: a collection is provided, and the `for` loop provides a local variable to hold each item as it iterates.

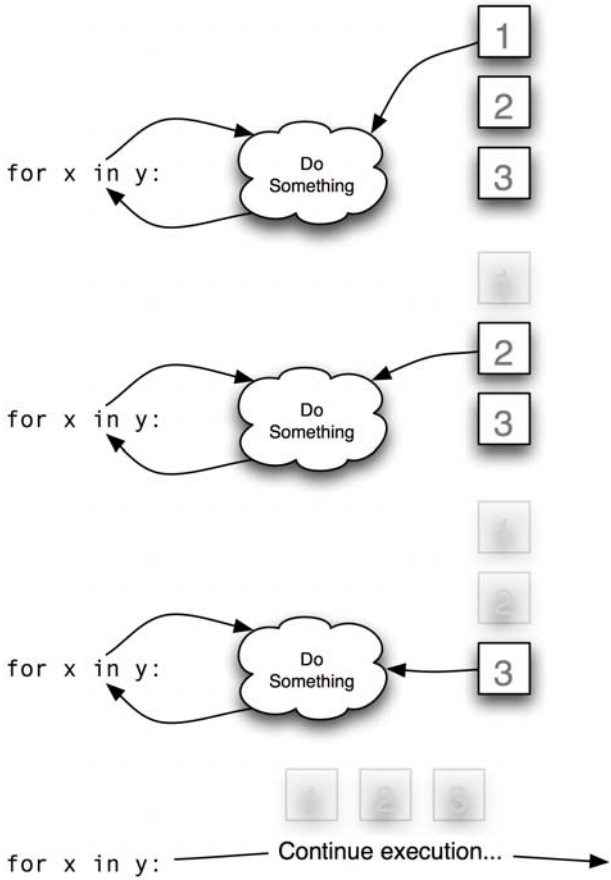


Figure 10: Iterators make things much easier and safer than conditional loops when using collections.

Let's look at a code example; we'll create an array of even numbers, and then print them out using an iterator.

```
my_evens = [2,4,6,8,10,12,14]

for my_int in my_evens
  print my_int.to_s + ", "
end

→ 2, 4, 5, 6, 8, 10, 12, 14,
```

This is like saying, "I want to create a local variable `my_int` for each object in `my_evens` and do something with it." Every time the loop iterates, the next value in `my_evens` is copied into the variable `my_int`; this variable allows you to manipulate the item in the collection easily (and, since the original object is copied into that variable, you can manipulate it without risk of bit twiddling in the original collection).

The `for` loop is really just salacious syntactic sugar for an iterator; an iterator is simply a block of code that iterates over a collection provided by the each method of a class. For example, let's look at the above example in the iterator form:

```
my_evens = [2,4,6,8,10,12,14]

my_evens.each do |my_int|
  print my_int.to_s + ", "
end

→ 2, 4, 5, 6, 8, 10, 12, 14
```

An iterator is formed by creating a `begin/end` block (discussed above) but using `do/end` as the initiator; you are essentially creating a special type of `begin/end` block that will be iterated for each item in the collection. This type of loop naturally guards against the variable discord that you may encounter by incrementing an index using something like a `while` loop (i.e. you go over the value you want if using the == operator, your value never gets above the needed value if using a < operator, etc.) while also providing a more natural way to access the value you wanted. Unfortunately this convenience comes at the price of flexibility. In a `while` or `until` loop you can alter how quickly your conditional reaches the state that allows the loop to break (also called step) by adding more than 1 to the value you're keeping track or something like that; this ability is severely limited in an iterator loop (i.e. it doesn't exist unless you want to use the `next` method somehow to make it happen). Keep this in mind when you decide which type of loop to use; it can greatly affect the performance and stability of your application.

Statement Modifiers

Loops offer a construct very similar to the `if` statement's modifier. Loops are great ways to save time and code, but they aren't very "natural"; I mean when repeating a task (such as banging your head into a mirror out of sheer frustration) you typically don't think in the form of a loop construct.

```
while(self.conscious == true)
  self.head_bang("mirror")
end
```

That's simply not how our mind usually works, and fortunately, Ruby, in all of its readable glory, has decided to bless us with the loop statement modifier construct. For example:

```
self.head_bang while self.conscious == true
```

This construct is not only more natural to read and say, but it is also more concise; you still have all the same functionality but in much less code and hassle. This construct works with both `while` and `until` loops (but not any of the iterating loops); it also works with `begin/end` blocks which allows you to do post-test loops (i.e. loops that always execute at least once before checking the conditional as opposed to the pre-test loops that we've been using). For example, if you wanted to output the English translation of "Tora! Tora! Tora!" you could write:

```
counter = 0
```

```
begin
  print translate("Tora! ")
  counter += 1
end until counter == 3

→ Briefcase! Briefcase! Briefcase!
```

In this setup, the block will always execute at least once regardless of the truth of the attached conditional (unlike if you simply use the `while` or `until` loop). Do keep in mind when using this construct that you are still, at its core, using the `while` and `until` loop, which means that it is still susceptible to the same pitfalls and problems with variables. Make sure to test your application thoroughly for any potential problems related to this.

Controlling Loops

Does it seem that loops have your application in a strangle hold? They do seem to be hulking, domineering, unstoppable behemoths that stop only when they darn well feel like it. Well, actually they probably just seem like really useful constructs that are hard to control without artificial bit twiddling in the conditional. For example:

```
my_x = 115
my_y = 40
temp = 0

while(my_x < 150)
  if (my_x % my_y) == 0: # if the quotient is even
    temp = my_x
    my_x = 151
  else
    my_x += 1
  end
  puts my_x
end

my_x = temp
puts my_x
→ 120
```

That's a bit dangerous if you ask me (and if you're reading this, you just might); artificially altering the conditional value can lead to some craziness in your application (i.e. accidentally skipped code, the variable being used outside of the loop without the temp value being stored back into it, etc.). In Ruby, there are various keywords that allow you control the flow of loops. The `next` keyword allows you to skip the current iteration, while the `break` keyword allows you to exit the current loop completely. Let's look at our previous example again:

```
my_x = 115
my_y = 40

while(my_x < 150)
  my_x += 1
  puts my_x
  if (my_x % my_y) == 0: # if the quotient is even
    break
  else
    next
```

```
  end
end

puts my_x
→ 120
```

The usage of the `next` keyword in this example is rather inordinate (i.e. I should have simply let the loop iterate again rather than forcing it), but I wanted to demonstrate how the `next` keyword works. This loop works just as before, except more concise and less bloated.

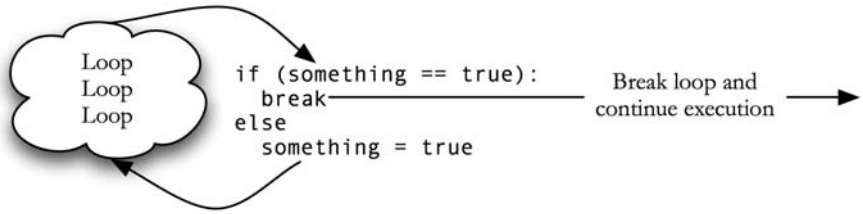


Figure 11: Break. I always get a great degree of satisfaction from breaking a loop. Take that sucker!

The `break` keyword breaks the loop and continues on to the code after the loop just as if the conditional had been met. The `next` keyword skips past all remaining code (which in this example wasn't much) to the end of the loop and continues on to the next iteration. But, wait! It gets fancier:

```
my_x = 115
my_y = 40

while(my_x < 150)
  my_x += 1
  puts my_x
  break if (my_x % my_y) == 0
  next
end

puts my_x
→ 120
```

You can use a conditional modifier with the `break` or `next` keywords! And, yes, the `next` keyword isn't necessary, but I wanted to demonstrate two things. Firstly, the keyword's usage (I know you probably understand it by now, but another example never hurt anybody!). Secondly, all code after `break` is skipped. The loop does not iterate again because the conditional attached to `break` was satisfied.

EXCEPTIONS

OK, so you're zipping along on an application, but all of a sudden you get this crazy error and you don't know how to handle it. Maybe your users are 13 year old social rejects with more pimples than friends, and they feel the need to type in "YO d00D I"M 1337!!11" in a field in your application that is supposed to be all numeric, effectively crashing your application. Shall you just let your application crash and burn? Shall its cinders smolder forever? Shall you never see glory because of

"TO01337ANdY" from Bumpkinville, IA? Never fear, young neophyte! Ruby has you covered!

Exceptions provide a handy way to deal with errors and other problems in your application in a way that is uniform and easy to implement. All exceptions derive from the base class `Exception`; Ruby provides a number of built-in exceptions for you to use (or for your application to handle). Here is a table of the current available built-in exceptions:

Handling Exceptions

To handle an exception when one is raised, you must create a `rescue` block. Let's take a look at an example and then pick it apart like voracious vultures in search of delectable morsels of *delight* (or like some guys who just really want to know how to program in Ruby):

```
def method_of_doom
  my_string = "I sense impending doom."
  my_string.ah_ha_i_called_a_nonexistent_method
end

method_of_doom
→ ! NoMethodError
```

Now, to handle this exception properly:

```
def method_of_doom
  my_string = "I sense impending doom."
  my_string.ah_ha_i_called_a_nonexistent_method
rescue Exception:
  puts "Uhh...there's a problem with that there method."
end

method_of_doom
→ Uhh...there's a problem with that there method.
```

To create a `rescue` block, you simply place the `rescue` keyword at the beginning of a line followed by the exception class you would like to handle (Ruby offers a number of built-in exceptions for you to use; check the Ruby documentation for a full list). I used `Exception` because it catches any and all exceptions, but you can specify numerous `rescue` blocks to handle different, more specific types of exceptions and/or numerous exception types can be handled by a single `rescue` block:

```
def method_of_doom
  my_string = "I sense impending doom."
  my_string.ah_ha_i_called_a_nonexistent_method
rescue NoMethodError:
  puts "You're missing that method, fool!"
rescue Exception:
  puts "Uhh...there's a problem with that there method."
end

method_of_doom
→ You're missing that method, fool!
```

If you specify multiple `rescue` blocks, Ruby follows the first one it encounters that is able to handle an exception of same type as the raised exception. If we had specified the `Exception` block first as opposed to the `NoMethodError` block, we would have gotten the same output as the first example. If no usable `rescue` block is found in the current context (i.e. current block of code, current method, etc.), Ruby works its way up the call stack to see if it can find a suitable rescue block (i.e. it works its way from the offending method up to the method called it to the method that called it and so on). If no suitable `rescue` block is found before Ruby takes its search to the main application method, then the application thread exits and a message is shown that is something along the lines of, "I pity the fool that be missin' an exception handler!" followed by a series of loud crashes and breaking of wind.

A `rescue` block can specify a variable name when it is created in order to hold a more detailed explanation of the error. Again using our previous example:

```
def method_of_doom
  my_string = "I sense impending doom."
  my_string.ah_ha_i_called_a_nonexistent_method
rescue NoMethodError => e:
  puts "PROBLEM: " + e.to_s
rescue Exception:
  puts "Uhh...there's a problem with that there method."
end

method_of_doom
→ PROBLEM: undefined method [and so on...]
```

Leveraging this ability makes it far easier to know exactly what's going on, where, and sometimes how to fix it.

A `rescue` block also provides a few other features that aid in making sure that even if your application does choke, it will still hopefully run as smoothly as possible. The first of these features is the `else` caluse that you can tag on to a `rescue` block. This block will execute if there are no exceptions raised. For example:

```
def no_problemo
  x = 0
  x += 19
rescue Exception
  puts "Oh noes!"
else
  puts "All clear!"
end

no_problemo
→ All clear!
```

This is a useful feature, but be careful what you put in there. The `rescue` block in the enclosing code won't catch any exceptions raised in the `else` clause, so you may need to catch them later up the call stack or relegate the `else` block to menial tasks to avoid the risk of causing worse problems.

The second feature offered by `rescue` blocks is the `ensure` clause; this clause holds code that is always executed (i.e. regardless of the presence exceptions or not). For example:

```
def dance_a_jig
  "I'm a dancin'!"
  "Do si do!"
  rebel_yell = "yee haw!".upcase!
rescue Exception
  print "I fell down, dang it!"
else
  print rebel_yell
ensure
  print "  That's all folks!"
end

dance_a_jig
→ YEE HAW!  That's all folks!
```

The `ensure` clause is always executed no matter what; this construct is very useful for closing files that you have been reading from or writing to, closing open network connections, or making sure all your resource handles have been cleaned up. If you put these sorts of things in `ensure` clauses, they will always get done and cut down on problems you may have with resource access if your application crashes.

Rescue Statement Modifier Much like conditionals and loops, `rescue` blocks can be used as statement modifiers. For example:

```
not_an_object.do_something rescue puts "Crash!"
→ Crash!
```

Note you can't specify what sort of exception to rescue from, but that is better left to "formal" `rescue` blocks anyhow. You can assign values using this construct also:

```
my_value = not_an_object.give_a_value rescue "Burn!"
puts my_value
→ Burn!
```

This pitiful example doesn't show a real world case of course, but this construct is useful if a small adjustment in value can correct most any exception (a rare but possible case).

Retry Sometimes you just need a do-over. That loop didn't do well for you, or maybe that variable still wasn't clear on his motivation for this scene. Either way, you need to be able to redo part of your code in hopes that they will simply go better next time. This is where the `retry` keyword comes in; let's say you were building a fictional web browser:

```
def make_request
  if (@http11)
    self.send('HTTP/1.1')
  else
    self.send('HTTP/1.0')
  end
rescue ProtocolError
  @http11 = false
```

```
    retry
  end
```

You send the HTTP 1.1 headers, but the server on the other end doesn't like that, so it vomits a `ProtocolError`. Instead of rolling over and dying, you disable HTTP 1.1, retry the block, and your application is smart enough to switch to HTTP 1.0 instead. Fancy. Using `rescue` and `retry`, you can make an attempt to fix any errors that may cause exceptions and retry the block again. Keep a close watch on this though; it can cause some serious problems if the problem is never fixed the same block of code is looped over again and again because your application is retrying it.

Raising Exceptions

So now that you know how to handle exceptions, I think it's a fine time to put you out to pasture with your own exceptions. Raising your own exceptions is important if there are problems that will arise that won't necessarily cause a problem with Ruby itself. For example, let's take a look at a method that may exist in class `Person`:

```
def define_gender(gender)
  if (gender.upcase != 'FEMALE') && (gender.upcase != 'MALE')
    raise "You specified something wonky for gender!"
  end
end

my_guy = Person.new
my_guy.define_gender("nobody knows")
→ ! RuntimeError ("You specified something wonky for gender!")
```

Even though it won't cause a problem with Ruby itself, you obviously don't want someone to specify something abnormal for gender (well, maybe you do, but that's just weird). If such a condition arises, you can drop the `raise` keyword on a line followed by a message and Ruby will raise a new `RuntimeError` and set its message to the one you provide. The `raise` keyword (or its uncouth cousin from Wales, the `fail` keyword) can be called a number of ways:

```
raise "I crashed!  This message should be more informative!"
raise
raise NoMethodError, "That method ain\'t here!", caller
```

The first form you are already familiar with (i.e. provide a message to a new `RuntimeError`); the second form will re-raise the current exception so it can be passed further up the call stack or raise a new `RuntimeError` (with no message) if there is no exception. The last form is one you should use most constantly because it allows you to specify an exception type, a message, and a stack trace object (which is usually just `caller`, a reference to the `Kernel.caller` method). Good software practice says that you should be as specific as possible with your exceptions; instead of just throwing `RuntimeErrors` all the time, try to throw a `TypeError` if the provided object isn't the right type or your own exception type to match your needs (e.g. it would have been better to have raised an `GenderError` or some such in our example rather than a `RuntimeError`).

