

The
Pragmatic
Programmers

Rails for .NET Developers



Jeff Cohen and Brian Eng

Edited by Susannah Davidson Pfalzer

The Facets  of Ruby Series

Extracted from:

Rails for .NET Developers

This PDF file contains pages extracted from Rails for .NET Developers, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2009 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

<http://www.pragprog.com>

Copyright © 2008 Jeff Cohen and Brian Eng.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-20-4

ISBN-13: 978-1-934356-20-3

Printed on acid-free paper.

P2.0 printing, December 2008

Version: 2009-1-5

Chapter 2

Switching to Ruby

All good Rails developers share a common trait: they are also good Ruby programmers. A solid understanding of the Ruby programming language is essential to reaching those “ah-ha!” moments that will truly elevate your Rails expertise. Ruby is an object-oriented language, like C# or VB .NET, so many concepts in Ruby will be very familiar and easy to pick up.

However, because Ruby is also a scripting language and does not distinguish between a compilation phase and an execution phase, some details of Ruby semantics may not be as apparent. Ruby is also a *dynamically typed programming language*, which means any variable in memory is allowed to change its type while the program is running. This can be unsettling for those of us who are accustomed to a statically typed language like C# or VB .NET. Statically typed languages require that variables are not only declared as belonging to a specific type, but also they must remain so for the duration of the program’s lifetime. An object’s callable methods, properties, and internal field structures are immutable for the life of the object. Attempting to treat an object with the wrong type specification can be disastrous. The C# and VB .NET compiler catches type mismatch errors for us so that we’re much less likely to find a type error at runtime.

In contrast, since Ruby objects can, by design, change their list of callable methods at runtime, Ruby developers learn to rely on unit tests to not only test the type-compatibility of their code but also to ensure the correctness of all facets of a Ruby application.

But there is a lot of good news. The similarities between Ruby and the .NET languages far outweigh their differences. In this chapter, we’ll

highlight the essential elements of Ruby, often comparing them with their .NET counterparts, so that you can get up to speed with Ruby quickly and with confidence.

For a complete introduction to the Ruby language, see *Programming Ruby* [TFH05].

2.1 Ruby vs. .NET for the Impatient

If you've never written any code in Ruby, you're probably anxious to find out what's different and whether you'll have to throw out everything you already know about .NET and start over. Let's dive right in by answering some of the most frequently asked questions that .NET developers have when they start learning Rails:

- **In Ruby's object-oriented world, we work with *objects* and *methods*.** Unlike VB .NET, where some subroutines return a value (Functions) and others do not (Subs), all Ruby methods must return a value. When an explicit return statement is not used, the last-evaluated expression automatically becomes the return value.
- **Variables are not declared prior to their use.** Ruby automatically allocates memory for variables upon first use, and it also assigns their type based on inference. Like .NET, a garbage collector will reclaim memory automatically.
- **There is no distinction made between methods, properties, and fields (or “member variables”) like there is in .NET.** Ruby has only the concept of methods. However, Ruby classes do sport a convenient syntax for defining “attribute methods,” which are equivalent to defining .NET properties. `attr_reader` and `attr_accessor` automatically define methods that provide property-like access to instance variables.
- **Comments start with the hash character (#), unless the hash occurs inside a double-quoted string.** Everything after the hash is ignored. There is no multiline comment character in Ruby.
- **Ruby classes may define *instance methods* and *class methods*.** Class methods are called *static methods* in .NET.
- **Methods can be declared *public*, *protected*, or *private*, and these visibility scopes have the same meaning as they do in .NET.** There is no Ruby equivalent for “internal” or “assembly-level” visibility.

- **Instance variable names must start with an at (@) sign and are always private.** Class variables, or what we might call *static variables* in .NET, start with two at signs. The rules for memory allocation and object assignment for class variables can get pretty strange in Ruby, so we tend to avoid using them, especially since Rails provides an alternative syntax for using class variables in Rails applications. Here is an example of how to use simple instance variables in a Ruby class:

Ruby

[Download](#) ruby101/objects.rb

```
class Flight

  def prepare
    # Assign a value to an instance variable
    # This variable can be seen by all other instance methods
    # as well
    @num_engines = 2
    @num_wings = 2
  end

  def report
    puts "We have #{@num_engines} engines
        and #{@num_wings} wings."
  end
end
```

- **Ruby classes can be derived from only one base class but can “mix in” any number of *modules*.** A module in Ruby is simply a set of related methods packaged together using the module keyword instead of class.
- **There is no separate compilation step in Ruby.** If we execute this Ruby code:

Ruby

[Download](#) ruby101/objects.rb

```
class Flight

  puts "This is a flight"

  def fly_somewhere
    # ...
    # implementation code goes here
    #...
  end

end
```

you might not expect anything to happen, because it appears that we have a simple class definition. But in fact, you will see the

to_s vs. ToString()

.NET's base class, `Object`, provides a `ToString` method. All .NET classes override `ToString` so that each object can provide a string representation of themselves when needed.

Ruby's equivalent is `to_s`. Ruby objects override `to_s` to provide a suitable string representation.

In our example, we needed to emit the value of the `len` variable as a string. We used the `to_s` method to first convert the integer value to a string so we could concatenate them together.

string "This is a flight" emitted to the console! Ruby scripts are interpreted—that is, actually executed—one line at a time as the interpreter reads the file.

