This appendix is designed to act as both a Ruby primer and review, useful both to developers who want to brush up rapidly on their Ruby knowledge, and to those who are new to the language but who have existing programming knowledge and want to get a quick overview.

If you’re a new programmer, or at least are new to concepts such as object orientation, scripting languages, and dynamic languages, you’ll want to read through Chapter 2 and continue with the rest of the book instead of depending on this appendix to teach you about Ruby. This appendix is designed for either those who have finished reading the rest of this book and want to brush up on the basics, or those who want to look quickly through some basic elements of Ruby syntax in the flesh.

With that in mind, this appendix isn’t instructional, as most of the other chapters in this book are. A lot of concepts will be covered at a quick pace with succinct code examples. References to more explanatory detail found in this book are given where possible.

The Basics

In this section, I’ll give a brief overview of the Ruby programming language, its concepts, and how to use the Ruby interpreter.

Definition and Concepts

Ruby is an open source, object-oriented programming language created and maintained by Yukihiro Matsumoto (among others). Languages such as Perl, LISP, Smalltalk, and Python have inspired the syntax and styling of the language. It is cross platform and runs on several different architectures, although its informal “home” architecture is Linux on x86.

Among other things, Ruby has automatic garbage collection, is (mostly) portable, supports multitasking (both native and its own cooperative “green” threads), has a large standard library, and supports most features associated with dynamic languages (such as closures, iterators, exceptions, overloading, and reflection).

Ruby is an interpreted language. This is in opposition to languages that are compiled. Code developed in languages such as C and C++ has to be compiled into object code that represents instructions supported by a computer’s processor. Ruby, however, is compiled down into platform-independent bytecode that is run by a virtual machine. Python, Java, and C# share this characteristic, although they all run on different virtual machine implementations and have different execution characteristics. Table A-1 highlights some key differences between several popular programming languages.
Ruby has been developed with the “principle of least surprise” in mind, so the way you’d expect things to work is usually a valid way of doing something. This means Ruby is very much a “there’s more than one way to do it” type of language, in the same vein as Perl but quite different in philosophy from languages such as Python, where having one clear process to achieve something is seen as the best way to do things.

One important concept in Ruby is that almost everything is an object. For example, the following line of code calls a primitive, internal method called `puts` with a single argument of `10`. `puts` prints its arguments to the screen:

```ruby
puts 10
```

```
10
```

You could run this as a complete Ruby program, or perform it in an interactive manner using Ruby’s `irb` tool.

The following line of code calls the `class` method on the numeric object `10`. Even the literal number 10 is an object in this situation. The result demonstrates that `10` is an object of the `Fixnum` class.

```ruby
puts 10.class
```

```
Fixnum
```
Ruby's reflection, overriding, object orientation, and other dynamic features make it possible for developers to entirely override the behaviors of even built-in classes such as Fixnum. It's possible to make Fixnum objects work in totally different ways. You can override Fixnum to the point that 2 + 2 could well equal 5. Although some developers already experienced with languages such as Java and C see this as a downside, this level of control over the internals of the language gives Ruby developers a significant amount of power. The key is to use that power carefully.

The Ruby Interpreter and Running Ruby Code

As Ruby is an interpreted language, Ruby code is executed using the Ruby interpreter. On most platforms, that makes running a Ruby script as easy as this:

```ruby
ruby name_of_script.rb
```

---

**Note** Ruby program files usually end with the extension of `.rb`, although this isn’t a strict requirement.

---

The Ruby interpreter has a number of options. You can ask the Ruby interpreter to print out its version details using the `-v` (version) option:

```ruby
ruby -v
```

```ruby
ruby 2.2.0p0 (2014-12-25 revision 49005) [x86_64-darwin13]
```

You can also execute Ruby commands directly from the command line, using `-e`:

```ruby
ruby -e "puts 2 + 2"
```

```ruby
4
```

You can learn more about the Ruby interpreter’s command-line options by typing `man ruby` (on UNIX-related platforms) or by visiting a web-based version of the Ruby `man` page at `http://www.linuxcommand.org/man_pages/ruby1.html`.

---

**Note** On Microsoft Windows, you might choose to associate the Ruby interpreter directly with any `.rb` files so that you can double-click Ruby files to execute them.

---

On UNIX-related platforms, it's possible to add a “shebang” line as the first line of a Ruby script so that it can be executed without having to invoke the Ruby interpreter explicitly. For example:

```ruby
#!/usr/bin/ruby
puts "Hello, world!"
```
You can take this script, give it a simple filename such as hello (no .rb extension needed), make the file executable (using chmod), and run it directly using its filename rather than having to invoke the Ruby interpreter explicitly. Chapter 10 covers this technique in more depth. More information about the shebang line specifically is available at http://en.wikipedia.org/wiki/Shebang_(Unix).