My Own Exception

So how do you create your own exception types? It's really quite simple in Ruby; let's use our example from before:

```
class GenderError < RuntimeError
  attr :what_they_put

  def initialize(their_input)
    @what_they_put = their_input
  end
end
```

To create a new exception, you simply derive from any of the exception classes (e.g. I derived from `RuntimeError`, but you can derive from `TypeError`, `NoMethodError`, or even `Exception`). So, let's update our code above to use our new, more practical exception:

```
class Person
  def define_gender(gender)
    if (gender.upcase != 'FEMALE') && (gender.upcase != 'MALE')
      raise GenderError.new(gender), "Invalid input!"
    end
  end

  def initialize(gender)
    self.define_gender(gender)
  rescue GenderError => bad
    puts "You gave me some bad input: " + bad.what_they_put
    raise
  end
end

my_guy = Person.new("Who knows?")
→  You gave me some bad input: Who knows?
→ ! GenderError ("Invalid input!")
```

Notice that we raise the exception in `define_gender`, we handle it in `initialize`, and then pass it up the stack. In initialize, we output an attribute held in our new exception class; since exceptions are objects and thus have classes behind them, when you create your own you can add methods and attributes to it as I did to hold the value of the user's input. This is useful if you would like to provide more data to exception handlers or if you would like to provide help in recovering from exceptions by providing methods in the exception.

Throw and Catch

If you're a C# or Java programmer, you might have just gotten excited by the prospect of some familiarity in this area, but don't count your proverbial eggs before they hatch. In Ruby, a `catch` block is given an identifier as an argument; you can then "throw" this identifier in the ensuing code. Ruby will then look up the stack to see where the matching catch is, and if it's found, Ruby will break normal processing and exit the `catch` block. This sounds more confusing than it is, so let's look at an example:

```
princess = DamselInDistress.new
```

```
catch :hes_a_failure do
  # YAY!  Someone's here to save her...
  print "My prince is here!  "

  # OH NO!  The villain has eaten his liver!  He dies!
  princess.is_saved = false

  if (princess.is_saved == true)
    puts "Hooray!"
  else
    puts "Poo!  Not again!"
    throw :hes_a_failure
  end

  puts "I'm going to sleep until the next guy..."

  # Nap...
end

→ My prince is here!  Poo! Not again!
```

A `catch` block is started by placing the `catch` keyword, an identifier, then the `do` keyword on a line; the identifier is used with the `throw` keyword and can either be a symbol or a string. The code is run until a matching `throw` statement (i.e. the throw's identifier matches the catch's identifier) is encountered (if one is encountered) in which case the `catch` block exits without executing any code after the `throw` statement. In the example, you can see this happen in that "My prince is here!" and "Poo! Not again!" are output but "I'm going to sleep until the next guy..." is not; the block exited after the matching throw statement (i.e. `throw :hes_a_failure`) was found. This construct is extremely useful if you need to simply exit the code block if an error occurs or if your code is buried in deeply nested loops and you want to break out of them quickly and easily.

This Chapter

You learned about Ruby's flow control mechanisms. You learned...

- about `if/unless` statements and conditional operators.
- about loops, both conditional and iterating, and how to use the most effectively.
- about exceptions and their usage.

4

The System Beneath...

Recent advances in language libraries have had people ditching Perl and Bash scripts in favor of a more friendly solution like Python or Ruby. This is mostly because you can accomplish the same tasks with less effort and more robustness with these languages (it's also probably because Perl and Bash suck). Thanks to its libraries, Ruby can interact with the system just as well as these more esoteric solutions. Let's take a look at some of the system libraries and functions built into Ruby. You'll be trading in your copy of Perl Cookbook within the hour!

FILESYSTEM INTERACTION

The `File` class in Ruby is very rich compared to other similarly featured languages (i.e. Python). It not only has more methods, but the methods which are comparable are more logically named (e.g. what does `unlink` do in Python? Oh, it deletes? Why not call it that!?). Ruby's `File` class's general power and ease of use compared to many other languages should bring comfort to your heart, much like a warm bowl of soup and classical music can do on a snowy day.

First, let's look at what you can find out about a file. Does it exist? What kind of file is it? Is it a file? Here are a few examples (these assume there is a file named "`textfile.txt`" in the current directory):

```
File.directory?("textfile.txt")      → false
File.file?("textfile.txt")           → true
File.exists?("textfile.txt")         → true
File.size?("textfile.txt")           → 2063
File.extname("textfile.txt")         → ".txt"
File.extname("igotnoextension")      → ""
```

I won't insult your intelligence by explaining what each of these mean, but I would like to note two things. First, the `size?` method returns the size in bytes, not kilobytes. It seems silly, I know, but that frustrated the piss out of me when I first started using it

(mostly because I'm dumb, I know, but I'm trying to save you some frustration here!). Second, the `size?` method will return `nil` if the file size is zero (another gotcha that bothered me until I figured it out).

You can also use the `File` class to find information about metadata such as ownership and permissions:

```
File.executable?("textfile.txt")     → false
File.readable?("textfile.txt")       → true
File.writable?("textfile.txt")       → true
File.owned?("textfile.txt")          → true
File.grpowned?("textfile.txt")       → false
File.setgid?("textfile.txt")         → false
File.setuid?("textfile.txt")         → false
```

The `executable?` (which determines if the user has the ability to execute a file according to filesystem permissions, not whether or not the file is an executable), `readable?`, and `writable?` methods have companion methods called `executable_real?`, `readable_real?`, and `writable_real?` (respectively, obviously) which make sure that the owner of the process has that ability with that file. Of course, if you own the file it probably doesn't matter. You can find out if you own it using the `owned?` method, which will return `true` if the process owner indeed owns the specified file. Normally the `grpowned?`, `setgid?`, and `setuid?` are very helpful in finding out certain metadata about a file, but these methods don't apply to and will always return `false` on operating systems that don't support them (I'm looking right at *you* Windows!). For those not in the know, on UNIX filesystems a file is owned by a user in a group rather than "just" a user; the `grpowned?` gains you access to this data. The `setgid?` and `setuid?` check for a bit that is set on a file's filsystem entry that allows you to change the user and/or the group when accessing that file (this helps when a user needs elevated privileges for a certain task). Again, these methods allow you to see if these bits are set, but if you're on Windows or something else that doesn't support them then they always return `false`.

Reading from a file

I can hear you saying, "Who cares about that crap?! I need to *read* a file. I made the file. I know all that crap about it! Tell me how to read it or I challenge you to a knife fight, right now, behind the Waffle House! You and me, pal! *We're taking it to the matresses!*" I would like to now kindly respond to your anger with this little tidbit:

```
myfile = File.open("textfile.txt", "r")
myfile.each_line {|line| puts line }
myfile.close
```

Using the `File#open` method, you can open a file and create a new `File` instance. The first parameter for open is the file path (either relative or absolute), and the second parameter is the file mode. You can view the table of options you have for this parameter in the table at the end of this section; this parameter defaults to reading if you don't specify. After you call open, you can use the `each_line` method to grab each

line and print it out, play around with it, whatever you want to do inside the block. You can optionally feed `each_line` a parameter that will act as the line ending in place of "\n"; if you, like me, tend to end each line of text with the word "pastry" you can respect this feature. Always be sure to call the `close` method if you are opening files this way.

"But, Dad!" you whine. "I don't wanna call close!" Well, Son/Daughter/Androgynous Offspring, Ruby can help you cure your incessant moaning:

```
File.open("textfile.txt") do |myfile|
  myfile.each_line {|line| puts line }
end
```

This does the same thing, but now the file stream is automatically closed when the enclosing block exits. "Wow!" you exclaim. I'm glad you're amazed, but it gets better:

```
IO.foreach("textfile.txt") {|line| puts line }
```

Using the `IO#foreach` method does the same thing as the previous two examples, just simpler, more compact, and far more beautifully. It opens the file specified, feeds it line by line into a block, then closes it. Mmm...now that's *Rubylicious*.

Writing to a file

Your options for writing to a file are numerous; they all accomplish essentially the same objective but in slightly different ways. The first (and most obviously named) way I'd like to cover is the `write` method. It goes something like this:

```
File.open("textfile.txt", "w") do |myfile|
  myfile.write("Howdy!")
end
```

You open a file with the `File#open` method, create an enclosing block, and simply call the `write` method on the file instance created by the block. You can do writing the same way I showed you reading the first time (i.e. without a block at all and calling `close`), but I thought that would be needlessly redundant to include it here. You can write any sort of data to a file as long as it can be converted to a string (i.e. it has a `to_s` method); if it can't be converted to a string Ruby will simply issue it a string representation to the effect of "`#<ClassName:SomeData>`". Other methods such as `print` and `puts` can easily be plugged into where `write` is; they take the same number of parameters and behave essentially the same way (except that `puts` will tag a new line on the end of the string when it is written).

Another way of writing to a file is utilizing the << operator; if you've ever used `IOStream` in C++ then you should feel right at home with this:

```
File.open("textfile.txt", "w") do |myfile|
  myfile << "Howdy!\n" << "There are " << count << "pandas!"
end
```

Opening the file is the same as always, but now instead of calling a method and feeding in parameters (at least in the traditional sense) you are now using the << operator. It behaves the same as the other methods (i.e. it converts the data to a string if it is not a string and writes it to the file) so there shouldn't be any surprising parts there. BOOGABLARGABOO! Okay, maybe that surprised you, but nothing else should.

More file operations

The `File` class also supports a number of other file operations that promote all sorts of filesystem hooliganism. Here are a few:

```
File.delete("textfile.txt")
File.rename("textfile.txt", "textfile.txt.bak")
File.chown(nil, 201, "textfile.txt")
File.chmod(0777, "textfile.txt")
```

The first two method's names should give away their function. If the proverbial cat is not out of the bag, they delete and rename the provided files (with the renamed filename fed in as the second parameter). The `delete` method will return the number of files deleted (i.e. `1` for this case).

The last two methods may be a little confusing if you are not up to snuff on your UNIX/Linux filesystems and their associated commands. The `chown` command allows a user with superuser privileges to change the owner of a file (or the owner may change the group ownership to any group of which he/she is a member); note that the `chown` method takes numeric owner and group IDs rather than string names (which the command line version allows). The `chmod` method allows the owner of a file (or a superuser) to change the permissions of a file (i.e. which users/groups can read, write to, or execute a file); the first parameter is a bit pattern which represents the permissions on the filesystem. Check Appendix A for URLs with more information on UNIX filesystem metadata (including bit patterns to be used with the `chmod` method).

FILE ACCESS MODES

r	Read-only access; starts at beginning of file (default)
w	Write-only; truncates existing file to zero length or creates new file
a	Write-only; starts at end of existing file or creates new file
r+	Read-write; starts at beginning of file
w+	Read-write; truncates existing file to zero length or creates new file
a+	Read-write; starts at end of existing file or creates new file
b	Binary file mode; may appear with any of the above options (Windows only)

THREADS AND FORKS AND PROCESSES, OH MY!

Don't you hate it when you get slapped with an hourglass/beachball when you're doing something simple? It's not like your computer is overloaded or anything; what's the deal? The deal is that the programmer (probably) didn't use a multithreaded design, so everything it does happens in one thread. What's a thread you ask? I'm glad you asked (if you didn't ask because you already know, skip this section).

Think of a thread as a way of telling your computer you want it to multitask. Let's say you're developing a WinAmp/iTunes/Foobar clone, and you want to be able to play music, have wicked awesome visualizations, and grab CDDB information about your tracks all at the same time. This isn't going to work very well in a single threaded setup because you will have to wait for CDDB to respond before your track plays, and then you have to worry about trying to draw and play music at the same time. The easiest solution would be to split each task off into its own thread. The music would play in its own thread, completely untouched by the other things going on; the visualizations would draw in their own thread, not interfering with the music; CDDB could be contacted independently of the other two, so that if you have a slow Internet connection, downloading the data won't bother the playback. Threads let your computer do more things at once, and are pretty important if you plan on doing anything remotely complicated with Ruby.

Ruby thread basics

Using threads in Ruby is as simple as passing a block of code to the `Thread` class's constructor. For example, let's create three threads. They will each do something at different intervals (e.g., print some text to the screen).

```
first = Thread.new() do
  myindex = 0

  while(myindex < 10):
    puts "Thread One!"
    sleep 3
    myindex += 1
  end
end
```

```
second = Thread.new() do
  myindex2 = 0
  while(myindex2 < 5):
    puts "Thread Two!"
    sleep 5
    myindex2 += 1
  end
end

third = Thread.new() do
  myindex3 = 0

  while(myindex3 < 2):
    puts "Thread Three!"
    sleep 10
    myindex3 += 1
  end
end

first.join()
second.join()
third.join()

→ Thread One!
→ Thread Two!
→ Thread One!
(and so on...)
```

To get threads going, you first need to create an instance of the `Thread` class and pass a block of code to it; our code simply prints some text and then makes that thread pause a few seconds using the `sleep` method. The calls to the `join` method aren't necessary to make this work; the threads will run by themselves and be killed when your program ends. The benefit of calling `join` is that your program will wait until all threads that have been joined exit (i.e. the code has finished and the block exits). If you take the join calls out of the above program, each thread will print once and exit because the main thread exited; as it is above (i.e., with the `join` calls), it will run for about 30 seconds, with each thread printing a few times.

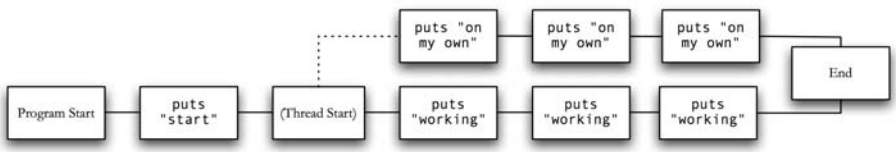


Figure 12: Threads allow you to do work in parallel to the main thread: multitasking for Ruby.

The join method is great, but what if you don't want a thread to run forever after you exit? Even further, what if you want to give it time to try to shut down after your program ends? Fortunately, the join method is pretty smart; you can feed it an integer as a parameter and it will use that as a timeout.

```
time_me_out = Thread.new() do
  while(true):
    puts "Keep loopin' loopin' loopin'..."
    sleep 5
    puts "Keep that script on loopin'!  RAWHIDE!"
```

```
    sleep 5
  end
end

time_me_out.join(15)

→ Keep loopin' loopin' loopin'
(5 second wait)
→ Keep that script on loopin'!  RAWHIDE!
(5 second wait)
→ Keep loopin' loopin' loopin'
(and so on...)
```

Since we gave join a timeout of 15 seconds, the script/song will only go for 15 seconds (since as soon as the thread is joined, the main thread exits; toss in a loop or something that runs for a while below it to make it run a little longer).

Controlling threads

Threads offer a few methods for controlling themselves. The first of these methods is `pass`, which will tell the thread scheduler to pass the execution to another thread. For example, let's say you have two threads and you'd like them to print things out and pass the control to each other as they do. Let's spell "weal" using two threads!

```
t1 = Thread.new { print "w"; Thread.pass; print "a" }
t2 = Thread.new { print "e"; Thread.pass; print "l" }

t1.join
t2.join

→ weal
```

The `pass` method basically tells the current thread to hang out for a second while the another thread does its thing. In the example, the threads switch off because they pause themselves to allow another thread to execute.