Let's get some Ruby code under our belt so we can examine the language in detail.

2.2 Our First Ruby Program

Start a new file named `hello.rb`, and open it with your favorite text editor. Type the following code into your file:

Ruby

[Download](#) `ruby101/hello.rb`

```
name = 'Joe'
len = name.length

puts name + " has " + len.to_s + " letters in his name."
```

We don't need to define a `Main` method anywhere. Ruby is an interpreted language, and the Ruby interpreter simply starts at the top of the file and executes each Ruby statement in order until it gets to the end of the file.

Let's walk through this short example one line at a time to understand what this code will do. We immediately see how easy it is to create new variables in Ruby. The variable `name` is assigned to the string value `Joe`. Next, we create a variable called `len` and assign it the length of the string held by `name`. Finally, we use the built-in Ruby method `puts` to print a string to standard output.

Let's see it in action. Save the file, open a Windows command prompt, and go to the directory where you saved it (in our case, it's in `c:\dev`):

```
c:\dev> ruby hello.rb
Joe has 3 characters in his name.
c:\dev>
```

Next, we'll build upon the Ruby program we've written to explore specific, practical elements of Ruby that you'll need to know as you write Ruby code. The best place to start is to learn about how we work with string variables and string literals in Ruby.

2.3 Working with String Objects

In our previous example, we used single quote marks for the string 'Joe'. In Ruby, single quotes are similar to using the `@` syntax for strings in C#. If you want to include special control characters in your string, such as a tab or carriage return, then you need to use double quotes instead. Using double quotes also allows us to use the string interpolation feature of Ruby, where we can embed any Ruby expression into a double-quoted string by surrounding the expression with `#{ }`:

Ruby

[Download](#) ruby101/hello.rb

```
puts "\t#{5*10}"
puts "Hello, #{name}. You have #{name.length} letters in your name"

c:\dev> ruby hello.rb
50
Hello, Joe. You have 3 letters in your name.
```

The `String` class provides so many helpful methods that we will just look at some of the more common methods we use in Rails applications. Let's start with how to search inside a string for a substring or pattern.

Searching and Replacing

Sometimes we want to know whether a string contains a particular substring and, if so, at which position it was found. This is done with the `index` method, which can take a literal substring to search for or a regular expression pattern.

If you've worked with regular expressions in .NET, you'll be glad to know that working with them in Ruby is significantly easier. Regular expression patterns are first-class citizens in Ruby, and denoting a regular expression in your code is as easy as working with strings. Instead of

using quotation marks, patterns are simply surrounded with forward slashes.

Regular expression patterns may look odd at first, because languages such as C# and VB .NET usually require the use of the `RegExp` class instead of being able to directly insert a pattern in your code.

In this next code sample, we use the `index` method to find out where each substring or pattern is found in our string:

Ruby

[Download](#) ruby101/hello.rb

```
word = "restaurant"
puts word.index('a')           # prints 4
puts word.index("ant")        # prints 7
puts word.index(/st.+nt$/)    # prints 2
puts word.index(/ANT$/i)      # prints 7
puts word.index('buffet')     # prints "nil"
```

That last line printed something strange, didn't it? When `index` can't find a match, it returns `nil`. In Ruby, `nil` serves the same purpose as `NULL` in SQL, `null` in C#, and `Nothing` in VB.

Sometimes we need to replace a given substring with another. The `sub` method comes to our aid here. `sub` will find the first occurrence of a substring or pattern and replace it with something else. To perform multiple replacements in the same string, we can use `gsub`:

Ruby

[Download](#) ruby101/hello.rb

```
flight = "United Airlines, Flight #312, ORD to LAX, 9:45AM to 11:45AM"
puts flight.sub('United', 'American')
puts flight.sub(/(\w+)to/, 'PDX to')
puts flight.gsub('AM', 'PM')
```

The resulting output will be as follows:

```
American Airlines, Flight #312, ORD to LAX, 9:45AM to 11:45AM
United Airlines, Flight #312, PDX to LAX, 9:45AM to 11:45AM
United Airlines, Flight #312, ORD to LAX, 9:45PM to 11:45PM
```

Notice that we did not actually modify the contents of the `flight` variable at all. This is because `sub` and `gsub` do not modify the string but simply return a new string with the desired replacements. We could have captured the result in a new variable if we wanted to work with the replaced version further. Or, we could have used the *bang* or “dangerous” versions of these methods, `sub!` and `gsub!`, to change the `flight` value in place. In Ruby parlance, “dangerous” methods are those that will modify the value of the object directly, and the Ruby convention is

to use an exclamation point in the method name to help communicate that fact.

Trimming Whitespace

Another common task when working with strings is removing leading and trailing whitespace. We could choose to combine our knowledge of `gsub` with regular expressions to do something like this:

Ruby

Download ruby101/hello.rb

```
# notice extra whitespace
flight = "  United Airlines, Flight #312, 9:45AM to 11:45AM  "
flight = flight.gsub(/\s+/, '') # remove leading whitespace
flight = flight.gsub(/\s+$/, '') # remove trailing whitespace
```

But the `String` class provides us with a much simpler alternative with the `strip` method:

Ruby

Download ruby101/hello.rb

```
# notice extra whitespace
flight = "  United Airlines, Flight #312, 9:45AM to 11:45AM  "
flight = flight.strip # removes leading and trailing whitespace
```

More likely, you'll want to use the “dangerous version” `strip!` instead: `flight.strip!` will remove leading and trailing whitespace from `flight`.