Interactive Ruby

With the normal Ruby interpreter also comes an interactive Ruby interpreter called irb. This allows you to write Ruby code in an immediate, interactive environment where the results of your code are given as soon as you type it. Here’s an example irb session:

```
# irb
irb(main):001:0> puts "test"
test
=>nil
irb(main):002:0> 10 + 10
=> 20
irb(main):003:0> 10 == 20
=>false
irb(main):004:0> exit
```

irb gives you the results of methods and expressions immediately. This makes it an ideal tool for debugging or putting together quick snippets of code, and for testing concepts.

Expressions and Flow Control

Expressions, logic, and flow control make up a significant part of any developer’s tools in any programming language. This section looks at how Ruby implements them.

Basic Expressions

Ruby supports expressions in a style familiar to almost any programmer:

"a" + "b" + "c"

```
abc
```

10 + 20 + 30

```
60
```

("a" * 5) + ("c" * 6)

```
aaaaacccccc
```
You can assign the results of expressions to variables, which you can then use in other expressions. Method calls, variables, literals, brackets, and operators can all combine so long as subexpressions always feed values of the correct type into their parent expressions or provide methods that allow them to be coerced into the right types. The next section covers this topic in more depth. (Expressions are covered in depth in Chapter 3.)

Class Mismatches

Ruby is a dynamic language, but unlike Perl, objects aren’t converted between different classes automatically (in this sense Ruby is a strongly typed language). For example, this expression is valid in Perl:

"20" + 10

However, in Ruby, you get an error response with the same expression:

```ruby
TypeError: no implicit conversion of Fixnum into String
from (irb):1:in `+
from (irb):1
```

In Ruby, you can only use objects that are of the same class or that support automatic translation between classes (coercion) in operations with one another (usually via methods called things like `to_s` and `to_h`, for conversions to strings and hashes, respectively).

However, Ruby comes with a set of methods that exist on many types of objects, which make conversion easy. For example:

"20" + 10.to_s

```
2010
```

In this example, the number 10 is converted to a string "10" in situ with the `to_s` method. Consider this inverse example, where you convert the string "20" into an integer object using the `to_i` method before adding 10 to it:

"20".to_i + 10

```
30
```
Note  Methods are covered in depth in Chapters 2, 3, and 6, as well as later in this appendix.

The to_s method provided by all number classes in Ruby results in a number being converted into a String object. C and C++ programmers might recognize this concept as similar to casting.

Other conversions that can take place are converting integers to floats using to_f, and vice versa with to_i. You can convert strings and numbers using to_s, to_i, and to_f. Many other classes support to_s for converting their structure and other data into a string (the Time class provides a good demonstration of this). This topic is covered in Chapter 3 in the section “Converting Objects to Other Classes.”

Comparison Expressions

Comparison expressions in Ruby, as in most other languages, return true or false, except that in some situations comparisons might return nil, Ruby’s concept of “null” or nonexistence. For example:

```
2 == 1
false

2 == 2
true

(2 == 2) && (1 == 1)
true

x = 12
x * 2 == x + 1
false

x * x == x ** 2
true
```

In each of the preceding examples, you test whether variables, literals, or other expressions are equal to one another using == (symbolizing “is equal to”). You can check that multiple expressions result in true (logical “and”—if x and y are true) using && (symbolizing “and”).

As in other languages, the concept of a logical “or” is symbolized by ||:

```
(2 == 5) || (1 == 1)
true
```
This expression is true because even though 2 is not equal to 5, the other sub-expression is true, meaning that one or another of the expressions is true, so the whole comparison is also true.

Last, it can be useful to negate expressions. You can do this with the ! operator, as in many other programming languages. For example, you might want to see if one thing is true but another thing is false. Here's an example:

\[(2 == 2) \&\& ! (1 == 2)\]

true

The expression is true because both sub-expressions are true. 2 is equal to 2, and 1 is not equal to 2. You can also check that one thing is not equal to another with the inequality operator !=:

\[(2 == 2) \&\& (1 != 2)\]

true

**Flow**

Ruby supports a few different forms of flow control. In this section, you'll see several techniques you can use for branching and looping. (All the topics in this section are covered in more depth in Chapter 3.)