Another method that is used to control threads from within is the stop method. This method simply stops the thread's execution, which can be started again at a later time. The stop method is really useful for situations where a thread needs to pause until you can accomplish another task. Let's say you were designing some robotic sailors, and the first mate couldn't drop anchor until the captain says it's okay to do so. You'd probably do something like the following.

```
mate = Thread.new do
         puts "Ahoy! Can I be dropping the anchor sir?"
         Thread.stop
         puts "Aye sir, dropping anchor!"
       end

Thread.pass

puts "CAPTAIN: Aye, laddy!"

mate.run
mate.join
```

```
→ Ahoy! Can I be dropping the anchor sir?
→ CAPTAIN: Aye, laddy!
→ Aye sir, dropping anchor!
```

Rumor has it that is how the Love Boat actually started: robotic sailors. Anyhow, the stop class method stops the current thread, but as you can see in the example, the run instance method will restart it (remember: stop is a class method, run is an instance method). The thread can then be joined to continue on its merry little way.

Threads can be altogether exited also. You can do this one of two ways: either from within using exit or the outside using kill.

```
homicide = Thread.new do
             while (1 == 1):
               puts "Don't kill me!"
               Thread.pass
             end
           end

suicide = Thread.new do
            puts "This is all meaningless!"
            Thread.exit
          end

Thread.kill(homicide)

→ Don't kill me!
→ This is all meaningless!
→ Don't kill me!
```

They work the same, they just accomplish it different ways. It's usually better practice to kill a thread off from within simply because you know when and where it will be killed; killing threads off at will from wherever can lead to some serious confusion, and frankly, needless killing of innocent threads.

Getting information from threads

There are a few methods that can be used to grab information about threads. The first of these being `Thread.current` and `Thread.list`. The `current` method, of course, gives you access to the current thread. The `list` method lists all threads that are runnable or stopped. If you would like to get some information about these threads, then you can call the instance methods `alive?` and `status`. The `alive?` method will tell you whether or not the thread is active or not; it will return `true` if the thread is running or sleeping and `false` if it has exited or is stopped. The status method will return `"run"` if the thread is running as normal, `"sleep"` if the thread is sleeping, `"aborting"` if the thread is aborting, `nil` if terminated with an exception, and `false` if the thread terminated normally. Testing whether or not a thread is simply runnning can be done using the `stop?` method. For example:

```
mythread = Thread.new { Thread.stop }
mythread.stop?        → true
Thread.current.stop?  → false
```

Ruby returns `true` if the thread is stopped or sleeping; otherwise it will return `false`. You can also get the value returned from a thread using the `value` method.

```
calculator = Thread.new { 12 / 4 * 3 }
calculator.value
→ 9
```

This is excellent for long calculations whose value isn't needed right away; doing them this way lets you run them on another thread so they don't interrupt the main thread and the execution of your program.

Processes, the other way to do stuff

Sometimes you need to spawn a new process altogether. Whether you need to execute a third party program or just invoke another Ruby instance that runs your script, spawning external processes can be pretty important at times. Ruby offers a few ways to spawn and control new processes.

The `system` method PHP programmers rejoice! Ruby has a system method that operates like the PHP `system` method. Perl programmers may also rejoice, as Ruby also supports backtick notation for starting external processes. For those of you who are unfamiliar with both, let's just look at an example.

```
system("cat /etc/passwd")
extern = `whoami`
puts ("Your username is #{extern}.")          → jeremy
```

The functions of these methods is fairly obvious. The `system` method spawns an external application in a subprocess; it returns true if it is exited successfully and false otherwise (with the exit code in `$?`). The unfortunate thing about `system` is that it vomits the output on to wherever your application's output is being streamed to, which means you can can't capture it either. That's where the backticks come in; they also spawn an application in a subprocess but also allow you to capture its output.

Pipe dreams The `system` method works well enough in a lot of cases, but what if you need provide some interactivity with the application? Say you need to give it some input or perhaps it gives you delayed output and you'd like to start processing it before it's done executing. Ruby offers the `IO.popen` method that does just that.

The `popen` method will spawn an application and then give you a stream which you can read from and write to just like any other stream (e.g., file stream).

```
rb = IO.popen("ruby", "w+")
rb.puts "puts 'Whoa!  Radical subprocess, dude!'"
rb.close_write
puts rb.gets
```

As you can see, you open the pipe with `popen` just as you would a file stream, specifying a target and access mode (which are the same access modes for files). You can then use `puts` to write to to the stream and any other stream method (e.g., `gets`,

read, etc.). Do be aware that the close_write call was required for me. I'm not sure if this a platform issue or not, but it might be safe to just go ahead and throw it in there for good measure.

Independent Execution If you're on a machine that implements fork (i.e., not Windows) then you can use the exec method to execute things in a less hands on method while still retaining a little control.

```
exec("apachectl restart") if fork.nil?
# restarting apache...
Process.wait          # Wait for the process to exit (optional)
```

This has the same basic effect as using system, except that you can tell your application to wait until that process is done using Process.wait; this is a good idea if you're running a subprocess that could cause irreparable damage if exited prematurely (e.g., moving files or something like that).

FOR THE ENVIRONMENT!

Accessing much of the environment that your Ruby program is in is as simple as raising your Planeteer ring to the sky and calling down your personal power. Wait, that's not right. It's as simple as using a few neat functions that allow Ruby to access environment variables, program arguments, and a bit about Ruby's own environment. The power is yours!

Environment variables and the like

It's easy to access environment variables in Ruby; if you've ever used PHP or something similar to access environment variables, it's very much the same concept in Ruby.

```
ENV['SHELL']   → /bin/sh
ENV['HOME']    → /home/mrneighborly
ENV['USER']    → mrneighborly
```

The above values are fairly common if you're on a Linux-type system; if you're on Windows they change a bit. For example, on Linux USER is the environment variable holds your username, but on Window this value is placed in USERNAME. The other two example values evaluate to nil. I suggest that if you're going to use environment variables, that you pop open irb and do two things. First, make sure the value you want to use is valid on the platforms you'll be using it on. Secondly, call up the ENV collection and look at all the values available. If the value you planned on using isn't available, an alternate equivalent might be (e.g. USER and USERNAME).

To write to an environment variable, you simply assign a value much like you would a normal variable.

```
ENV['USERNAME'] = 'dontdothis'
puts ENV['USERNAME']
→ dontdothis
```

Your changes to environment variables are inherited by any child processes that you spawn, but they do not propagate back up to the parent of the Ruby application. All of your changes stay local to your application. If need be, you could spawn a process to use a command like `set` or `export` to change it in the shell.

The command line and you

Most Ruby scripts are invoked from the command line; this means that, if need be, you can pass arguments to it on the command line. For example, if you write a text editor, you could pass the file name you want to open to it on the command line (e.g. myeditor.rb myfile.txt) rather than having to use File -> Open or Ctrl+O or whatever. Perhaps you remember when we installed Ruby way back when, I had you type `ruby -v`, where Ruby is a command line argument. Ruby's faculties for accomplishing this very same thing are fairly simple to employ; you are given a global array, `ARGV`, to do with what you wish.

```
ARGV.each{|arg| puts "Arg: #{arg}; "}
```

If you were to invoke the above script with a few arguments, you would see them printed out sequentially.

```
ruby argv.rb "My args!" 123 19
→ Arg: My args!; Arg: 123; Arg: 19;
```

You can use this feature to gather information (e.g. filenames, numerical parameters, etc.) as I have done here, or you could use it to allow the user to specify command line switches (like -v on the Ruby command).

Ruby and its little corner of your computer

In the greater ecosystem of your computer, Ruby has its own little microcosm. When loading libraries and such, Ruby doesn't just magically know where they are; their paths are part of Ruby's environment configuration. Use the following Ruby invocation to see where Ruby looks.

```
ruby -e "puts $:"
```

On a typical Windows installation, you might see a list very similar to one like this.

```
C:/ruby/lib/ruby/site_ruby/1.8
C:/ruby/lib/ruby/site_ruby/1.8/i386-msvcrt
C:/ruby/lib/ruby/site_ruby
C:/ruby/lib/ruby/1.8
C:/ruby/lib/ruby/1.8/i386-mswin32
.
```

As you can see, Ruby looks in multiple locations typically within the Ruby directory; also note that it is version specific (i.e. it only looks at libraries installed in the 1.8

directory since we are using 1.8 here). The results on a Linux or OSX box should be similar; simply replace "`C:/`" with something like "`/usr/local/lib/`" or "`/usr/lib/`" and you should get a very similar list.

You can also gather some information about how Ruby was built and in what environment. This information can be useful is a bug exists for a specific build environment, and you want to see if a problem you are experiencing might be a result of that bug. This information is written to the `Config` module in the file `rbconfig.rb` in the library directory, usually in a directory under that which is labeled by the build environment (i.e. `i386-mswin32`). You can access this information programmatically also (since it is part of the library).

```
include Config

CONFIG['host_os']             → "mswin32"
CONFIG['target_os']           → "mswin32"
CONFIG['libdir']              → "C:/ruby/lib"
CONFIG['build_cpu']           → "i686"
```

Since the Config module is exposed, you can simply include it and call values from the global constant hash CONFIG to use them. Now the next time someone calls you a liar and tells you that can't possibly be running Ruby on an Apple II*e*, you can prove them wrong. *Dead* wrong.

WIN32 AND BEYOND.

Ruby is typically associated with the UNIX-based operating systems; heck, until about a year ago, Matz himself was prone to kicking you in the teeth for even mentioning Windows and Ruby in the same sentence. Fortunately, recent activity has promoted Windows to a position of (at the very least) tolerance within the Ruby community, evidenced in projects such as the One-Click Ruby Installer (http://rubyinstaller.rubyforge.org/) and the wonderful no-install Rails project, InstantRails (http://instantrails.rubyforge.org/). Ruby libraries for Windows have also been enhanced. There are a number of them available (search RubyForge if you'd like to see a sampling), but I want to focus on the `Win32API`, `Win32`, and `WIN32OLE` modules in the standard library.

API

Many applications today are directly tied to the Windows API. Many tasks such as INI file interaction are much easier to accomplish with the Windows API, and mechanisms like printing are only accessible through these channels. As such, when replacing current functionality in another language with Ruby code or when trying to use these mechanisms in your Ruby code, your only choice is to figure out a way to get Ruby and the Windows API talk to each other. Ruby offers the Win32API library to help ease the pain of this integration.

The `Win32API` module allows you to make calls to any Windows API module (e.g. `kernel32`, `user32`, and so on). You can make these calls by instantiating a `Win32API` object and calling the call the method on it. For example, let's say you needed to read from and write to INI files that are used by another part of your organizations system. These INI files house login information other essential bits that let the applications access mutual resources; as an example, let's build a little test file.

```
[database]
login = dbuser
password = foobaz

[fileshare]
username = shazbot
location = //server/path
```

Because of performance reasons (i.e. you don't want to write the regular expressions to parse the INI file) you decide to use the Windows API function `GetPrivateProfileString` to parse out values. To use GetPrivateProfileString within Ruby, you need to first look at the function definition and the function definitions of any related functions you will need (in this case, we will need the `lstrlenA` function also).

```
DWORD GetPrivateProfileString(
  LPCTSTR lpAppName,
  LPCTSTR lpKeyName,
  LPCTSTR lpDefault,
  LPTSTR lpReturnedString,
  DWORD nSize,
  LPCTSTR lpFileName
)

int lstrlen(
  LPCTSTR lpString
);
```

Now that we have the parameters and return types of these functions, we have all the information we need to contruct Win32API objects and make calls to these functions (this is all explained below!).

```
require 'Win32API'

# GetPrivateProfileString instance
getvalue = Win32API.new('kernel32', 'GetPrivateProfileString',
                        %w(P P P P L P), 'L');

# lstrlenA instance
strlen = Win32API.new('kernel32', 'lstrlenA', %w(P), 'L');

retstr = ' ' * (255 + 1)
getvalue.Call('database', 'login', '', retstr, 255,
'C:/test.ini')

length = strlen.Call(retstr)
puts retstr[0..length - 1]

→ dbuser
```

As you can see, you need to first include the Win32API module. Then, create instances of the Win32API class for each function you are going to call. In this case, we created two instances: `getvalue` which holds a reference to

GetPrivateProfileString and strlen which holds a reference to lstrlenA (both of which are labeled with comments). The first parameter for the constructor is the API module to look in (e.g. GetPrivateProfileString lives in kernel32), and the second parameter is the function name that you wish to call. The third parameter is an array of parameter types that the function takes; for example, in the above function declarations, we saw that lstrlenA takes a single constant string pointer (LPCSTR). The parameter type can be specified as one of three types: P for string pointers, N and L for numbers, I for integers, and V for void. So in our example, GetPrivateProfileString takes 5 string pointers and one long number parameter. The fourth parameter is the return type (e.g. we specified L for long number since DWORD, the type specified in the function definition, is simply a typedef for long). Using INI files is great for legacy systems, but even Microsoft knows that keeping all your important data in the Registry is where it's at nowadays!

The Registry

The Win32 module provides you with a very friendly interface to operate on the Windows registry. Is'nt that exciting? Now you can stick tons of essential information in there (such as license keys), have it overwritten, deleted, and generally molested by a virus, or even better, lost in a system restore because of said virus (I'm allowed to be bitter)! Even so, it's a dandy place to store general configuration information for your application if used properly.

To operate on the registry, you need to first call open on the Win32::Registry class and give it a registry path to open followed by a block. Think of this in the same way that you can open a normal file and have it close after the ensuing block is finished.

```
require 'win32/registry'

Win32::Registry::HKEY_LOCAL_MACHINE.open('SOFTWARE\Microsoft\Wi
ndows NT\CurrentVersion\') do |reg|
  # Do your dirty work here!
end
```

Let's just look at opening the registry up right now, and we'll look at what you can do when it's open in just a little bit. The above block will open up the registry path HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\. If you are unsure about what this means, just look it up on Google, but basically what we've done is open up the registry, look inside at the software area for the local machine, open up the CurrentVersion "folder" inside the Windows NT "folder" inside the Microsoft "folder " (I say "folder" because they're not real "folders" in the sense that they are navigable on the hard drive; they are phantom folders, mere ghosts, floating in the never ending ether that is the registry). If you get an error when running this, that probably means you're on Windows 98 or something like that; if that's true, then take off the NT on Windows. If you got an error, and you are on Linux, please put this book down and call your nearest tech support center. Now that we've got the registry opened up and ready to do something, let's grab some values and use them.

Reading Reading values in from the registry can be done a couple of ways; generally, you simply provide the key name and the value is returned.

```
value = reg['ProductName']
value = reg['PathName', Win32::Registry::REG_SZ]
type, value = reg.read('BuildLab')
```

The first call above is the most basic (and probably most practical in most cases); values from the registry can be accessed much like they are accessed from a hash. Simply enclose the registry key name (in this case "ProductName") in brackets. The next call will return the value, but will throw an error if the value returned does not match the type given as the second value (e.g., if the value for `PathName` were a `DWORD` in the registry, when you requested a `REG_SZ` it would have thrown a `TypeError`). The value of this type parameter can only be a certain number of constants, which are directly related to key types in the registry. The table below lists these constants for your reference.

REGISTRY TYPE CONSTANTS

`REG_NONE`	No specific type.	`REG_DWORD_LITTLE_ENDIAN` `REG_DWORD_BIG_ENDIAN`	A 32-bit number in little- or big- endian format (Windows is designed to run on little-endian platforms).
`REG_SZ`	A null-terminated string.	`REG_LINK`	Reserved for system use.
`REG_EXPAND_SZ`	A null-terminated string with some expandable expression like an environment variable.	`REG_MULTI_SZ`	A sequence of null-terminated strings, terminated by an empty, null-terminated string.
`REG_BINARY`	Binary data in any form.	`REG_RESOURCE_LIST` *	Nested arrays that store a resource list used by a hardware device driver or one of the physical devices it controls.
`REG_DWORD`	A 32-bit number.	`REG_RESOURCE_REQUIREMENTS_LIST` *	A complex data type for hardware configuration.
`REG_FULL_RESOURCE_DESCRIPTOR` *	Nested arrays of binary data that store a resource list used by a physical hardware device.	`REG_QWORD` `REG_QWORD_LITTLE_ENDIAN`	A 64-bit number.