Splitting a String into Parts

We'll wrap up our quick overview of Ruby strings with the `split` method, which is handy when you need to break up a string into pieces based on a delimiter or delimiting pattern of some kind. The `split` method returns an instance of an `Array` class:

Ruby

Download ruby101/hello.rb

```
flight = "United Airlines, Flight #312, ORD to LAX, 9:45AM to 11:45AM"
info = flight.split(',')
puts info.size # prints 4
puts info     # prints the contents of the info array

info = flight.split(/\s*,\s*/)
puts info.size # prints 4
puts info     # show the info array, this time without extra spaces
```

This concludes our quick look at handling strings. You're encouraged to explore the rest of the `String` methods on your own. Next, we'll take a look at `irb`, a handy utility that makes exploring Ruby objects easier.

2.4 irb Is Your New “Immediate Mode”

Let’s learn about a useful command-line tool that comes with Ruby called `irb` (interactive Ruby). `irb` is an interactive tool for learning Ruby, and for doing all sorts of Ruby experiments, without having to endure the laborious edit/save/run cycle that we’ve been doing so far. If you’ve ever used Visual Studio’s “immediate mode” window, `irb` should feel very familiar.

To use `irb`, just open a Windows command prompt, and type `irb`. You should see something like this:

```
c:\dev> irb
irb(main):001:0>
```

Enter any Ruby expression, and `irb` will evaluate it and emit the result:

```
irb(main):001:0> 1 + 2
=> 3
irb(main):002:0>
```

`irb` prefixes the result with `=>`. In fact, `irb` always displays the resulting value of the expression or statement entered, even when it might seem unexpected; for example:

```
irb(main):002:0> puts 'Hello'
Hello
=> nil
irb(main):003:0>
```

We had indeed wanted the string `Hello` to be displayed, but why did `irb` also emit the `=> nil` after that? Like every method call in Ruby, `puts` must return a value, even if it’s `nil`, and `irb` always displays the return value of the expression or method call that we’ve entered at the `irb` prompt. Here, we’ve discovered that the return value from `puts` is `nil`.

2.5 Arrays

When we are building an application in Rails, we will find ourselves working with arrays quite a bit. Arrays in Ruby are always *dynamic* and *heterogeneous*. They are said to be dynamic because arrays have variable length. We don’t have to decide ahead of time how many elements the array will have. The array will grow automatically as elements are inserted into the array. They are said to be heterogeneous because we can store elements of any type into any element of an array.

The built-in `Array` class provides a lot of functionality right out of the box. So, let's take a look at how to get started with arrays in Ruby.

Creating an Array

We will begin by looking at a small C# console application that creates a couple of arrays and displays some information on the console:

.NET

[Download](#) ruby101/array.cs

```
List<String> colors = new List<String>();

// Add some elements
colors.Add("Purple");
colors.Add("White");

// Emit first color to console
Console.WriteLine(colors[0]); // prints "Purple"

// Create a new array, initialized with some data
List<String> sports = new List<string>
    { "Hockey", "Baseball", "Football" };

// Emit first sport to console
Console.WriteLine(sports[0]); // or, sports.First(); if using LINQ

// Combine two arrays
List<String> favorites = new List<string>();
favorites.AddRange(colors);
favorites.AddRange(sports);

// Emit the size of the combined array
Console.WriteLine(favorites.Count); // prints 5
```

Now let's see how we accomplish the same thing in Ruby. There are two easy ways to create new array objects in Ruby. The first is to explicitly declare a new `Array` instance. The second uses a special, implicit bracket syntax to create an array on the fly.

In the following example, we create a new array instance and assign it to the variable `colors`. Remember that in Ruby, we don't need to declare the type of variable to the interpreter. Instead, Ruby surmises the variable type at runtime.

Ruby

[Download](#) ruby101/array.rb

```
Line 1 colors = Array.new
- colors.push 'Purple'
- colors << 'White' # => same as .push
- puts colors[0] # => prints "Purple"
5
```

```

- sports = [ 'Hockey', 'Baseball', 'Football' ]
- puts sports.first # => prints "Hockey"
-
- favorites = colors + sports
10 puts favorites.size # prints 5

```

The `sports` array shows an alternate style of using arrays. In Ruby, arrays are denoted by square brackets. Here we've used the bracket syntax to easily populate our array with an initial list of values.

The `Array` class defines many useful methods that help us operate on arrays, and here we've used the `push` to append new elements to the array. Elements in an array can be retrieved directly by providing a zero-based index value. Therefore, `puts colors[0]` will display the first element of our array.

The `Array` class also provides many convenient “operator overloads.” We created the `favorites` array by simply “adding” the elements of the two previous arrays together. When operating on `Array` instances, the addition operator creates a new array by starting with the elements of the first array and then appending all the elements from the second array. We then captured this new array into a variable named `favorites`.