**Branching and Conditional Execution**

The simplest form of conditional execution is with just a single line using if or unless:

```ruby
puts "The universe is broken!" if 2 == 1
```

This example won't print anything to the screen because 2 is not equal to 1. In this case, if performs the comparison before the rest of the line is executed. This usage will be familiar to Perl programmers (indeed, that's where this construction came from), but might appear back-to-front to developers from languages such as C.

Ruby also supports a multiline construction that's more familiar to non-Perl or Ruby coders:

```ruby
if 2 == 1
    puts "The universe is broken!"
end
```

This multiline construction is less space efficient than the previous, single-line construction, but it allows you to put multiple lines between the condition and the end of the block, which isn't possible with the "end of line" technique. Pascal coders should note the absence of a begin, though otherwise the style is similar to that in Pascal.

**Note** unless is the opposite of if. It executes code if the expression is false (or nil), rather than true. Some Rubyists think of it as "if not," because unless acts like if with the expression negated. Other developers avoid it entirely due to the potential confusion it can cause.
Ruby also supports the else directive, as found in languages such as C, Perl, and Pascal:

```ruby
if 2 == 1
  puts "The universe is broken!"
else
  puts "The universe is okay!"
end
```

The universe is okay!

If the expression (2 == 1 in this example) is true, the main block of code is executed, else the other block of code is. There's also a feature called elsif that lets you chain multiple ifs together:

```ruby
x = 12
if x == 1 || x == 3 || x == 5 || x == 7 || x == 9
  puts "x is odd and under 10"
elsif x == 2 || x == 4 || x == 6 || x == 8
  puts "x is even and under 10"
else
  puts "x is over 10 or under 1"
end
```

The preceding rather obtuse example demonstrates how you can use if, elsif, and else in tandem. The only thing to note is that end always finishes an if (or unless) block, whether end is on its own or features elsif and else blocks, too. In some languages, there's no need to delimit the end of if blocks if they contain only a single line. This isn't true of Ruby.

---

**Note**  
C coders will be used to else if. Ruby's variation is based on the Perl standard of elsif.

Ruby also supports another construction familiar to C, C++, Java, and Pascal coders, called case (known as switch in C, C++, and Java):

```ruby
fruit = "apple"
color = case fruit
  when "orange"
    "orange"
  when "apple"
    "green"
  when "banana"
    "yellow"
  else
    "unknown"
end
puts color
```

green
This code is similar to the `if` block, except that the syntax is a lot cleaner. A case block works by processing an expression first (supplied after `case`), and then the case block finds and executes a contained `when` block with an associated value matching the result of that expression. If no matching `when` block is found, then the `else` block within the case block will be executed instead.

The `case` is, essentially, a substitution for a large, messy clump of `if` and `elsif` statements.

### The Ternary Operator (Conditional Expressions)

Ruby supports a construction called the _ternary operator_. Its usage is simple:

```ruby
x = 10
puts x > 10 ? "Higher than ten" : "Lower or equal to ten"
```

```
Lower or equal to ten
```

The ternary operator works like so:

```ruby
expression ? true_expression : false_expression
```

It works like an expression, but with built-in flow control. If the initial expression is true, then the first following expression will be evaluated and returned. If the initial expression is false, then the final following expression will be evaluated and returned instead.

### Loops

Ruby supports loops in a similar way to other programming languages. For example, `while`, `loop`, `until`, `next`, and `break` features will be familiar (although with possibly different names) to most programmers.

**Note** Ruby also supports iteration and code blocks, which can prove a lot more powerful than regular loops. These are covered later in this appendix and in Chapters 2, 3, and 6.

Loop techniques are covered in Chapter 3, but some basic demonstrations follow. Here's a permanent loop that you can break out of using `break`:

```ruby
i = 0
loop do
  i += 1
  puts i
  break if i > 100
end
```

**Note** It's worth noting that unlike in C or Perl, you cannot increment variables by 1 with `variable++` in Ruby. `variable = variable + 1` or `variable += 1` are necessary instead.
Here's a `while` loop, using `next` to skip even numbers (using the `%` modulo operator):

```ruby
i = 0
while (i < 15)
i += 1
next if i % 2 == 0
puts i
end
```

1 3 5 7 9 11 13 15

---

**Note** until is the opposite of `while`. until `(i >= 15)` is equivalent to `while (i < 15)`.

Further looping techniques are covered in Chapter 3 and throughout the book.

## Object Orientation

Ruby is considered a *pure* object-oriented language, because everything appears to Ruby as an object. An earlier example in this appendix demonstrated this:

```ruby
puts 10.class
```

`Fixnum`

Even literal data (such as strings or numbers embedded directly in your source code) is considered to be an object, and you can call the methods made available by those objects (and/or their parent classes).