* Keys of this type are not editable, but can be read.

The third and final call in the example uses the read method; the advantage to calling the value this way is that it gives you the type of the key. This is useful if you are

iterating over a set of keys and you need to perform specific operations on keys that are DWORD keys but not REG_SZ keys.

Enumerating You also have the option to enumerate values and subkeys (or sub "folders"), and in turn walk over the entire collection of values or a series of subkeys keys doing operations, storing or changing values, etc. You can use the each_value or each_key method to enumerate values or keys, respectively.

```
reg.each_value { |name, type, data| puts name + " = " + data }
reg.each_key { |key, wtime| puts key + " :: " + wtime }
```

The each_value method will iterate over each value in the current key and return its value, type, and name. The above code should output something like "ValueName = MyValue" for each value in the current key (e.g., if use with the previous code to open up the `CurrentVersion` key in the `Windows NT` key, you should see something "`BuildLab = 2600.xpsp_sp2_rtm.040803-2158`" for the first result).

Writing Writing values to the registry is very similar to reading them. The methods and their parameters are laid out very similarly.

```
reg['RevisionNumber'] = '1337'
reg['Name', Win32::Registry::REG_MULTI_SZ] = 'Mr.\0Neighborly\0\0'
reg.write('MyPath', Win32::Registry::REG_EXPAND_SZ, '%PATH%')
```

The first call shown above will simply write the value `2600` to the value `RevisionNumber`; notice that much like when reading values, you can make the registry act like a hash. The next call allows you to write a value with a specified type; like its read counterpart, it will throw a TypeError if it's the wrong type. The last call is very similar to the read method demonstrated above; the first parameter is the value name you wish to set, the second parameter is the type, and the third is the value to assign to it.

Deleting Deleting keys and values is as simple as calling a method; just hope you don't blow away something important on accident, because there's no undoing it!

```
reg.delete_value('MyVersion')
reg.delete_key('FavoriteCheese')
reg.delete_key('Ex-Wives', true)
```

The first method, `delete_value`, will delete the value which you specify as the first parameter. The second method shown, `delete_key`, will delete the key which you specify, and if the second, optional boolean parameter is provided, it will delete the key recursively (or not if you provide `false`).

OLE Automation

OLE Automation (well, officially just "automation" but the OLE term has sort of stuck) is a nifty little mechanism that Windows and many Windows applications offer that allows you to automate their operation. For example, Excel exposes an automation interface which allows you to launch, create new documents, edit documents, and so on. These interfaces are built with scripting in mind, so often

applications will provide a built-in way to tap into these interfaces (i.e. Visual Basic for Applications), but you can get to these interfaces with other clients also (e.g. C++ with COM, or, in our case, Ruby).

Automation Basics Ruby's OLE automation interface is very, very simple compared to most other languages. As an example, let's pop open Internet Explorer and navigate to the web page for this book.

```
require 'win32ole'

myie=WIN32OLE.new('InternetExplorer.Application')
myie.visible=true
myie.navigate("http://www.humblelittlerubybook.com")
myie.left = 0
myie['top'] = 0
```

As you can, the code is very simple. First you need to import the `win32ole` library using require. Next, create a new `WIN32OLE` instance, feeding in the application interface you want to talk to (in this case it's Internet Explorer, so `InternetExplorer.Application`). Note that parameter may or may not always be `Whatever.Application`, but it usually is; you may need to consult the application's documentation to find out exactly what it is if you are having problems. After that, you simply call methods on the interface (these should be outlined in the automation interface's documentation). In our case, we make the IE window visible, tell it to navigate to our web page, and move the window to the upper left corner of the screen (i.e. set the `left` and `top` properties to `0`). Note that properties can be set using either attribute notation (i.e., `myie.left`) or hash notation (i.e., `myie['top']`); since both forms do the same thing, it's really a matter of preference as to which one you should use.

Automation Events In addition to causing things to happen in an application, the Win32OLE class can also be used to be notified of what's going on in an application. This is done through an event sink mechanism that is exposed by the application and then consumed by your Ruby application.

```
require 'win32ole'

# Handler methods
def stop_msg_loop
  puts "Application closed."
  throw :appclosed
end

def handler(event, *args)
  puts "Event fired! : #{event}"
end

# Main code
ie = WIN32OLE.new('InternetExplorer.Application')
ie.visible = TRUE
ie.gohome
sink = WIN32OLE_EVENT.new(ie, 'DWebBrowserEvents')

sink.on_event {|*args| handler(*args)}
sink.on_event("Quit") {|*args| stop_msg_loop}

catch(:appclosed) {
  loop {
```

```
    WIN32OLE_EVENT.message_loop
  }
}
```

To subscribe to events, you need to follow the general procedure for consuming an OLE interface: import the `win32ole` library, create a new `WIN32OLE` instance, and call methods and/or attributes. The first new part of this code is the creation of a `WIN32OLE_EVENT` instance; the constructor for this class is given a `WIN32OLE` instance and the name of an event sink exposed by this interface (if the event sink doesn't exist, an error is thrown). You can then hook into events using the `on_event` method, which is given a block as a parameter; this block is then in turn given the event's arguments. You can use `on_event` in one of two ways. The first is to give it a general handler, as in the first call to `on_event; t`his handler becomes a sort of catch all for any events that don't have explicit handlers. You can also give events explicit handlers, like the second call gives the the "Quit" event. Note that right now there is no way to easily detach from an event, so our little "hack" to use catch to break out of the message loop seems to the be the easiest way to do it.

Windows Management Instrumentation The win32ole library also allows you to use the Windows Management Instrumentation (WMI) since it's simply a COM interface. WMI can be used for a number of administrative and management tasks, such as service management, process management, event watching, log auditing, and so on, on both local and remote machines. For example, to get a list of the services on the local machine along with their descriptions and status, you would do something like the following.

```
require 'win32ole'

mywmi = WIN32OLE.connect("winmgmts:\\\\.")

mywmi.InstancesOf("Win32_Service").each do |s|
  puts s.Caption + " : " + s.State
  puts s.Description
  puts
end
```

The `connect` method does basically the same thing as the `new` method, except the `connect` method hooks into an existing instance of an OLE server whereas the `new` method creates a new instance (i.e., WMI is already running as a server on your machine unless you disabled it, but for something like Word or Outlook a new, application-specific instance is needed). In this case, we used the InstancesOf method that is exposed by WMI to get an array of the instances of the WMI class `Win32_Service`, which is simply a representation of an entry in the service list for your machine. In our block we could have called methods such as `StopService` or `StartService` to control it or if there were processes we could use `Create` to start them, but for the sake of brevity (and the sanity of your machine), I opted to simply display the name, description, and status. When you run this script, you should see output for each service that looks something like this:

```
Task Scheduler : Running

Enables a user to configure and schedule automated tasks on
this computer. If this service is stopped, these tasks will not
```

```
be run at their scheduled times. If this service is disabled,
any services that explicitly depend on it will fail to start.
```

You can also get lists of processes, computers, users, groups, and so on, but that is out of the scope of this book. Look at the MSDN documentation for WMI linked in Appendix A for more on what information is exposed by WMI.

The `win32ole` library also allows you to subscribe to WMI events; WMI events span the whole gamut of system-wide events such as file creation, process creation, service actions, log entries, and so on. As an example, we'll watch for a process that we create to end.

```
require 'win32ole'

locator = WIN32OLE.new("WbemScripting.SWbemLocator.1")
service = locator.ConnectServer("./","","","")

proc = service.Get "Win32_Process"
rc = proc.Create('notepad.exe', nil, nil, nil)
processid = WIN32OLE::ARGV[3]
puts "New process id: #{processid}"

query = "select * from __InstanceDeletionEvent within 1 where
targetinstance isa 'WIN32_Process' and targetinstance.Handle =
#{processid}"

event = service.ExecNotificationQuery(query)
event.nextevent

puts "Process terminated."
```

Because of the way the WMI operates, you can't use the built-in event mechanism in the win32ole library, but it's fairly simple to get this working nonetheless. First, create a new instance of the `WIN32OLE` class using the new method and point it at the server `Wbemscripting.SWbemLocator`. This OLE server is the basic equivalent of connecting like we did before (`winmgts://`), but using this in conjunction with the subsequent call to `ConnectServer` allows you to do two things.

First, you can connect to remote computers, meaning you could use this code or similar snippets to do various tasks on a number of computers you may be managing. Second, you can provide login credentials as the last two parameters to this method. None are needed on the local machine usually (unless you are not an administrator or privileged account), but if this were a remote machine call you would probably need a user name as the third parameter and the password as the fourth. Next we use the WMI method to get a reference to the `Win32_Process` class and create an instance of it, passing in "`notepad.exe`" as the process to start. Doing so creates a new Notepad instance (i.e., you should see Notepad pop up on your screen) and returns some data about it, which we use to get the process ID.

Next, we use WQL (WMI Query Language) to tell WMI that we're going to be watching for processes to be destroyed that have the process handle (ID) of the process that we created; the WQL statement is executed using the `ExecNotificationQuery` method that is exposed by WMI. This method returns an event notifier object, which we can call the nexteven method on; this method tells the current thread to pause until the next event is fired, so if you were to use this in an

actual application that should continue running, this should be forked into its own thread. Close Notepad and you should see the output `"Process terminated"`; this means that the next event has fired (i.e., the process has been deleted) and the current thread has been given control once again. This is a fairly simplified example, but hopefully it should give you the basic concepts to translate more complicated examples from VBScript, C++, or C# into Ruby and make use of them.

This Chapter

You learned about Ruby's system level interaction. You learned...

- about Ruby's filesystem interaction capabilities.
- how to start threads and fork processes.
- how Ruby can interact with Microsoft Windows on a number of levels.

5

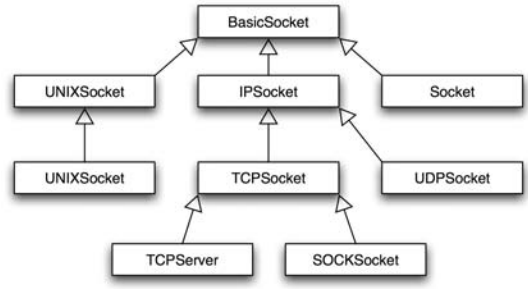
Looking Beyond Home

In this day and age, the rage with all the kids seems to be those "networked" computers. Well, whatever that means, those young'uns don't need to look any further than Ruby for their network fix, whether it be getting cracked out on HTTP or hopped up on databases. One of my favorite thrills in high school was hitting an Ethernet line and then listening to Link Void's "Darkfiber Line to the Room" and watching "The Wizard of Oz" at the same time. Fun times.

NETWORKING AND THE WEB

Ruby's networking abilities have been favorably compared to languages such as Python and Perl; while Ruby doesn't have huge libraries such as CPAN or tightly integrated suites like Twisted (yet), it does present a nice suite of networking capabilities built right in.

Socket Programming



Ruby offers a number of socket types that you can use to connect to different transport and network protocols. Both connection- and connectionless-oriented transports are offered, along with classes that offers client and server support. All sockets hail from the Genesis of all socket classes: `BasicSocket`. The `Socket` class provides you with a very C-ish interface to sockets; it is much more complex and a generally pain in the bum compared to the other classes such as `IPSocket`, `UDPSocket`, or `TCPSocket`.

The rest of this section (i.e., the part dealing with actually networking an application) will concentrate on the `TCPSocket` and `TCPServer` class, since these classes will be the most common classes you will use and are easier than using the same functionality using the `Socket` class. There is a lot of information available on using the other classes, so if you need to use something like a UDP socket, then consult one of the links in Appendix A or use your favorite search engine to find what you need.

Server The `TCPServer` class offers a simple way to get socket server functionality up and going in your application with minimal fuss when dealing with accepting connections and the like. There's not much background other than what's already been discussed with regard to sockets in general, so let's get right into some code and construct the de facto standard example socket server example program: the echo server.

```
require "socket"

myserver = TCPserver.new('localhost', 0)
sockaddr = myserver.addr

puts "Echo server running on #{sockaddr.join(':')}"

while true
  Thread.start(myserver.accept) do |sock|
    puts("#{sock} connected at #{Time.now}")

    while sock.gets
        sock.write($_)
        puts "User entered: #{$_}"
    end

    puts("#{sock} disconnected at #{Time.now}")
    s.close
  end
end
```

Let's take this apart. First, you must include the socket module, which will give you access to `TCPServer` and all its related modules and classes. Next, you must create a `TCPServer` instance using the either the `new` method or its synonym, the `open` method. The first parameter is the interface to bind to; if this is left blank, then Ruby binds to all available interfaces on the host. The second parameter is the port number; if this is `0` like in the example, then the system automatically selects an available port. Then upon entering the loop, we tell the application to wait until a new socket is connected (which fires myserver.accept); when the socket connects, a new `Thread` object is created and the enclosed block is executed. I use a `Thread` here simply because this setup allows you to connect more than one client to this particular server instance. You could do it "blocking" using a while loop or some such, but it's generally better practice to do it like this. The `Thread`'s block has logic to output a message upon connect and disconnect of a host and output user data entry on the console and back to the socket. Upon running this, you should have a simple socket server that will echo back any input given to it.

```
Echo server running on AF_INET:3160:localhost:127.0.0.1
```

When the server starts, you should see a little bit of information about it, including the port number (which is highlighted above). Now if you open up a telnet client and telnet to localhost on the given port number, you should be able to enter some text and have it echoed back to you.

```
myhost> telnet localhost 3160

some more text!!
some more text!!
the pandas are livid!!
the pandas are livid!!
Pancakes are a far better source of Omega-3.
Pancakes are a far better source of Omega-3.
```

After you close the telnet window, flip over to the Ruby console that your application has been running in and look at the output.

```
Echo server running on AF_INET:3160:localhost:127.0.0.1
#<TCPSocket:0x27ebefc> connected at Wed Sep 13 19:44:28 Eastern
Daylight Time 2006
User entered: some more text!!
User entered: pandthe pandas are livid!!
User entered: Pancakes are a far better source of Omega-3.
#<TCPSocket:0x27ebefc> disconnected at Sat Sep 13 19:45:48
Eastern Daylight Time 2006
```

When the application started, you were given a nice little startup message, which is followed by messages indicating the various events going on in the application. Now let's take a look at TCP client services.

Client TCP client services are offered by the `TCPSocket` class; this class allows you to make a TCP connection to various services, such as finger, ftp, your own service on a random port, or even a telnet server. Let's connect to the BOFH Excuse telnet server on blinkenlights.nl as an example; this way, when your boss asks you why you're reading this book instead or working, you can provide him or her with a proper excuse.

```
require "socket"

print("Connecting...")
clientsock = TCPsocket.open("towel.blinkenlights.nl", 666)
print(" done\n")

puts("Connected from #{clientsock.addr[2]}
to #{clientsock.peeraddr[2]}")

while (excuse = clientsock.gets)
  print(excuse)
end

clientsock.close
```

First, you need to import the socket library, of course. Next, you open a TCPSocket much like you open a file stream. The first parameter is the hostname or network address to connect to, and the second is the port to connect on. Now that you have a socket, you can treat it just like a stream that you would get from a file or something similar, hence why you can use `gets` to get data from it. You can also use `read`, `readline`, `write`, `writeline`, and any other stream reading/writing command.