Here is an example of an array that contains both strings and numbers:

Ruby

[Download](#) `ruby101/array.rb`

```

mixed = [ "Cars", 36, 10*50, "Tables" ]
puts mixed[0] # prints "Cars"
puts mixed[2] # prints 500

```

Common Array Operations

Given an array, we can determine how many elements are in the array with either the `length` or `size` method. This is an example of how Ruby sometimes provides more than one method to perform the same function. Simply choose the method that, in your opinion, makes your code more understandable or more readable. While other languages strive for a *minimal interface*, Ruby tends toward what is sometimes called a *humane interface*, because Ruby tends to value readability and clarity of code over the principle of having as few public methods as possible.

In addition to accessing an element directly by its index, we can also obtain the first and last elements and search for elements that meet some desired criteria with the `select` and `find` methods.

Take a look at this code:

Ruby

[Download](#) ruby101/array.rb

```
sports = [ 'Hockey', 'Baseball', 'Football' ]
puts sports.first # prints "Hockey"
puts sports.last # prints "Football"

puts sports.find { |sport| sport =~ /ball/ } # prints "Baseball"
puts sports.detect { |sport| sport =~ /ball/ } # prints "Baseball"

uses_a_ball = sports.select { |sport| sport =~ /ball/ }
puts uses_a_ball.size # prints 2
```

The `select` and `find` methods are examples of methods where you supply a *block* to the method. The `select` method returns a list of all matching elements, whereas `find` will find and return only the first match.

You may have noticed that `detect` and `find` returned the same result. Again, you are free to choose whichever method gives your code the most natural feel.

Transforming an Array into a String

Earlier we learned how to use the `split` method on `String` to create an array of strings. Sometimes we want to do the inverse: given an array of strings, we need to join them together into one string. If we call the `join` method without any arguments, it will simply glue the strings together. We can also pass a string value that we would like to use as the “glue” between the elements as they are joined. Here are some examples:

Ruby

[Download](#) ruby101/array.rb

```
colors = [ "Blue", "Green", "Orange" ]
puts colors.join
puts colors.join(' and ')
puts colors.join("\n")

c:\dev> ruby array.rb
BlueGreenOrange
Blue and Green and Orange
Blue
Green
Orange
```

Notice that for the last line, we used double quotes around our joining string so that we could specify a newline (carriage return) to join the strings together. This is how we got each element of the array to be printed on its own line.

The `puts` method will call `join("\n")` when given an array to output.



Joe Asks...

What's a Block?

Blocks are a lot like .NET anonymous methods: sections of code that behave like a function but just don't have a name. Some methods expect to collaborate with a snippet of code that you must provide. Your code snippet is a block. Simply enclose your code in a `do...end` block (one-liners should instead use curly braces).

Blocks can even take parameters, just like a method can. Block arguments are surrounded with pipe `|` symbols.

[Download](#) `ruby101/blocks.rb`

```
# The Array.detect method expects us to define a block
# and for our block to return true
# when the given element is the desired
# element we want to find.
sports.detect { |sport| sport.index('ball') != nil }

# Logic that requires more than one line
# should use a do..end pair instead.
# Here, we detect the first sport
# that can be played on a grass surface.
sports.detect do |sport|
  category = find_category(sport)
  category == 'played on grass'
end
```

Blocks arrange your code right alongside the method that's requiring it, making your code more readable than locating a method somewhere else in your program.

This is just one way that Ruby promotes more generic, decoupled, and collaborative program architectures.

Finding Array Elements with Regular Expressions

You may already be familiar with `grep`, a common command-line utility that searches text using regular expressions. Some Ruby classes also implement a method named `grep`, which indicates it has some kind of similar searching feature. The `Array` class provides a `grep` method so that you can easily search an array's elements for those that match a given regular expression:

Ruby

[Download](#) `ruby101/array.rb`

```
colors = [ "Blue", "Green", "Orange", "Red", "Purple" ]
my_colors = colors.grep(/u/)
puts my_colors

c:\dev> ruby array.rb
Blue
Purple
```

Shortcut for Creating an Array of Strings

Ruby provides a shortcut when we want to create an array from a list of words:

Ruby

[Download](#) `ruby101/array.rb`

```
colors = %w(Blue Green Orange Red Purple)
```

The `%w` is a special sequence in Ruby code. The parentheses are commonly used, but you can use any character that you want to use to mark the beginning and ending of your list of words. Whitespace is used to split the string up into the array of words.

Deleting Elements from an Array

Removing an object from an array is easy. Just call the `delete` method. You must provide a reference to the object you want to delete. If the object is found in the array, it will be removed and returned to you. If the object is not found, the `delete` method will simply return `nil`, instead of raising an exception like .NET languages would.

Alternatively, if you don't have a reference to the object but you do know the index position of the element you need to delete, you can use the `remove_at` method to remove whatever element is at that position and have it returned to you. You'll get `nil` back if you supply an index that's out of range for the array.