---

**Note** Object orientation, classes, objects, methods, and their respective techniques are covered in full in Chapters 2 and 6. This section presents merely a brief overview.

Ruby implements object orientation in a simple way (syntax-wise), but offers more dynamic features than other major languages (see Chapter 6 for many examples of such features).
Objects

Objects in Ruby have no special qualities beyond objects that exist in any other object-oriented programming language. However, the key difference between Ruby and most other major object-oriented languages is that in Ruby everything is an object. With this in mind, you can call methods on almost everything, and even chain methods together.

In C or Perl, it would be common practice to write code in this form:

```
function1(function2(function3(something)))
```

However, in Ruby you’d do this:

```
something.function3.function2.function1
```

Periods are used between an object and the method to call, as in C++ or Java (as opposed to -> used in Perl). In this example, you call the function3 method on the something object, then the function2 method on the result of that, and then the function1 method on the result of that. A real-world demonstration can illustrate:

```
"this is a test".reverse
```

```
tset a si siht
```

```
"this is a test".reverse.upcase.split(' ').reverse.join('-')
```

```
SIHT-SI-A-TSET
```

This example is deliberately long to demonstrate the power of method chaining in Ruby. The syntax is a lot cleaner than the equivalent in Perl, C, or C++, and almost reads like English. This example takes your string "this is a test", reverses it, converts it to uppercase, splits it into words (splitting on spaces), reverses the position of the words in an array, and then joins the array back into a string with each element separated by dashes. (Objects are covered in depth in Chapters 2, 3, and 6.)

Classes and Methods

Ruby classes are similar in style to those in Perl, C++, or Java, but keep the benefits of Ruby’s dynamic features. Let’s look at an example class definition:

```
class Person
  def initialize(name, age)
    @name = name
    @age = age
  end

  def name
    return @name
  end
end
```
This class features an `initialize` method that is called automatically when you create a new instance of that class. Two parameters or arguments are accepted (name and age) and assigned to instance variables. Instance variables are variables associated with a particular instance of a class and begin with an @ sign (as in @name). Java developers should recognize @name as being similar to `this.name`.

After the initializer come two methods (name and age) that act as basic accessors. They simply return the value of their respective instance variables.

---

**Note** In Ruby, if no value is explicitly returned from a method, the value of the last expression is returned instead. Therefore, `return @name` and just `@name` as the last line in the `name` method would be equivalent.

With the preceding class definition, it’s trivial to create new objects:

```ruby
person1 = Person.new('Chris', 25)
person2 = Person.new('Laura', 23)
puts person1.name
puts person2.age
```

Chris
23

One benefit of Ruby is that you can add features to classes even if they’ve already been defined. Within the same program as before, you can simply “reopen” the class and add more definitions:

```ruby
class Person
  def name=(new_name)
    @name = new_name
  end
  def age=(new_age)
    @age = new_age
  end
end
```

These new methods are added to the `Person` class and are automatically made available to any existing instances of that class. These new methods are setter methods, as signified by the equals sign following their names. They allow you to do this:

```ruby
person1.name = "Barney"
person2.age = 101
puts person1.name
puts person2.age
```
Ruby can simplify most of the preceding work for you though, as it provides the `attr_accessor` helper method that automatically creates accessors and setter methods within a class for you.

```ruby
class Person
  attr_accessor :name, :age
end
```

You can also create class methods: methods that don’t exist within the scope of a single object, but that are bound directly to the class. For example:

```ruby
class Person
  @@count = 0

  def initialize
    @@count += 1
  end

  def self.count
    @@count
  end
end

a = Person.new
b = Person.new
c = Person.new
puts Person.count
```

This Person class implements a count class method (notice that it is defined as `self.count`, rather than just `count`, making it a class method). The count class method returns the value of a class variable (`@@count`) that stores the total number of Person objects created so far. Class variables begin with two `@` signs and exist within the scope of a class and all its objects, but not within the scope of any specific object. Therefore, `@@count` equals 3 and only 3 once you’ve created three Person objects.

This section has given only a brief overview of classes, objects, and their special variables. For a detailed look at classes and objects, refer to Chapter 6.