Sockets also have their own methods, such as `recv` to receive data from the socket and `send` to send data over the socket, but it is not necessary to use these. It's simply a matter or preference and style. When you run this code, you should get output similar to the following.

```
Connecting... done
Connected from host.domain.com to towel.blinkenlights.nl

=== The BOFH Excuse Server ===
Your excuse is: It's not plugged in.
```

First, you're given a few messages about the connection that is made, followed by the output returned from the server. We've basically telnetted (is that a verb?) into the server and received the output, which means that we could point a TCPSocket at a TCPServer and have them talk to each other. Let's create a client for our echo server.

```
require "socket"

print("Connecting...")
clientsock = TCPsocket.open("localhost", 1390)
print(" done\n")

while (gets())
  clientsock.write($_)
  puts(clientsock.gets)
end

clientsock.close
```

Notice this code is very similar to our previous code, with the exception of a few things. First, we changed the hostname and port number (which you will need to get from the echo server beforehand or take as a command line parameter to your Ruby script). Secondly, the main loop now operates on user input rather than socket output, and in this loop, take user input and write it to the socket. The socket's output is then written to the console. When you run this and enter some data, you should get something like the following.

```
Connecting... done
localhost to localhost
Chicken livers and potbellies!
Chicken livers and potbellies!
My bowler is dusty from my superfluous emissions.
My bowler is dusty from my superfluous emissions.
```

Now, if you flip over to the server's console window, you should see something very similar to the following.

```
Echo server running on AF_INET:1390:localhost:127.0.0.1
#<TCPSocket:0x27ebefc> connected at Sat Sep 16 17:34:07 Eastern
Daylight Time 2006
User entered: Chicken livers and potbellies!
User entered: My bowler is dusty from my superfluous emissions.
#<TCPSocket:0x27ebefc> disconnected at Sat Sep 16 17:41:51
Eastern Daylight Time 2006
```

Congratulations! You've successfully written a fully networked Ruby application. This example is frivalous and useless of course, but it should illustrate the basic

concept enough to build off of. Before you know it, you'll be porting Apache to Ruby or telling your boss that you don't need no stinkin' web browser if you've got Ruby.

HTTP Networking

Speaking of the web, HTTP networking is also a very important component of today's network programming landscape. HTTP, or HyperText Transfer Protocol, is the standard for Internet communication these days. I'm still not sure why the world ever got away from ALOHA, but then again I'm not sure why 640K of memory isn't enough for everyone nowadays either. In any case, Ruby provides a robust library of HTTP functionality built right in to the standard library.

Server HTTP serving with Ruby is incredibly easy. You could build a server based off of TCPServer, meticulously implementing each part of the HTTP protocol, solving concurrency and blocking issues as they came up, and figure out a way to reliably send binary data. Then again, you could just use WEBrick, the full-Ruby web server that is included in the standard library.

```
require 'webrick'
include WEBrick

myserver = HTTPServer.new(:Port => 8080,
                          :DocumentRoot => Dir::pwd + "/temp")
myserver.start
```

Using the above four line snippet, you now have a fully operational HTTP server running on the port your specify that will serve HTML documents out of the path you specify as the DocumentRoot option (e.g., in the example it's the folder the script is launched from and "temp" like "/home/yourname/temp"). To start the server, simply give create a new instance and call the `start` method. The parameters shown for the constructor are the most commonly used, but there are a lot you can assign. If you're really interested in finding out more, you can see all of them in the WEBrick documentation at http://www.ruby-doc.org/stdlib/libdoc/webrick/rdoc/index.html.

Back to the example, if you have an HTML file named index.html dropped into the path you specified, you should be able to navigate to http://localhost:8080 and see it.

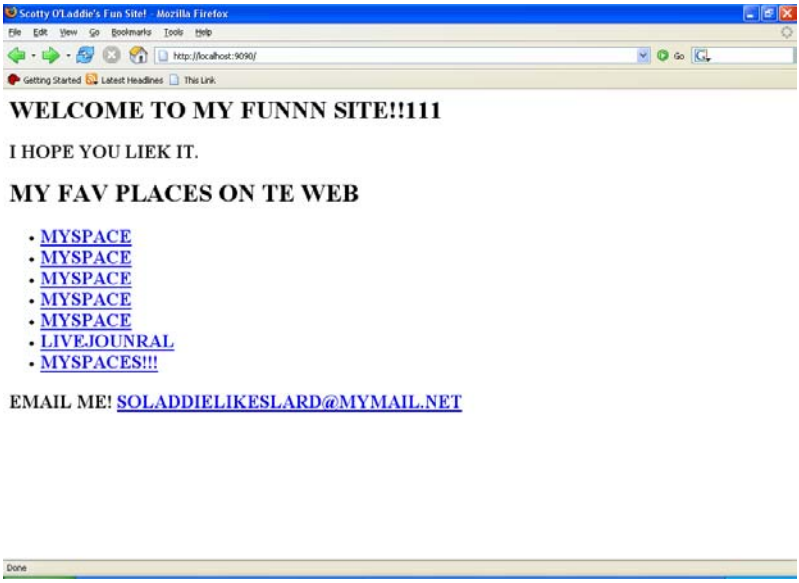


Figure 13: Scotty O'Laddie's Fun Site! Pretty much the best website in the world.

If you don't have an index.html file, it will show you an index of the files that are in there; you can then request one of those either by clicking the link or navigating to it directly. Now that you have a working HTTP server, let's take a look at Ruby's HTTP client facilities.

Client Ruby's HTTP client library is the `Net::HTTP` class. This class provides a simple interface to making HTTP requests to local and remote servers for any type of file. Possible uses include periodically downloading something like logs or new feeds or if downloading a large number of files without making a request for each one through a web browser or something like `wget`. As an example, let's say you wanted to download Digg's RSS feed so you could parse it into a Ruby on Rails website or your own homegrown RSS reader; you could do so by making a request with the `Net::HTTP` class.

```
require 'net/http'
Net::HTTP.get_print 'www.digg.com', '/rss/index.xml'
```

First, you need to make include the net/http library using require. In this instance, I used the `get_print` method to print out the file to the console., which takes a hostname and a file to retrieve. This is the best way to do do this if you want to get the same file from a number of servers (i.e., you could leave the file as a constant string and have the hostname be a variable) or a lot of files from the same host (i.e., you could leave the hostname as a constant string and the file variable), but if you are generally fetching files from different servers and of different filenames, there is a more intuitive way to do so.

```
require 'net/http'
require 'uri'

Net::HTTP.get_print URI.parse('http://www.ruby-lang.org')
```

The above snippet will grab the Ruby Language home page's source code and print it on the console. If you notice, rather than feeding the get_print method two separate parameters (i.e., the hostname and the filename), I used the URI.parse method from the uri library; this method allows you to take a normal URL (e.g., http://www.bn.com/, http://www.ruby-lang.org/, http://rubyonrails.org/index.php, and so on) and pars e it for use with any of the Net::HTTP methods. I show this early in this section because it is the easiest way to give URLs to these methods; speaking of those methods, let's take a look at some of the other methods in this library.

So far I've been using the `get_print` method to get and print the contents of a request. This method is great if that's all you're doing, but what if you need to get the contents and store in a variable? Or what if you want to manipulate the HTML some way? That's where the `get` method comes in; this method will execute a request and then return the results for you to have your way with.

```
require 'net/http'
require 'uri'

src = Net::HTTP.get(URI.parse('http://www.cnn.com/'))
puts src
```

As you can see, the same basic process is followed as before with `get_print`, except this time the results are first stored in the `src` variable.

The Net::HTTP library also allows you to post form data to a URL. You can do this over GET by appending the form data to the end of the URL (e.g., http://your-host.com/index.php?underpants=secured&sandpaper=finegrain&lotion=yes), but a little more effort is required to make a POST request. A POST request are the sorts of form posts that don't append anything to the URL, but rather submit the form data as part of the HTTP request. You can use the post_form method to make this kind of request with Net::HTTP.

```
require 'net/http'
require 'uri'

postit =
Net::HTTP.post_form(URI.parse('http://zip4.usps.com/zip4/zcl_3_
results.jsp'), {'zip5'=>'37998'})

puts postit.body
```

The above snippet will use the USPS ZIP code lookup to find out what city a ZIP code is from; using the `post_form` method, you can post a form by providing the form keys and values in a hash as the second parameter (i.e., in this case, `zip5` was a text field on the normal page which was for the ZIP code you wish to look up). The results are returned just like a normal GET request.

The final portion of this library I would like to cover is the `Net::HTTP.Proxy` class, which allows you to connect through a proxy. Instances of HTTP.Proxy have

the same methods as a normal Net::HTTP instances, so you can use it almost the same way.

```
require 'net/http'
require 'uri'

src = Net::HTTP::Proxy('myproxyhost', 8080, 'username',
'password').get(URI.parse('http://www.cnn.com/'))

puts src
```

When creating an instance of the `HTTP::Proxy` class, you must give it a proxy address and port; the last two parameters for a username and password are optional (but, of course, required if your proxy requires authentication). Now any request you make using that instance will go through the provided proxy.

I didn't cover everything about `Net::HTTP` here. Think of this as a mere brushing of the surface, uncovering a mere snippet of what can be seen at http://www.ruby-doc.org/stdlib/libdoc/net/http/rdoc/index.html. I encourage you to go there and look over the documentation, as we have only uncovered "te wolechla" of the the whole enchillada.

Other Network Services

Though I've covered the two most popular protocols, that doesn't mean the little guys won't get any love. Since when is it that only the popular kids matter? Well, probably since about 1944, but that's beside the point. I'm going to cover the nerds and geeks of the network protocol community because I want to (that and the Commission for Equal Treatment of Network Protocols and Their Ugly Cousins That Suck at File Transfers might yell at me if I don't).

FTP The File Transfer Protocol (FTP) is a fairly popular method for transferring and storing files, particularly in open source circles. As such, being able to grab these files (or upload them, whatever thc case may be) with a Ruby script may prove to be incredibly handy. Fortunately, Ruby provides a built-in library to handle all of your FTP needs.

```
require 'net/ftp'

ftp = Net::FTP.new('ftp.gnu.org')
ftp.login('anonymous', 'me@mrneigborly.com')

ftp.chdir('gnu/readline/')
files = ftp.list('r*')
puts files

ftp.getbinaryfile('readline-5.1.tar.gz', 'rline.tar.gz')

ftp.close
```

The above snippet will login to the GNU FTP server and download the readline library. Ruby's FTP library, just like every other library, must first be included using `require`. Next, you need to create a new `Net::FTP` instance, providing the constructor with a hostname to connect to. The `login` method will then log you into the FTP server; if you are logging in anonymously (i.e., like in the example) it's not

completely necessary to provide a username and password. If left blank, Ruby will substitute `anonymous` in as the username and `anonymous@yourhostname` as the password. Next, you need to navigate to the proper directory; here, I used the `chdir` method to change directories to the `gnu/readline/` folder and the `list` method to get a list of the files in that directory, which is then printed to the console. Once you're where you want to be, you can use `gettextfile` or `getbinaryfile` (as I have here) to get files or `puttextfile` or `putbinaryfile` to upload local files to the server (if you're allowed to). The get methods require two parameters: the remote file to get, the local filename to copy it to. The put methods require two parameters also: the local file to copy, and the remote filename to copy to. If you'd like to see more examples, go to http://www.ruby-doc.org/stdlib/libdoc/net/ftp/rdoc/index.html to view this library's documentation.

SMTP The SMTP Protocol, or Simple Mail Transfer Protocol...er...Protocol, is the de facto standard for sending e-mail. Thanks to this protocol and it's usually crappy implementation, you and I get thousands of spam e-mails every day. Thanks Jon Postel, Eric Allman, Dave Crocker, Ned Freed, Randall Gellens, John Klensin, and Keith Moore for inflating my inbox (and thanks Wikipedia for telling me who worked on SMTP)! So, anyhow, using SMTP in Ruby is actually quite painless.

```
require 'net/smtp'

message = <<MESSAGE
From: Your Name <you@you.com>
To: That Guy <him@him.com>
Subject: My fine hat is misplaced
Date: Sat, 15 Sep 2006 16:26:43 +0900
Message-Id: <thisisunique@you.com>

Where is my fedora?
MESSAGE

smtp = Net::SMTP.start('me.com', 25)
smtp.send_message msgstr,'you@you.com','him@him.com', 'you.com'
smtp.finish
```

First, tell Ruby to require the `net/smtp` library (I know it seems redundant, but you'd be surprised how many times even I forget this). Next, I constructed a message in a string. Message construction would probably be handled differently in a "real" application, but this method works for this example. Next, create a new `Net::SMTP` instance by calling the start method, giving the constructor a hostname and port for your SMTP server. Next, call the `send_message` method, giving it the message, the sender's e-mail address, and the recipient's e-mail address, and HELO domain as parameters. The HELO domain is used by some mail servers and spam filters to make sure that the e-mail is legitimate. Finally, call the `finish` method to end the SMTP session. If your SMTP server requires authentication, you need to make the start method call like the following.

```
Net::SMTP.start('me.com', 25, 'you.com', 'you', 'mypw', :plain)
```

The three additional parameters handle your authentication. The first of these is the username and password combination; the last parameter indicates which authentication scheme will be used. In this case, I used the evil and much maligned

`plain` scheme, but there are the `login` scheme and the slightly more secure `cram_md5` scheme which you can use. Check with your server's administrator to make your server supports these methods before trying to use them as some crappier mail servers don't support them.

POP The POP protocol, or Post Office Protocol...protocol, is (obviously) used to retrieve e-mail. It has largely fallen out of favor in recent years since more people have discovered IMAP (which is covered in the next section) but is still in widespread use. Ruby's POP library is an easy to use interface to a POP server, and quite honestly one of the more natural POP interfaces I've used. Other interfaces can be compared to a McDonald's worker that speaks Engrish. You know what you want them to do, but either they don't understand or they do it in a way that may make sense but only to someone who is either stupid or simply strange. Fortunately, Ruby's POP interface is natural and readable.

```
require 'net/pop'

pop = Net::POP3.new('mymailserver.com')
pop.start('myaccount', 'salacious')

if pop.mails.empty?
   puts 'No mail to grab.'
else
  pop.each_mail do |msg|
    puts msg.pop
    # msg.delete
  end
end

pop.finish
```

This code snippet will open a new connection to a POP server and log you in by creating a Net::POP3 instance and calling the `start` method with login credentials as parameters. Next, it checks whether or not the mailbox is empty by using the `empty?` method on the `mails` attribute of the Net::POP3 instance. If they mailbox is empty a message is displayed indicating its emptiness, but otherwise the `each_mail` method is used to iterate the mail messages and print them out. The commented out code would delete the message, but I didn't want to put destructive code in something that you're likely to copy and paste without regards to content. Since the `mails` attribute is simply an array of mail messages, you could also use the `each` method to iterate the message, but it's probably a better idea to use the `each_mail` method since it is specialized (not that it makes a difference, but it's probable that the other one will become deprecated eventually).

I didn't cover all the features in the section; one reason is that I don't have a server that supports them all (i.e., APOP). Another reason is that they're not common enough to warrant anymore documentation than you can find on the documentation page at http://www.ruby-doc.org/stdlib/libdoc/net/pop/rdoc/. Let's move along to a superior mail retrieval method: IMAP.

IMAP IMAP (Internet Message Access Protocol) is the e-mail flavor du jour currently since it's a little more efficient than POP. It allows you to have many logical mailboxes rather than a single inbox, multiple client access to one mailbox, and much more. Put

simply, it's a natural evolution of POP. Ruby's IMAP interface is surprisingly robust; it gives you a natural interface to all of IMAP's commands. Let's take a look at an example from the Ruby documentation that will grab all mail messages and print out their sender and subject.

```
require 'net/imap'

imap = Net::IMAP.new('mail.example.com')
imap.authenticate('LOGIN', 'joe_user', 'joes_password')

imap.examine('INBOX')

imap.search(["RECENT"]).each do |msg_id|
  envelope = imap.fetch(msg_id,"ENVELOPE")[0].attr["ENVELOPE"]
  puts "#{envelope.from[0].name}: \t#{envelope.subject}"
end
```

This example works much like the POP example in the first few lines: create a new instance parametrized with the hostname and call a method to authenticate). The two authentication schemes supported by IMAP are `LOGIN` (used here) and `CRAM-MD5`. After these lines though, IMAP's architecture really starts showing. The `examine` method gives you read-only access to a mailbox or folder (the `select` method, which is used the same way, allows editing access); you can then use the `search` method to grab a list of mails to operate on. You can use the `fetch` method to get a mail object, which you can get information from (as in the example) or use methods like `copy` or `store`. To get more information, check out the IMAP library's documentation at http://www.ruby-doc.org/stdlib/libdoc/net/imap/rdoc/index.html.