Ruby

Download ruby101/array.rb

```

colors = [ "Blue", "Green", "Orange", "Red", "Purple" ]
colors.delete 'Blue' # returns "Blue"
colors.delete 'Green' # returns "Green"
colors.delete 'Brown' # returns nil
colors.delete_at(0) # returns "Orange"
colors.delete_at(5) # returns nil

# colors is now [ "Red", "Purple" ]

```

2.6 Symbols

The string and array classes we've examined so far correspond well to similar .NET classes. Symbols, on the other hand, don't have an exact counterpart. They play a vital role in Rails programming, so we need to become familiar with them before we move on.

We will start by looking at two common .NET constructs: constants and enums. Let's start with a typical use of the `const` keyword:

```

public const int Circle = 1;
public const int Square = 2;
public const int Octagon = 3;

public void DrawShape(int shape)
{
    switch (shape)
    {
        case Circle:
            // draw a circle
            break;
        case Square:
            // draw a square
            break;
        case Octagon:
            // draw an octagon
            break;
    }
}

// Let's draw a square
DrawShape(Square);

```

By using names instead of passing raw integers, our code becomes clearer. Inside `DrawShape()`, we used the names of our shapes instead of relying on hard-coded integer values. Outside the class, the benefit is just as large. We could have written `DrawShape(2)` instead, but that would be harder to read and understand.

Some .NET developers want to guarantee that the `DrawShape()` method can receive only a valid shape constant, instead of just any old integer. After all, the actual values we assigned to each shape constant were arbitrary; it didn't really matter what value we assigned to each constant. Enumerated values in .NET are useful for associating names with constant values when you don't really care about the actual value. For example, we can rewrite the previous example using an enum instead:

```
public enum Shape { Circle, Square, Octagon }
```

```
public void DrawShape(Shape shape)
{
    switch (shape)
    {
        case Shape.Circle:
            // draw a circle
            break;
        case Shape.Square:
            // draw a square
            break;
        case Shape.Octagon:
            // draw an octagon
            break;
    }
}
```

```
// Let's draw a square
DrawShape(Shape.Square);
```

This code retains the same level of clarity, and we didn't have to specify any integer values anywhere. (Behind the scenes, the C# compiler will assign integer values to each enumerated value, but it's a hidden, unimportant detail as far as the programmer is concerned.)

Here's the key point: a .NET enum value is a *globally unique, named representation of a single location in memory*. The actual value held at that memory location is unimportant. When all we want is a named value in C#, enums are our friends.

At last, we're ready for Ruby symbols. What is a *symbol*?

A symbol is globally unique, named representation of a single location in memory. (Sound familiar?) So, any time we want to have a nice name for something and we don't care about what value it really has, we use a symbol.

Here's the same code, this time written in Ruby:

```
def draw_shape(shape)
  case shape
  when :circle
    # draw a circle
  when :square
    # draw a square
  when :octagon
    # draw an octagon
  end
end

# Let's draw a square
draw_shape(:square)
```

Symbols always start with a colon, as in `:circle`. Unlike .NET enumerations like `Shape`, we don't need to define symbols before we use them. Ruby will automatically spring them into existence for us the first time they're used.

Symbols can be used like any other type in Ruby. They are objects of type `Symbol`, and you can call methods on them. Here's how we can get a string representation of a symbol, for example:

```
puts :square.to_s # prints "square"
```

As another example, here's how we create an array of them:

```
# An array of three symbols
[:circle, :square, :octagon]
```

The most common place we're going to find ourselves using symbols in Rails is when we work with hashes, which we will explore next.

2.7 Hashes

In Ruby, a *hash* is very much like a .NET Dictionary. Like an array, a hash is a data structure that contains a series of elements. Unlike an array, hash elements are pairs of objects: a *key* and a *value*. Hashes are extremely easy to use in Ruby, and they come in handy in web development where we often want to associate one object with another.

Like arrays in Ruby, hashes are also heterogeneous structures. There is no requirement that all the objects in the hash are of the same type. We can mix and match any type of objects into any of the keys and values. Ruby doesn't care. It's up to you to do whatever you think is appropriate for your situation.

Let's start by looking at a few simple examples. There are two common ways of creating hashes in Ruby. We can create a new instance of the Hash class:

```
irb(main):001:0> hangar_status = Hash.new
=> {}
irb(main):002:0> hangar_status['waiting'] = 3
=> 3
irb(main):003:0> hangar_status['repairing'] = 7
=> 7
irb(main):004:0> hangar_status
=> {"waiting"=>3, "repairing"=>7}
```

Here we see that we can create new key/value pairs by using the [] operator. If the given key does not exist, it is created automatically, and the given value is paired with the given key. If the key already exists, it is simply paired up with the new value.

If we try to access a key that doesn't exist, we simply get a nil value back (whereas some .NET implementations would raise an exception):

```
irb(main):003:0> hangar_status['damaged']
=> nil
```

Alternately, we can use curly braces to directly create a populated Hash:

```
irb(main):001:0> hangar_status = { 'waiting' => 3, 'repairing' => 7 }
=> {"waiting"=>3, "repairing"=>7}
```

Notice that when we work with hash elements, we use the special => syntax to specify the key/value pair.