Web Services

Web Services are growing in popuarity these days, and as such I thought it was pretty important that I at least mention them. I'm only going to cover a very basic client side example, as this could honestly be a book on its own if anyone wanted to write it. There are a few links in Appendix A that talk about using Ruby and web services together if you're interested; otherwise let us march on towards our web service laden Promised Land. We have yet to cross the Red Sea, but, brothers and sisters, it is in sight!

XML-RPC Consuming an XML-RPC (XML Remote Procedure Call) web service in Ruby is as simple as fetching a web page, except instead of giving you a rather unusable raw return value, Ruby gives your beautifully and transparently converted values that you can use right away. Let's look at a slightly edited example from the documentation.

```
require 'xmlrpc/client'

server = XMLRPC::Client.new2("http://xmlrpc-
c.sourceforge.net/api/sample.php")

result = server.call("sample.sumAndDifference", 5, 3)

puts "#{result['difference']} and #{result['sum']}"

→ 2 and 8
```

First, you need to include the library and create a new instance using the `new2` method (or `new` with a lot of parameters or `new3` with a hash of every parameter needed; it's sort of up to you). Then use the `call` method to make an XML-RPC call to the server; the call method takes the remote method as the first parameter and the method's parameters as the remaining parameters. The results are then returned to you as a usable hash (i.e., result[difference]) rather than a random, annoying, unparsed XML string. If you are interested at looking at the documentation for the XML-RPC client, look at http://www.ruby-doc.org/stdlib/libdoc/xmlrpc/rdoc/index.html; if you want to implement an XML-RPC server, look in Appendix A for links to some information.

SOAP SOAP (which used to stand for Simple Object Access Protocol, but now it apparently stands for SOAP) is the supposed successor to XML-RPC. It was supposed to be a way to share objects over a network, but somewhere along the way everything went awry, the FCC got involved, and SOAP doesn't do what it was meant to do. In any event, it's a fairly common and powerful way to do web services. Here is an example from the documentation for soap4r, the standard way to do SOAP in Ruby, which will translate from English to German using a SOAP interface for Babelfish.

```
text = 'Here I lie in a sweater poorly knit.'
lang = 'en_de'

require 'soap/rpc/driver'

server = 'http://services.xmethods.net/perl/soaplite.cgi'
InterfaceNS = 'urn:xmethodsBabelFish'
wireDumpDev = nil

drv = SOAP::RPC::Driver.new(server, InterfaceNS)
drv.wiredump_dev = wireDumpDev
drv.add_method_with_soapaction('BabelFish', InterfaceNS +
"#BabelFish", 'translationmode', 'sourcedata')

puts drv.BabelFish(lang, text)
→ hier liege ich in einer Strickjacke stricke schlecht
```

This looks complex, but it's mostly just setting up the SOAP parameters. First, I put in some seed data to work with, which is basically a sentence that will be translated from English to German. Next, you can see that I included the library and then put some SOAP parameters into variables. The first value, `server`, is the location of the service to connect to; the second, `InterfaceNS`, is the namespace that the method you wish to call is located in. The last one is the "dump" handler for errors, that if set to `nil` becomes the standard error channel. Next, a new `SOAP::RPC::Driver` is created that points to the service and namespace specified earlier and is told to dump to stderr (standard error). The next little bit of code demonstrates an interesting feature of soap4r, which is probably what sets it apart from most other SOAP wrappers. The `add_method_with_soapaction` uses Ruby's dynamic nature to add a method to the `Driver` object. This means that rather than marshalling and calling and doing a lot of other random stuff, you can now call the method on the `Driver` object like any other method (which we do at the end of this snippet).

Once again, I'm not covering nearly all of this library. There are so many features in there that I couldn't dream of trying to do it justice in one little section;if

you're really curious and want to find out more, you can check out the soap4r website at http://dev.ctor.org/soap4r/ or the other links Appendix A for more information.

IT'S LIKE DISTRIBUTED OR SOMETHING...

In today's world of enterprise software, loosely coupled systems, and kosher beef franks made by Jews, distributed computing is becoming increasingly more important. Perhaps you need to expose an interface to the world for usage of your resources; maybe you need to distribute heavy computing across a grid of computers; maybe you want to calculate something in a number of environments with minimal effort. The DRb (Distributed Ruby) package provides a simple way to handle distributed computing in Ruby.

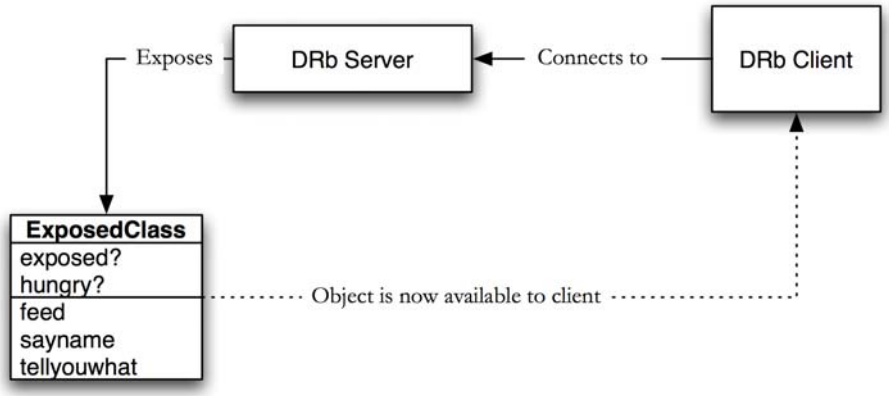


Figure 14: A typical DRb setup: server and client.

Every DRb application has two components: a server and clients. The server will start a TCP server, which will expose objects, accept connections, and respond to actions requested over those connections. Clients will establish connections to a DRb server, bind to objects, and send messages to those objects similar to messages that are sent to local objects when calling methods and attributes. Let's build an example to get a feel for how this works; let's say you needed to keep track of your log sizes on a remote server and wanted to keep that figure on your desktop for easy monitoring.

Server Setting up a DRb server is simple if you have the class you want to expose already constructed. In this case, I'm exposing a rather trivial class that simply takes one value as a parameter (the log file to watch) returns one value (the size of the log file).

```
class FSWatcherService
  def initialize(filename)
    @file = filename
  end
```

```
  def getspace
    return File.size(@file)
  end

end
```

Now that you have a class to work with, you need to wrap it in a DRb server. This is done by simply calling the `start_service` method from the drb module and giving it a few parameters.

```
require 'drb/drb'
require 'thread'

class SizeService

  def initialize(filename)
    @file = filename
  end

  def getspace
    return File.size(@file)
  end

end

DRb.start_service("druby://:9876", SizeService.new('cats.log'))
puts DRb.uri

DRb.thread.join
```

First you need to include both the DRb module (`drb/drb`) and the `thread` module so you can join the DRb server's thread into the main one. Next you see the class we made earlier. Then you need to call the `start_service` method from the DRb class; this will start a DRb server on the specified URI and expose the indicated object. The DRb URI is then printed to the console (for reference and for your benefit when trying to connect to it). The DRb server's thread is then joined to the main thread, and the server idles until connected to.

Client DRb makes constructing a client dirt simple. The basic process is that you call start_service, bind to the remote object, and then use it just like a remote object.

```
require 'drb/drb'

Drb.start_service
remoteobj = DRbObject.new(nil, 'druby://domo:9876')
puts "Log file usage is #{remoteobj.getspace()} bytes."
```

As you can see, you simply create a new `DRbObject` and give it a DRb URI as a parameter, and voila! You've got a remote object you can use just like a local object; in this example, I called the `getspace()` method on the remote object and printed its value to the console.

I didn't cover every nook and cranny of DRb here (and I didn't intend to; it's a robust package); it also offers ways to make your application thread safe (so that when you have a lot of clients connecting and disconnecting that your data stays kosher for all of them) and a host of security measures. Check the links in Appendix A to find out about all that stuff and more.

DATA MY BASE, PLEASE!

Database driven applications are incredibly common nowadays; everyone's got one. Your wife, your mom, your son, the creepy guy at work with a nasty mustache and halitosis. If you don't get your butt in gear and get one, your boss is going to fire you for being the only square who isn't rolling on dubs, RDBMS style.

DBI Ruby's DBI package, much like the Perl package, is a database vendor-independent database connection suite. Its robust support for database features and vendors is nearly unparalelled. As an illustration, here is a table of supported database vendors and connection types.

ADO (requires Win32)	DB2	Frontbase	InterBase
mSQL	MySQL	ODBC	Oracle
OCI8	PostgreSQL (Pg)	SQLite	SQLRelay

It supports more connection types, such as a Proxy through DRb, and even more vendors through interfaces like ADO and ODBC. Let's take a look at how to connect to one of these databases.

The basic process for making DBI work is to connect to the server, providing a host and credentials, execute or prepare and execute SQL statements, fetch the results, and, when you're done, disconnect from the server. Let's build a quick example that will show off most of the functionality of DBI; let's say you needed to keep a database of all of the specimens that are in your exotic slug collection.

```
require 'dbi'

DBI.connect('DBI:Mysql:mydb', 'myuser', 'mypass') do | dbh |

  dbh.do('CREATE TABLE slugs(name, age);')

  sql = "INSERT INTO slugs (name, age) VALUES (?, ?)"

  dbh.prepare(sql) do |sth|
    1.upto(43) { |i| sth.execute("Slug #{i}", "#{i*3}") }
  end

  dbh.select_all('SELECT * FROM slugs;') do |row|
    puts "Hello, #{row[0]!"
  end

  sth = dbh.prepare("SELECT * FROM slugs WHERE name='Slug 1';")
  sth.execute

  while row = sth.fetch do
    puts row
  end

end
```

To start up DBI, call `connect`, feeding it the database driver, database, and host to use, and open a block. This block is provided the database handle as a parameter

(dbh), which it can then operate on. First, we create a table to work with using the do method; the do method immediately executes a SQL statement. Next, we build a SQL statement and prepare it using the obviously named prepare statement. This will allow us to execute the same SQL statement over and over again quickly, easily, and efficiently using the execute method (as I've done here when inserting the data); another upside is that you can have it use placeholders as I've done here using question marks. This lets you substitute values in each time the statement is executed; for example, here I substitute in the slug's name and age, which changes each time the statement is called. Next, you can see the select_all method being used, which will return every row from a SELECT statement and feed it to a block; there is also a select_one method for those times when you really just need one row. You can also prepare, execute, and fetch results if that's your flavor; that method is shown here using the prepare and execute combination with a while loop to fetch the results. Finally, when the block exits, the client disconnects. If you opt to not use the block form of this code, you will need to call disconnect to disconnect from the server.

ActiveRecord If you have any experience with Ruby on Rails at all, you've surely heard of its great ORM (Object Relational Mapping) package ActiveRecord, but did you know you can use ActiveRecord without Rails? It's really quite simple once you see it in action. You first need to make sure you have it installed typing the following in a Ruby file and running it.

```
require 'rubygems'
require 'activerecord'
```

If you get an error, then make sure you have installed RubyGems. If you don't, install it. If you do have RubyGems installed and lack ActiveRecord, then enter gem install activerecord in a console to install ActiveRecord.

To get ActiveRecord going in your application, you essentially you import the gems library and ActiveRecord, construct a configuration that would normally live in a Rails database configuration file (i.e., config/database.yml), define your ActiveRecord classes, then call methods on them. For example, let's create a database in your favorite database server (e.g., mySQL, PostgreSQL, and so on) and a table named people that has three fields: name, job, and location. Insert a few rows and let's map that into an ActiveRecord application.

```
require 'rubygems'
require_gem 'activerecord'

ActiveRecord::Base.establish_connection(
  :adapter  => "mysql",
  :host => "localhost",
  :database =>  "test",
  :username =>  "root",
  :password =>  ""
)

class Person < ActiveRecord::Base
end

mypeople = Person.find(:all)
puts "Total length: #{mypeople.length}"
```

```
→ Total length: 14

mypeople.each do |personobj|
  puts "Hello #{personobj.name}, #{personobj.job}."
end

→ Hello Imogen, child psychiatrist and superhero.
```

As you can see, rather than having a pre-constructed configuration, the `establish_connection` method is fed a longer set of parameters. Then, a class that inherits from `ActiveRecord::Base` is created that is the singular form of the table name we made earlier, hence `Person` for the table `people` (this is part of the magic of ActiveRecord); this maps the table to a class that you can instantiate and use to manipulate rows in the database (check out the ActiveRecord documentation at http://api.rubyonrails.org/ for more information on what you can do with ActiveRecord objects).

This Chapter

You learned about Ruby's networking faculties. You learned...

- about Ruby's basic networking capabilities with TCPSocket/TCPServer.
- how to use HTTP networking with Ruby.
- consume web services with Ruby.
- about DRb and Ruby's distributed computing capabilities.
- how to connect to a database with Ruby.

6

It's a Library!

As you've seen, Ruby provides a number of built-in classes for you to access; it has a huge library of functions to cover anything from simple text input and output to networking to multimedia (not counting the boat load of third party libraries floating about). We've taken a look at some of the more "targeted" classes (i.e., win32, networking, and so on), but I want to take some time to devote a really quick overview to a few classes that are more miscellaneous in nature.

STRING MANIPULATION

Ruby's string support easily rivals that of Perl and other "power" languages. As a matter of fact, I've heard that when Larry Wall goes home at night he secretly moon lights as a Ruby programmer. Don't tell anyone I told you. There are two ways to manipulate strings: instance methods on string objects and regular expressions.

Instance Methods

Ruby strings are, of course, objects, and as such, offer methods to manipulate themselves in a variety of ways. First we will look at the simplest manipulation of a string: the splice. The first way I'd like to show splicing is the splice operator; this operator is used just like the array operator (i.e. it uses the `object[index]` form to reference and set the value of elements) with a few little string specific extras. For example:

```
the_alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
puts the_alphabet[0..2]
→  ABC
puts the_alphabet["HIJ"]
→  HIJ
puts the_alphabet[1,2]
→  BC
```

```
the_alphabet[1] = "!"
puts the_alphabet
→  A!CDEFGHIJKLMNOPQRSTUVWXYZ
```

As an alternative, you can use the `slice` method to grab elements in a similar manner:

```
puts the_alphabet.slice(0, 3)
→  ABC

puts the_alphabet.slice(0..5)
→  ABCDEF
```

Once you have your string data sliced and diced just the way you want it, you can manipulate it in a number of ways; these methods include ways to change case, edit certain content, or generally finagle with the string. Here are some examples:

```
a_str = "I likE To MesS with STRINGS!"

a_str.downcase
→  i like to mess with strings!

a_str.upcase
→  I LIKE TO MESS WITH STRINGS!

a_str.swapcase
→  i LIKe tO mESs WITH strings!

a_str.capitalize
→  I like to mess with strings!

a_str.capitalize!
puts a_str
→  I like to mess with strings!

a_str.squeeze
→  I like to mes with strings!
```

The names of these methods should make their usage obvious, with the exception of `squeeze` which will remove sets of the same character and replace them with a single instance of that character (i.e. "`mess`" is now "`mes`"); each of these methods also offers an "in place" version as demonstrated by `capitalize!`.

The `insert` method allows you to insert a string at a given index; be careful because this method will modify the string in place (I'm not sure why this method lacks a `!`, but it does):

```
into = "12345"
into.insert(2, "LETTERS!")
puts into
→  12LETTERS!345
```

On the other hand, you can use the `delete` method to remove characters or a range of characters from a string:

```
gone = "got gone fool!"
gone.delete("o", "r-v")
puts gone
→  g gne fl!
```

Placing a hyphen between two letters will tell Ruby to create an inclusive (i.e. also include the lower and upper limit characters, such as `r` and `v` in the example) range of characters. You can use this technique with a number of methods, but it is exceptionally useful with delete.

Removing the last character from a string can be really annoying and much longer that it should be in some languages (I'm looking right at you PHP); fortunately, like any good language that has strings, Ruby offers `chomp` and `chop`:

```
eat_me = "A lot of letters\n"
eat_me.chomp
→  A lot of letters

eat_me.chop
→  A lot of letter

eat_me.chomp("ter")
→  A lot of let
```

The `chomp` method will remove the record separator (stored in the global object `$/`) from the end of a string; if `$/` hasn't been changed from the default, this means it will remove `\n`, `\r\n`, and `\r` (essentially any sort of newline character). You can also specify what to `chomp` by passing it in as a parameter as in the last example. The `chop` method will simply chop off the last character regardless of what it is (note: it will treat \r\n as one character).

If you simply need to remove the whitespace from the beginning and/or end of a string, then use one of the strip methods: `lstrip`, `rstrip`, or simply `strip`. For example:

```
stripper = "    La di da!    \t"
puts "[" + stripper.lstrip + "]"
→ [La di da!         ]

puts "[" + stripper.rstrip + "]"
→ [     La di da!]

puts "[" + stripper.strip + "]"
→ [La di da!]
```

If it isn't obvious from the examples, the `lstrip` and `rstrip` methods strip off any sort of whitespace (i.e. spaces, tabs, newlines, etc.) from a string on the left or right side, respectively. The `strip` method strips whitespace from both sides.

The `split` method will probably be one of your most often used string methods; it breaks a string into substrings based on a specified delimiter. That's a bit of a mouth full, so let me show you an example:

```
breakable = "break,it,down,fool!"
breakable.split(",")
→ ["break", "it", "down", "fool!"]
```

The `split` method breaks a string into pieces; these pieces are formed by breaking the string at a delimiter, removing that delimiter, and placing the remaining pieces into an

array. This allows you to then use an iterating loop to go over the collection and do various operations on the data within.

But what if you need to do an operation that isn't allowed on a string? For example, you want to do some arithmetic on some numbers you read in from a text file, but text from files is always read in as strings. Ruby offers the `to_f` and `to_i` methods to allow just that:

```
numerical_string = "1234"
numerical_string + 6
→ ! TypeError
numerical_string.to_f + 6.7
→ 1240.7
```

It's important to remember that all data read from sockets, files, and users will be strings; if you plan on doing any sort of math with this data, you must run it through one of these methods first.

Regular Expressions

I mentioned regular expressions in passing earlier so the appearance of the syntax wouldn't confuse you if you were looking at Ruby examples elsewhere; now I'd like to take some time to discuss what regular expressions are and how incredibly useful they can be.

Firstly, let me say that I am not going to give you a grand tour of regular expressions, nor will I bore you with a biography of his third cousin, Samuel del Fuega IV. I will, however, provide some very rudimentary examples whereby I can show the usage of these methods *and,* for the special price of *nothing at all*, I am providing an appendix with URLs of pages to visit if you'd like to find out more about the dark craft of regular expressions (it's in Appendix A). Aren't you excited? Let's proceed.

A regular expression (sometimes called a regexp or regex) is a string that describes or matches a set of other strings, according to a set of syntax rules. Their uses range from simple searches to full string transformation involving searching, replacing, and shifting of data. They're a delightful addition to any programming language, a useful tool for string manipulation, and make a delectable topping for any dessert.

Regular expressions have a rich syntax which allows you to do a number of things, but for the sake of brevity and sanity of scope, I will use very simple regular expressions. You will need to go and read an introduction to regular expressions to fully and easily understand what is going on, but perhaps you can gather what's going on without it.

Let's begin by looking at the simplest use of regular expressions: matching. You can search within a string using regular expressions easily; let's say you were wanting to see if a string was a substring of another string. You could do this:

```
matchmaker = "I'm a cowboy, baby!"
matchmaker =~ /cow/
→ 6
matchmaker.match("cow")
→ #<MatchData:0x61d3120>
```

Using the =~ will return the index of the first match of the pattern, but using the `match` method of a string, you can get a `MatchData` object which gives you a number of options for accessing the matches. For example:

```
my_message = "Hello there!"
my_message.match(/(..)(..)/).captures
→ ["He", "ll"]
```

Using the captures method, you can grab the matches for each expression; the `to_a` method will offer you similar output but will also tag on the fully matched string. That was sort of a silly example, so let's look at a more plausible example. Say you wanted to grab the text between two delimiters; here is one way to do it:

```
html_element = "<html>"
my_element_text = html_element.match("(<)(.*)(>)").captures
puts my_element_text[1]
→ html
```

If you're the curious type, you can look in the Ruby API Documentation for more information on the `MatchData` class.

If you'd like to kick out the middle man and simply get an array of matches back from the method, you can use the `scan` method:

```
my_message = "Hello there!"
my_message.scan(/(..)(..)/)
→ [["He", "ll"], ["o ", "th"], ["er", "e!"]]
```

Other than the difference in return type between the two methods, `match` and `scan` also differ in "greediness." A "greedy" regular expression or method will match every occurrence of a pattern rather than just matching one. The `scan` method is greedy; it will match every occurrence of a pattern in a string (note: you can make `match` greedy using a certain kind of regular expression).

Another usage of regular expressions is substitution; substitution using regular expressions is very flexible if you use the right combination of potent regular expressions. Again, because I do not discuss advanced regular expressions, the true usefulness of this technique won't really register, but I hope that you will take a serious look at regular expressions and use them to your advantage. To substitute using a regular expression, you use the `sub` or `gsub` method of a string instance:

```
go_away = "My name is Freak Nasty!"
go_away.sub("y", "o")
→ Mo name is Freak Nasty!
```

```
go_away.gsub("y", "o")
→ Mo name is Freak Nasto!
go_away.sub("Freak Nasty", "Fred Jones")
→ My name is Fred Jones!
```

The `sub` method will only replace the first match for the pattern, but the `gsub` method is greedy (I'm sure the `g` didn't give it away) and will replace every match. Again, the greediness of each method can be gaged by the usage of certain regular expression constructs. The more powerful regular expressions you learn, the more you can do with them; remember to check out Appendix A for more information.

DATE/TIME

Have you ever found yourself in the deli, browsing the ham and other delectable meat products, and then realized that you left your calendar in Bermuda? That happened to me just last millenium, and let me tell you, I felt vulnerable. No calendar means, no days. No days means no nights, which means I could *die*. Fortunately, Ruby provided me with a fairly useful date and time library to use until I could get my half brother in law's pet monkey's trainer's dog to mail my calendar back yesterday.

There are three date and time classes in the Ruby library: `Date`, `Time`, and `DateTime`. I heard a rumor that `Date` and `Time` hooked it up around version 1.4 and got `DateTime` about 9 months later, but then again, this source also told me that Rails was a gentlemen's club for Ruby programmers.

Dates

The first class I'd like to cover is `Date`; this class simply exposes an interface to store, manipulate, and compare dates in a Ruby application.

```
mydate = Date.new(1999, 6, 4)                → 1999-06-04
mydatejd = Date.jd(2451334)                  → 1999-06-04
mydateord = Date.ordinal(1999, 155)          → 1999-06-04
mydatecom = Date.commercial(1999, 22, 5)     → 1999-06-04
Date.jd_to_civil(2451334)                    → [1999,6,4]
Date.jd_to_civil(2451334)                    → [1999,6,4]
mydatep = Date.parse("1999-06-04")           → 1999-06-04
```

As you can see, creating a `Date` instance is rather simple in its literal form; simply call new (or the `civil` method; the two are synonyms), providing a date as the following parameters. This method uses the date form we usually see, but `Date` also supports other date forms. The `jd` method allows you to create a `Date` instance based on a Julian day number; the `ordinal` method creates a `Date` object based on a provided Ordinal date, or a date created providing the year and day number; `commercial` creates a `Date` object from the provided Commercial date, or a date created by providing the year, week number, and day number. Methods are provided to convert between these date forms also (e.g., `commercial_to_jd`, `jd_to_civil`, and so on).

These all work well enough, but notice the last example using the `parse` method; this allows you to parse strings into `Date` objects. I find this is the most intuitive way of creating `Date` objects second to using the `new` method.

You can also test input with various class methods, and, once you have a `Date` object, get all sorts of information from it with a few instance methods.

```
Date.valid_jd?(3829)                    → 3829
Date.valid_civil?(1999, 13, 43)         → nil
mydate.mon                              → 6
mydate.yday                             → 155
mydate.day                              → 4
```

As you can see, you can test its validity in a certain format, and, using instance methods, convert between the different formats. You can also get various components of the date, such as the year day (`yday`), month (`mon`), and so on. You can also compare and manipulate dates using standard operators.

```
date1 = Date.new(1985, 3, 18)
date2 = Date.new(1985, 5, 5)

date1 < date2                           → true
date1 == date2                          → false
date3 = date1
date1 == date3                          → true

date1 << 3                              → 1984-12-18
date2 >> 5                              → 1985-10-05
```

As you can see, comparing dates is just like comparing a standard numerical value or something similar; a date that comes before another date is judged to be "less than"; a date that comes after is judged to be "greater than." You can also use the >> and << operator to add or subtract months (see the last two examples). Now that you have a familiarity with the `Date` class, let's move on to the `Time` class.

Times

The `Time` class is very similar in function to the `Date` class, except it concentrates on times and timestamps rather than simply dates. Much like the `Date` class, various constructors are available.

```
rightnow = Time.new
       → Sun Sep 10 21:36:15 Eastern Daylight Time 2006
Time.at(934934932)
       → Tue Aug 17 20:08:52 Eastern Daylight Time 1999
Time.local(2000,"jan",1,20,15,1)
       → Sat Jan 01 20:15:01 Eastern Standard Time 2000
Time.utc(2006, 05, 21, 5, 13)
       → Sun May 21 05:13:00 UTC 2006
```

```
Time.parse("Tue Jun 13 14:15:01 Eastern Standard Time 2005")
       → Tue Jun 13 14:15:01 Eastern Daylight Time 2006
```

As you can see, you can create a new `Time` object that holds the values for the current time and timezone by simply calling `new` (or optionally, `now`; they do the same thing). If you require a certain time, you can use `at`, which operates on epoch time (i.e., seconds from January 1st, 1970); you can also use the `utc` and `gm` methods to create times based on those timeszones and the provided parameters (or the `local` method to use the current local timezone). You can, just like `Date`, use the `parse` method to parse a timestamp into a `Time` object.

The `Time` class also offers a few instance methods that allow you to get portions of the object's value, convert the value, and output the value in other formats.

```
rightnow = Time.new
       → Sun Sep 10 21:42:30 Eastern Daylight Time 2006
rightnow.hour
       → 21
rightnow.mon
       → 9
rightnow.day
       → 10
rightnow.to_i
       → 1158543750
rightnow.to_s
       → Sun Sep 17 21:42:30 Eastern Daylight Time 2006
```

As you can see, the methods for Time are very similar to `Date` with regards to getting portions of the value, and also notice that you can convert the `Time` objec to other classes, such as a `Fixnum`.

Let's concentrate on one instance method for a moment; the `strftime` method is a very useful method that allows you output a timestamp in the format of your choice by providing you with a formatting interface. This interface acts very, very similarly to `printf` in C++; it uses delimiters like `%f` to indicate the placement of values in the output string. Here are a few examples:

```
rightnow = Time.now
rightnow.strftime("%m/%d/%Y")
       → 09/10/2006
rightnow.strftime("%I:%M%p")
       → 09:13PM
rightnow.strftime("The %dth of %B in '%y")
       → The 17th of September in '06
rightnow.strftime("%x")
       → 09/17/06
```

The `strftime` method is one of the most complex in the `Time` module; check out the `Time` class's documentation at http://www.ruby-doc.org/core/classes/Time.html if you'd like more information about `strftime` and what you can do with it.

Dates and Times

The `DateTime` class combines the previous two classes into one convenient yet slightly less efficient class. The `DateTime` class is really just a subclass of `Date` with some time functionality slapped in there for good measure; it's a fine endeavour to be sure but not really worth the time if you ask me. Even so, it has some interesting functionality.

```
rightnow = DateTime.now
        ⟶ 2006-09-10T21:56:45-0400
maytime = DateTime.civil(2006, 5, 6)
        ⟶ 2003-05-06T00:00:00Z
parsed = DateTime.parse("2006-07-03T11:53:02-0400")
        ⟶ 2006-07-03T11:53:02-0400

parsed.hour                                    ⟶ 11
parsed.day                                     ⟶ 3
parsed.year                                    ⟶ 2006
```

As you can see it works very similarly to the `Date` class; you can construct using the various date formats or parse a date/time string. Also notice that like the `Date` and `Time` classes, you can query various parts of the value inside the object.

You may be scratching your head right now asking which one you should use and when. Personally, I would never use `DateTime`, but rather `Time` or `Date` if at all possible. Sometimes this is unavoidable, but be aware that using just `Date` or `Time` in lieu of `DateTime` yieldds approximately 800% better performance. Sometimes performance, like size and Poland, does matter.

HASHING AND CRYPTOGRAPHY

Sometimes you simply don't want people to be able to see your data transparently; I mean, maybe you've got this rash that you don't want people to know about. Or maybe there's something you just want to forget, so you hash it and never worry about it again. Forunately for me...er, I mean you...Ruby comes stock with a neat little hash library and has a gem that can be installed to offer cryptography.

Hashing

Think of hashing as one-way encryption; hashes are encrypted strings that are derived from a stream of data. Typical uses include password verification (i.e., you store the hash in the database, then test user input by hashing it and seeing if the hashes match) and file verification (i.e., two of the same file should have the same hash). Ruby offers the two most common hash types, MD5 and SHA-1, as built-in modules.

MD5 MD5 is the most widely used cryptographic hash function around; it was invented in 1994 as a successor to MD4 by Ronald Rivest, a professor at MIT. It's fallen out of mainline use as a secure hash function because of vulnerabilities that have been found, but it's still useful for matching values and such. Ruby's MD5 functionality isn't quite as easy as something like PHP (i.e., `md5('your data');`), but it's still usable and friendly enough.

```
require 'digest/md5'
md5 = Digest::MD5.digest('I have a fine vest!')
       → sXm(1r\371\353\027\367\235u!\266\001\262

md5 = Digest::MD5.hexdigest('I have a fine vest!')
       → 73586d283172f9eb17f79d7521b601b2
```

The MD5 class offers two methods for getting a hash digest; the first is simply the `digest` method. This returns a pretty unsafe byte stream of the hash digest; I say unsafe because you could not embed this in something like an XML or HTML (or some databases) and expect it to behave properly. A better choice for these situations would be the `hexdigest` method (second example); this runs the results of the hash through a base64 hex algorithm, which is fancy talk for a method that makes it more friendly.

SHA-1 The SHA-1 hash algorithm is far more secure than MD5; though still not the most secure (i.e., exploits reportedly exist for it), it should work for most situations. It is widely used as a hashing algorithm in many secure contexts, such as in packages like TLS, SSL, PGP, SSH, and IPSec. Ruby offers the same interface to the SHA-1 algorithm as it does the MD5 algorithm.

```
require 'digest/sha1'
sha1 = Digest::SHA1.digest('I have a fine vest!')
       → \225J{{\233\025\236\273\344\003X\233\33 [...]

sha1 = Digest::SHA1.hexdigest('I have a fine vest!')
       → 954a7b7b9b159ebbe403589bdaa8f981003a2fbc
```

As you can see, it functions exactly the same as the `MD5` class, except you get a stronger hash. Now let's get away from hashes and take a look at cryptography.