We can use any kind of objects we want in a hash at any time. Let's keep track of how many passengers are onboard our airplanes. Here we create two Airplane objects and use them as keys in a hash. The values in the hash represent the number of passengers onboard each plane. Note how we create an initially empty hash by just using two curly braces:

```
irb(main):001:0> ord_to_jfk = Airplane.new '747'
=> #<Airplane:0x8c050 @altitude=0, @model="747", @speed=0>
irb(main):002:0> pdx_to_sfo = Airplane.new '707'
=> #<Airplane:0x870a0 @altitude=0, @model="707", @speed=0>
irb(main):003:0> passengers = {}
=> {}
irb(main):004:0> passengers[ord_to_jfk] = 165
=> 165
irb(main):005:0> passengers[pdx_to_sfo] = 104
=> 104
irb(main):006:0> passengers[ord_to_jfk]
=> 165
```

Hashes are a powerful way to store data. It's very important that you become comfortable working with hashes as you begin to develop Rails applications.

Finally, here are some examples that use symbols as keys into a hash:

```
options = { :model => '747', :capacity => 250, :engines => 2 }

puts options[:capacity] # prints 250
options[:engines] = 3   # we now have 3 engines
```

We've completed a quick tour of four of Ruby's data types that we will use frequently in Rails applications: strings, arrays, symbols, and hashes. We haven't yet talked much about the object-oriented nature of the Ruby language. We need to do that now before we can continue to explore the other fundamental elements of the language.

2.8 Everything Is an Object

In Ruby, we like to say everything is an object. Although many .NET languages like C# and VB .NET are considered to be object-oriented, Ruby's definition of object-oriented is more hardcore. Every element of code in Ruby really is an object. To illustrate what we mean, let's look at some of the more startling examples of how objects work in Ruby as we continue to highlight the similarities and differences with .NET.

Even Built-in Types Are Objects

Ruby has built-in types like strings, arrays, and fixnums. Fixnums are like the `Int32` data type in .NET. When we do something like this in Ruby:

Ruby

[Download](#) `ruby101/objects.rb`

```
puts 1 + 2 # prints 3
```

we are still using objects even though it looks like we're using built-in literal values for 1 and 2. The literal number 1 in Ruby is an object! It's actually an instance of the `Fixnum` class. So is the literal number 2. The plus sign is syntactic sugar that allows us to call a method named `+` on the 1 object.

Mirror, Mirror on the Wall

An interesting feature of the Ruby language is the built-in ability for every class and object to tell us about themselves. Every Ruby class is derived directly or indirectly from a built-in class named `Object`. `Object` defines many methods, including some that help us "reflect" upon an

object's type. Here is an example of how we can use the class method to find out the class of a given object:

Ruby

[Download](#) ruby101/objects.rb

```
puts 1.class # prints "FixNum"
puts 2.class # prints "FixNum"
```

.NET calls this technique *reflection*, because we're asking an object to look itself in the mirror and tell us what it sees. Rubyists sometimes like to call it *introspection*, but it's the same concept. Reflection is very useful and powerful in a dynamic language like Ruby, since an object's behavior is subject to change at runtime. Let's learn two more ways to have an object tell us more about itself.

We can use methods to find out what methods we can call on an object:

```
irb(main):001:0> s = "hello"
=> "hello"
irb(main):002:0> s.methods
=> [ "%", "select", "[]=", "inspect", "<<", "each_byte", "clone",
  "method", "gsub", "casecmp", "public_methods", "to_str", "partition",
  "tr_s", "empty?", "instance_variable_defined?", "tr!", "equal?",
  "freeze", "rstrip", "!", "match", "grep", "chomp!", "+", "next!",
  "swapcase", "ljust", "to_i", "swapcase!", "methods", "respond_to?",
  "upto", "between?", "reject", "sum", "hex", "dup", "insert",
  "reverse!", "chop", "instance_variables", "delete", "dump", "__id__",
  "tr_s!", "concat", "member?", "object_id", "succ", "find", "eq?",
  "each_with_index", "strip!", "id", "rjust", "to_f",
  "singleton_methods", "send", "index", "collect", "oct", "all?",
  "slice", "taint", "length", "entries", "chomp", "frozen?",
  "instance_variable_get", "upcase", "sub!", "squeeze", "include?",
  "__send__", "instance_of?", "upcase!", "crypt", "delete!", "detect",
  "to_a", "unpack", "zip", "lstrip!", "type", "center", "<",
  "protected_methods", "instance_eval", "map", "<=>", "rindex",
  "display", "any?", "==" , ">", "split", "===", "strip", "size",
  "sort", "instance_variable_set", "gsub!", "count", "succ!",
  "downcase", "min", "kind_of?", "extend", "squeeze!", "downcase!",
  "intern", ">=", "next", "find_all", "to_s", "<=", "each_line",
  "each", "rstrip!", "class", "slice!", "hash", "sub", "tainted?",
  "private_methods", "replace", "inject", "=~", "tr", "reverse", "nil?",
  "untaint", "sort_by", "lstrip", "to_sym", "capitalize", "max",
  "chop!", "is_a?", "capitalize!", "scan", "[]"]
```

Wow, that's a lot to look at. Let's make it easier to read by sorting the methods first and by using `grep` to find only the methods that start with, say, the letter `s`:

```
irb(main):003:0> s.methods.sort.grep(/s/)
=> ["scan", "select", "send", "singleton_methods", "size", "slice",
  "slice!", "sort", "sort_by", "split", "squeeze", "squeeze!", "strip",
  "strip!", "sub", "sub!", "succ", "succ!", "sum", "swapcase",
  "swapcase!"]
```

Use methods inside an `irb` session when you want to discover exactly what a class or object can do. Here we learned how handy the `sort` and `grep` methods can be.

Quacking Like a Duck

Every Ruby object, like the string object we just examined, ultimately inherits from `Object`. This is the same as it is in `.NET`. As a result, many of the methods that a string object has are inherited from `Object`. Perhaps the most interesting method of all of these is the `respond_to?` method. With this method, we can find out at runtime whether an object can respond to a particular message or method:

```
irb(main):005:0> s.respond_to?('split')
=> true
irb(main):006:0> s.respond_to?('translate_to_spanish')
=> false
```

Here we've used the `respond_to?` method to discover that the object `s` implements a method called `split` but does not have a method named `translate_to_spanish`.

This is the main idea behind *duck typing*: Ruby can detect available runtime behaviors at the method level. Instead of achieving polymorphic behavior only through base class inheritance or interface implementation, Ruby code can work with objects that simply “talk” and “walk” as expected, regardless of their actual type.

2.9 Classes and Objects

Declaring classes and using classes are the bread and butter of any object-oriented system, and the reasons for using classes, instantiating objects, and calling their methods are the same in Ruby as they are in `.NET`. Let's look at some concrete code examples to see how they compare.

Let's Fly an Airplane

We will start by reviewing how we define classes in the `.NET` language `C#`. Here is a `C#` class that represents an airplane, perhaps to be used in a flight simulator program or as part of an air traffic control system:

`.NET`

[Download](#) `ruby101/classes.cs`

```
using System;
using System.Collections.Generic;
```

```
public class Airplane
{
    private string model;
    private int altitude;
    private int speed;
    public Airplane(string model)
    {
        this.model = model;
        this.altitude = 0;
        this.speed = 0;
    }

    public string Model
    {
        get
        {
            return this.model;
        }
    }

    public int Altitude
    {
        get
        {
            return this.altitude;
        }
    }

    public int Speed
    {
        get
        {
            return this.speed;
        }
    }

    public void Fly()
    {
        if (this.model == "777")
            this.altitude = 40000;
        else
            this.altitude = 30000;

        this.speed = 500;
    }

    public void Land()
    {
        this.altitude = 0;
        this.speed = 0;
    }
}
```

```

public static List<String> Models
{
    string[] availableModels = { "707", "747", "777" };
    return new List<string>(availableModels);
}
}

```

We've defined a public class called `Airplane` that must be initialized with a model parameter. A static method called `Models()` returns a list of available models that can be used. An `Airplane` instance can keep track of its speed and altitude, and these are changed by calling the `Fly()` and `Land()` methods. Finally, we defined three read-only properties, `Model`, `Altitude`, and `Speed`.

Now, let's translate this class directly into Ruby. Our first attempt may not result in the most concise Ruby code ever written, but we'll refine the code after we've performed our initial translation:

Ruby

[Download](#) `ruby101/classes.rb`

```

Line 1  class Airplane
-
-
-   def initialize(model)
-       @model = model
5       @altitude = 0
-       @speed = 0
-   end
-
-   def model
10      return @model
-   end
-
-   def altitude
-       return @altitude
15  end
-
-   def speed
-       return @speed
-   end
20
-   def moving?
-       return @speed > 0
-   end
-
25  def fly
-       if @model == '777'
-           @altitude = 40000
-       else
-           @altitude = 30000
30  end
-
-

```

```

-     @speed = 500
-   end
-
35  def land
-     @altitude = 0
-     @speed = 0
-   end
-
40  def self.models
-     return [ '707', '747', '777' ]
-   end
-
45  end

```

Ruby is a very readable language, but let's go through it a step at a time. We introduce a new class scope with the `class` keyword. Our `Airplane` class doesn't explicitly derive from any other class, so it will be derived from the built-in `Object` class by default.

On line 3, we see how to define the *initializer* for our class. The initializer is Ruby's equivalent of a .NET constructor. You can define your initializer to accept parameters, but note that only one initializer is allowed, since Ruby does not support method overloading. Here we've declared that our initializer will require one parameter, the "model" of airplane that should be created.

In the body of the initializer, we have defined three instance variables, `@model`, `@altitude`, and `@speed`. The `@` sign declares them as private instance variables, whose type is automatically determined by the assigned values.

On line 9, we see how to declare a method that will give us property-like syntax for reading the value of an instance variable. We've defined the `model` method, which simply returns the value of our `@model` attribute. In this way, we provide read-only access to the `@model` variable. We've done the same thing for the `@altitude` value, by providing a wrapper method for it as well.

The `fly` method on line 25 uses a simple `if` statement to determine at which altitude we should fly, based on the type of airplane being flown. The instance variable `@speed` is set to 500 regardless of the airplane type. The `land` method, as its name implies, bring us back to Earth and brings the plane to a stop.