Cryptography

Ruby does not have cryptographic capabilities built-in, so you have to resort to installing a gem. I guess technically this isn't a Ruby built-in library, but it's important enough to warrant a short mention. The third-party crypt library available at http://crypt.rubyforge.org is a pure Ruby cryptography library. You can install it by issuing the gem install crypt command to install its gem; look in Appendix A for a link on how to install and setup RubyGems if your Ruby installation doesn't have them already.

The crypt library offers four encryption algorithms: Blowfish, GOST, IDEA, and Rijndael. Fortunately, the interface for each on is relatively the same. Let's take a look at an example using the Blowfish algorithm from their documentation.

```
require 'crypt/blowfish'
blowfish = Crypt::Blowfish.new("A key up to 56 bytes long")
plainBlock = "ABCD1234"
encryptedBlock = blowfish.encrypt_block(plainBlock)
       → \267Z\347\331~\344\211\250
decryptedBlock = blowfish.decrypt_block(encryptedBlock)
       → ABCD1234
```

This is one of the easiest cryptography libraries out there; simply feed it a key in the constructor, call the `encrypt_block` method to encrypt the data, and then `decrypt_block` to decrypt it. Since the developers went to great lengths to keep the API basically the same for all the algorithms, you can simply subsitute the other algorithm names in place of Blowfish to get them working (i.e., put Rijndael in place of Blowfish and it should work just the same). There are other restrictions on key length and such, along with other methods and functions you can use. Check out http://crypt.rubyforge.org/ to learn more.

UNIT TESTING

Test Driven Development is the new hotness, especially in the Ruby development world. I'm sure the Ruby core team took this into account when they built a unit testing framework into the standard library of the language: `Test::Unit`. Ruby's unit testing framework is excellent (and has been made better and/or replaced and improved by other frameworks) yet very simple to use.

The basic premise of testing is to make sure that your code is behaving correctly in as many contexts as you can simulate programmatically. This might sound stupid, but trust me: you'll catch twice as many bugs using a unit test as you will by just playing around with your application because you know how it is supposed to operate but the computer doesn't. It has no "developer's discrimination" when it comes to using your application. You know what I'm talking about; no one wants their application to break, so they unconsciously tip-toe around what might become a bug. I do it all the time. That's why I use testing.

Ruby's unit testing framework provides a simple interface for performing tests. Let's say you wanted to test your class that stores MAC addresses for your locally networked client applications.

```
class MacAddr
  def to_s
    return @mac_parts.join(":")
  end

  def initialize(mac)
    if mac.length < 17
      fail "MAC is too short; make sure colons are in place"
    end

    @mac_parts = mac.split(':')
  end

  def [](index)
    return @mac_parts[index]
```

```
  end
end
```

This simple class has three methods: `to_s`, `intitialize`, and an index method (`[]`). The constructor, `initialize`, takes a string with a MAC address in it. If it is the wrong length, an exception is thrown; otherwise it's broken up on the colons (part of the standard MAC notation) and placed in an instance variable. The `to_s` method joins this array together with colons and returns it. The index method (`[]`) will return the requested index from the MAC address array (`@mac_parts`). Now that we have something to work with, let's build some tests.

Tests with Test::Unit center around inheriting from the Test::Unit::TestCase for each test case. So, if we were to create a test case for our MAC address class, we would do something like the following.

```
require 'test/unit'

class TestMac < Test::Unit::TestCase
end
```

Simple enough, right? Now that you have a test case class, you need to fill in tests. Tests could be written as a bunch of ugly if statements, but unit testing frameworks do their best to get away from that by providing you with assertions (i.e., wrappers for those conditional statements that hook into the framework in a meaningful way). The first type of assertion we'll look at are the equality tests. There are two equality assertions: assert_equal and assert_not_equal. Let's create a couple of those tests now in the same file we created the class.

```
require 'test/unit'

class TestMac < Test::Unit::TestCase
  def test_tos
    assert_equal("FF:FF:FF:FF:FF:FF",
        MacAddr.new("FF:FF:FF:FF:FF:FF").to_s)
    assert_not_equal("FF:00:FF:00:FF:FF",
        MacAddr.new("FF:FF:FF:FF:FF:FF").to_s)
  end
end
```

We've basically created two tests. The first makes sure that if we give it a MAC address, it will return it properly when using to_s. The second one does the same thing, but in an inverse conditional, we feed it a different value to make sure they're not equal. Upon running the tests, we should hopefully see success.

```
Loaded suite unit_test
Started
.
Finished in 0.0 seconds.

1 tests, 2 assertions, 0 failures, 0 errors
```

And we do. Excellent. Now let's take a look at another type of assertion: nil assertions. These assertions, assert_nil and assert_not_nil, do basically the same thing as if you did an assert_equal and tested for nil. Let's create a test using assert_not_nil.

```
require 'test/unit'

class TestMac < Test::Unit::TestCase
  def test_tos
    assert_equal("FF:FF:FF:FF:FF:FF",
        MacAddr.new("FF:FF:FF:FF:FF:FF").to_s)
    assert_not_equal("FF:00:FF:00:FF:FF",
        MacAddr.new("FF:FF:FF:FF:FF:FF").to_s)
    assert_not_nil(MacAddr.new("FF:AE:F0:06:05:33"))
  end
end
```

Again, upon running the tests, we should hopefully see a successful run without and errors or failures.

```
Loaded suite unit_test
Started
.
Finished in 0.0 seconds.

1 tests, 3 assertions, 0 failures, 0 errors
```

And we do. Great! Now, let's look at one final type of assertion that deals with exceptions. Ruby's testing framework allows you not only to test the return value of units of code, but also to test whether they raise exceptions or not. We told our class to raise an exception if the MAC address isn't the right length, so let's write a test to test that.

```
require 'test/unit'

class TestMac < Test::Unit::TestCase
  def test_tos
    assert_equal("FF:FF:FF:FF:FF:FF",
        MacAddr.new("FF:FF:FF:FF:FF:FF").to_s)
    assert_not_equal("FF:00:FF:00:FF:FF",
        MacAddr.new("FF:FF:FF:FF:FF:FF").to_s)
    assert_not_nil(MacAddr.new("FF:AE:F0:06:05:33"))
    assert_raise RuntimeError do
      MacAddr.new("AA:FF:AA:FF:AA:FF:AA:FF:AA")
    end
  end
end
```

Now, if we run these tests again, we'll hopefully see another wonderfully successful run.

```
Loaded suite unit_test
Started
F
Finished in 0.015 seconds.

  1) Failure:
test_tos(TestMac) [test21.rb:27]:
<RuntimeError> exception expected but none was thrown.

1 tests, 4 assertions, 1 failures, 0 errors
```

Oops! If you look at our constructor, we merely test if the MAC address is too short. Let's switch that < to a != so that it catches it whether it's too short or too long and try these tests again.

```
Loaded suite unit_test
Started
.
Finished in 0.0 seconds.

1 tests, 4 assertions, 0 failures, 0 errors
```

Great! We've built a small test suite for our class. Of course, this is just one class in a whole application, and each class should have its own test suite. As test suites grow, you'll inevitably want to break them into separate files, since you wouldn't want to keep all 1,200 of your test cases for your breakdancing panda screen saver in one file. Fortunately, Test::Unit is smart enough to pick up on numerous test files being included into one test run. This means you could do something like the following without any problems.

```
require 'test/unit'
require 'pandatest'
require 'breakdancetest'
require 'breakdancingpandatest'
require 'somewhatperipheralelementstest'
require 'somewhatperipheralelementsbestfriendsunclestest'
```

I've just given you a basic rundown of testing; I'm providing a list of links in Appendix A that can take you deeper into Test Driven Development and testing with Ruby. Also be sure to check out the Test::Unit documentation at http://www.ruby-doc.org/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit.html to find about other available assertions (there are a few I don't cover here because they're not very common).

PERORATION

I hope you've enjoyed this journey through Ruby as much as I have.

Please do check out my Rails book at http://www.rubyonrailsbook.com/ and my blog at http://www.mrneighborly.com/. Feel free to drop me an e-mail or comment there with any questions or comments.

Enjoy your newfound avocation which will hopefully turn in an occupation for you rather than an exasperation over any sort of snag or complication that you may encounter in the course of your application of the principles in this publication.

That rhymed a lot. There's a reason they call me Kill Masta Neighborly. Fo' rizzle.

Peace out,

Mr. Neighborly

Appendix A

Links and the Like

THE RUBY LANGUAGE

Ruby Language main site http://www.ruby-lang.org/
The main site for the Ruby language; get downloads and information here

RubyForge http://www.rubyforge.org/
Looking for a Ruby library or application? Chances are you can find it here.

RubyCorner http://www.rubycorner.com/
Ruby blog aggregator; blogs are added by their owners, so there are currently 200+.

Ruby Central http://www.rubycentral.com/
David Black's excellent Ruby organization; you can find information and links here.

PlanetRuby http://planetruby.0x42.net/
Another Ruby blog aggregator; blogs are selected by the site owner.

DOCUMENTATION

RubyDoc http://www.rubymanual.org/
Points to a number of Ruby documentation sources, including the Ruby API docs.

Why's Poignant Guide to Ruby http://www.poignantguide.net/ruby/
Another Ruby book; if you found my writing too dull, perhaps this is more your flavor.

RubyManual http://www.rubymanual.org/
PHP Manual style documentation that allows user comments.

RubyGems Installation http://rubygems.org/read/chapter/3
Article on how to install and setup RubyGems

REGULAR EXPRESSIONS

Regular Expressions Tutorial http://www.regular-expressions.info/
The tutorial I learned from; it's not the best (from what I hear) but it worked for me.

Regular Expressions Library http://www.regexlib.com/
A large library of user-submitted regexen that do anything you can think of.

Wikipedia Article http://en.wikipedia.org/wiki/Regex
The Wikipedia article for regular expressions; as always, pretty useful information.

UNIX INFORMATION

The Filesystem http://www.unix.org.ua/orelly/networking/puis/ch05_01.htm
Excellent source for information on UNIX filesystems and their metadata.
Linux Documentation Project http://www.tldp.org/
The source for Linux documentation.
General Linux command list http://www.linuxdevcenter.com/linux/cmd/
Huge list of Linux commands; useful for those mystery commands or finding new ones.
A Beginner's Tutorial for Linux http://www.linux-tutorial.info/
Very basic information about Linux.

WEB SERVICES

Ruby Web Services h ttp://www.devx.com/enterprise/Article/28101
Great article that discusses building web services using Ruby.
Soap4r Homepage http://dev.ctor.org/soap4r/
The official homepage of the official SOAP client for Ruby.
xmlrpc4r - XML-RPC for Ruby http://www.fantasy-coders.de/ruby/xmlrpc4r/
Provides an XML-RPC server in Ruby.

DISTRIBUTED RUBY

DRb Rdocs http://www.ruby-doc.org/stdlib/libdoc/drb/rdoc/index.html
The official documentation for DRb.
DRb Tutorial http://www.chadfowler.com/ruby/drb.html
Chad Fowler's excellent DRb tutorial.
Segment7 DRb Page http://segment7.net/projects/ruby/drb/
More DRb information from Segment7.

DATABASES

Official DBI website http://ruby-dbi.rubyforge.org/
Official homepage of the Ruby DBI package
Ruby and the mySQL package http://www.kitebird.com/articles/ruby-mysql.html
A tutorial on using the mysql package and Ruby
ActiveRecord Docs http://api.rubyonrails.org/classes/ActiveRecord/Base.html
The official Ruby on Rails rdocs for ActiveRecord; go here for more info on how to use it

UNIT TESTING

ZenTest http://rubyforge.org/projects/zentest/
Testing, on steroids. If you get bored with Test::Unit, check this out.

Why and How: Ruby (and Rails) Unit Testing
http://glu.ttono.us/articles/2005/10/30/why-and-how-ruby-and-rails-unit-testing
An excellent article by Kevin Clark on unit testing with Ruby (and Rails).

Appendix B

High Performance Ruby with C/C++

There is a lot of talk these days about extending Ruby with C and C++, and how those extensions can greatly improve Ruby's performance. No package has done more to promote this than RubyInline, a port of Perl/Python/whatever else's inline C module.

Many developers question a mindset that says "Rewrite it in C" if a part of an application is too slow in the current language; many developers from other camps feel that this unnecessary step is a weakness of Ruby and that Ruby should be faster rather than having to use C/C++. I don't see it that way. I think Ruby's nature and the ability for developers to be able to directly write C code inline is great. Telling developers to rewrite it in C for speed isn't any different than what many language interpreters do. Look at Python's interpreter; anything that needs any degree of speed has been rewritten in C rather than solid Python. It's a fact of dynamic languages that they will occasionally need these speed boosts.

I'm not going to go into a huge spill about RubyInline or how to use it; it's too big of a subject. This is one of those things that could warrant its own book if anyone would write it. There are so many applications for this technology, and people are beginning to take notice. I typically wouldn't write about subjects like this, since I am not a C/C++ master (I have experience, but I prefer to enjoy programming rather than abuse myself), but it's becoming an increasingly important subject.

Using RubyInline is much easier than trying to write a C extension from scratch. The basic premise of the package is to allow developers to embed C/C++ code directly into the Ruby code for their application, which will then be compiled and dynamically executed as the application runs. The idea is that you profile your application using something like the Ruby profiler and figure out where your bottlenecks are. For example, Pat Eyler wrote about rewriting parts of a prime number calculator in C; he used Ruby profiler to figure out that a block of code that he was passing was causing a massive slow down. You should then take this data and figure out what you could rewrite in C, and *only rewrite that* (there's no reason to go

nuts and rewrite a lot of your application in C just to get some false illusion of perfomance at the cost of maintainability or readability). In most cases, rewriting these little parts make your performance numbers an order of magnitude better. It's really a neat concept, and incredibly helpful for adding a speed to your application.

Now, let's take a look at how this plays out when you actually get to using it. Install RubyInline using the installation instructions on ZenSpider's RubyInline page at http://www.zenspider.com/ZSS/Products/RubyInline/, and let's take a look at an example from the home page.

```
require 'inline'

class MyTest
  inline(:C) do |builder|
    builder.include '<iostream>'
    builder.add_compile_flags '-x c++', '-lstdc++'
    builder.c '

      void hello(int i) {
        while (i-- > 0) {
          std::cout << "hello" << std::endl;
        }
      }'
  end
end

t = MyTest.new()
t.hello(3)
```

The above code will print "hello" three times. This is, of course, a very contrived example, but I put it here to first explain how RubyInline works. As you can see, you simply open a block with the `inline` method to begin the process; the `:C` parameter isn't required if you are using C, since that is what RubyInline is designed for. Next, you can see that `iostream` is included using the `include` method; notice you can also add compile flags to the command line using the `add_compile_flags` method. This is useful if there are libraries you need to add (like `stdc++` in this example). Next, use the `c` method and provide a string of C code. The code provided here will be compiled and then can be called just like a Ruby method (which is demonstrated in our call to `hello` in the main loop).

I'll leave you to your imagination to see how you can use this library to improve performance in your own applications. Poke around the ruby-talk mailing list and Google to get some good ideas; the RubyInline homepage has more information, too, if you're interested in putting this library to work.

LINKS

RubyInline homepage
http://www.zenspider.com/ZSS/Products/RubyInline/
Pat Eyler's RubyInline experience
http://on-ruby.blogspot.com/2006/07/rubyinline-making-making-things-faster.html

Alphabetical Index