Finally, the `models` method is an example of a *class method*. Class methods are like static methods in .NET. They're a convenient way to house methods that logically belong to a class, even though a specific instance of that class is not needed to call the method. Here, we call `Airplane.models` to retrieve a list of the valid models that we can use when constructing an airplane instance:

Ruby

[Download](#) ruby101/classes.rb

```
model_to_use = Airplane.models[1]

airplane = Airplane.new(model_to_use)
puts "Our #{airplane.model} is starting at #{airplane.altitude} feet"

airplane.fly
puts "Our #{airplane.model} is currently at #{airplane.altitude} feet"

if airplane.moving?
  puts "Yes, the plane is moving."
end

airplane.land
puts "Our #{airplane.model} is now at #{airplane.altitude} feet"
```

What happens when we execute this Ruby script? See whether you guessed correctly:

```
c:\dev> ruby classes.rb
Our 747 is starting at 0 feet
Our 747 is currently at 30000 feet
Yes, the plane is moving.
Our 747 is now at 0 feet
c:\dev>
```

A More Idiomatic Approach

The class we just wrote was fairly readable, and it reflects how it would probably look if it were written by a C# programmer making his first Ruby program. However, in practice, this code would be a bit different. Let's learn about just a couple of standard idioms commonplace in Ruby programming. Here is a revised example that shows one way our code could have been written:

Ruby

[Download](#) ruby101/classes.rb

```
Line 1 class Airplane
-
-   attr_reader :model
-   attr_reader :altitude
5   attr_reader :speed
-
```

```

-   def initialize(model)
-     raise "Model not recognized!" unless Airplane.models.include?(model)
-
10    @model = model
-     @altitude = 0
-     @speed = 0
-   end
-
15  def fly
-     @speed, @altitude = 500, cruising_altitude
-   end
-
-   def land
20    @altitude = 0
-     @speed = 0
-   end
-
-   def moving?
25    @speed > 0
-   end
-
-   def self.models
-     [ '707', '747', '777' ]
30  end
-
- private
-
-   def cruising_altitude
35    @model == '777' ? 40000 : 30000
-   end
-
- end

```

What did we change?

- We omitted return statements wherever it seemed reasonable. Ruby always uses the value of the last-evaluated expression as the return value of a method, so often it's not needed to use the return statement explicitly.
- Like a *getter* in .NET properties, we used the built-in `attr_reader` method to automatically define read-only access methods to the `@model` and `@altitude` values. This helps us avoid writing boilerplate code like we did before to return those values.
- Because type safety is not ensured as it would be with a statically typed language such as C#, we added a check in the initializer to make sure a valid model was given and to raise an exception if we get an unexpected model.

- Finally, we've added a private section to the class and defined a new private method called `cruising_altitude`. This allowed us to make the `fly` method more readable and easier to maintain. We assign multiple values at once in Ruby using a comma-separated list of values on each side of an assignment statement.

Writing Ruby that feels natural takes some time and experience. Seek out Ruby libraries that have been written by longtime Rubyists to get a sense of how to recognize good Ruby style when you see it. Reading other code written by good Ruby coders is one of the best ways to learn how to write code in Ruby.

2.10 Loops

We will close this chapter by looking at another essential building block of Ruby programs: the concept of loops and iterators. Web applications spend a lot of time working with sets of data, usually enumerating over them for some purpose, perhaps to display a list of some kind, select a subset of data based on some kind of criteria, or transform a set of objects from one kind to another.

Some looping constructs in Ruby are quite different from the constructs you have experienced in .NET, while some of them are almost the same. There are two main differences to keep in mind when learning to read and write loop statements in Ruby as compared with C# or VB .NET:

- We don't need to explicitly declare any types, since Ruby always determines variable types automatically.
- Everything is an object, including seemingly built-in literals, so some loops are easier to write.

Simple Iterations

Here's a simple for loop that prints the digits 0 to 9 in C#:

.NET

[Download](#) `ruby101/loops.cs`

```
for (int i = 0; i < 10; ++i)
{
    Console.WriteLine(i);
}
```

And here's one way to do it in Ruby:

```
irb(main):001:0> 0.upto(9) { |n| puts n }
0
1
2
3
4
5
6
7
8
9
=> 0
```

Iterating Over Every Element of an Array

The most common way to simply loop through each element of an array is to use the `each` method:

```
irb(main):008:0> odds = [1, 3, 5, 7, 9]
=> [1, 3, 5, 7, 9]
irb(main):009:0> odds.each { |n| puts n }
1
3
5
7
9
=> [1, 3, 5, 7, 9]
```

The `each` method class is one example of an *iterator*. An iterator is a powerful pattern in Ruby programs by which a data structure's elements are visited in collaboration with a user-supplied block. Iterators are just one of the important dimensions of the Ruby language we'll be exploring in the next chapter.

We've started from the simplest of Ruby scripts; seen how primary elements like strings, arrays, and loops compare with their .NET counterparts; and dipped our toes in some advanced topics such as introspection and a few rules of idiomatic Ruby. It's time to learn more about iterators, a central concept in all Rails applications; see some advanced Ruby syntax; and take a closer look at code reuse techniques. These are the skills that will take us beyond thinking about simple .NET-to-Ruby translations so that we can instead “think in Ruby” as our native language.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Rails for .NET Developers' Home Page

<http://pragprog.com/titles/cerain>

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

<http://pragprog.com/updates>

Be notified when updates and new books become available.

Join the Community

<http://pragprog.com/community>

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

<http://pragprog.com/news>

Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/cerain.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com