**Reflection**

Ruby is often called a reflective language, as it supports reflection. Reflection is a process that allows a computer program to observe and modify its own structure and behavior during execution. This functionality can seem like a novelty to developers experienced with C, C++, and Perl, but it’s incredibly important in terms of Ruby’s operation and Ruby’s ability to define domain-specific languages, making other forms of development easier.
A brief demonstration of reflection is the ability to programmatically retrieve a list of all the methods associated with any object or class in Ruby. For example, here’s how to display a list of all methods of the Hash class:

Hash.methods


Similarly, you can retrieve a list of methods available on a String object directly:

"testing".methods

Note The results shown are from Ruby 2.2. Future versions of Ruby may show different results.

The results given by the `methods` method might seem overwhelming at first, but over time they become incredibly useful. Using the `methods` method on any object allows you to learn about methods that aren’t necessarily covered in this book (or other books), or that are new to the language. You can also use `methods` to retrieve a list of class methods, because classes are also objects in Ruby!

This section provides only a taste of reflection, but the topic is covered in more detail in Chapter 6.

Reopening Classes

It’s trivial to override already defined methods on classes. Earlier in this appendix, I mentioned that, if you so wish, you can adjust the `Fixnum` class so that 2 + 2 would equal 5. Here’s how you do that:

```ruby
class Fixnum
  alias_method :old_plus, :+

  def +(other_number)
    return 5 if self == 2 && other_number == 2
    old_plus other_number
  end
end

puts 2 + 2
```

5

The first thing this code does is to enter the `Fixnum` class, so you can define methods and perform actions within it. Next, you make an alias from the addition operator/method (+) to a new method called `old_plus`. This is so you can still use the normal addition feature, though with a different name.

Next, you redefine (or “override”) the + method and return 5 if the current number is 2 and the number you’re adding to the current number is also 2. Otherwise, you simply call `old_plus` (the original addition function) with the supplied argument. This means that 2 + 2 now equals 5, but all other addition is performed correctly.

You can redefine nearly any method within Ruby. This can make testing essential because you (or another developer) might incorporate changes that affect classes and objects being used elsewhere within your program. Testing is covered in Chapters 8 and 12.

Method Visibility

It’s possible to change the visibility of methods within Ruby classes in one of three ways. Methods can be public (callable by any scope within the program), private (callable only within the scope of the instance the methods exist upon), and protected (callable by any object of the same class). Full details about method visibility are available in Chapter 6.
To encapsulate methods as public, private, or protected, you can use two different techniques. Using the words public, private, and protected within a class definition causes the methods defined thereafter to be encapsulated in the respective fashion:

```ruby
class MyClass
  def public_method
  end

  private
  def private_method1
  end

  def private_method2
  end

  protected
  def protected_method
  end
end
```

You can also explicitly set methods to be encapsulated in one way or another, but only after you’ve first defined them. For example:

```ruby
class MyClass
  def public_method
  end

  def private_method1
  end

  def private_method2
  end

  def protected_method
  end

  public :public_method
  private :private_method1, :private_method2
  protected :protected_method
end
```

Declarations such as this should come after you define the methods, as otherwise Ruby won’t know what you’re referring to.

## Data

As everything is an object in Ruby, all forms of data represented within Ruby are also objects, just of varying classes. Therefore, some Ruby developers will try to correct you if you refer to types rather than classes, although this is merely pedantry.

In this section, we’ll take a quick look at some of the basic data classes in Ruby.
Strings

Strings in Ruby are generally unexceptional, except for the object-oriented benefits you gain. Previously in this appendix, we looked at how powerful classes and methods can be when working on strings:

```
"this is a test".reverse.upcase.split(' ').reverse.join('-')
```

SIHT-SI-A-TSET

The String class offers a plethora of useful methods for managing text. I’ll cover several of these in the following “Regular Expressions” section. However, if you want to see what other methods strings offer, it’s easy: just execute "test".methods.

Regular Expressions

In Ruby, regular expressions are implemented in a reasonably standard way, being somewhat aligned with the Perl style. If you’re familiar with regular expressions, Ruby’s techniques shouldn’t seem alien:

```
"this is a test".sub(/[aeiou]/, '*')
```

th*s is a test

```
"this is a test".gsub(/[aeiou]/, '*')
```

th*s *s * t*st

```
"THIS IS A TEST".gsub(/[aeiou]/, '*')
```

THIS IS A TEST

```
"THIS IS A TEST".gsub(/[aeiou]/i, '*')
```

TH*S *S * T*ST

sub performs a single substitution based on a regular expression, whereas gsub performs a global substitution. As in Perl, you use the /i option to make the regular expression case insensitive.

Ruby also makes matching easy, with the match method of String returning a special MatchData array you can query:

```
m = "this is a test".match(/\b..\b/)
m[0]
```

is
m = "this is a test".match(/(.)(.)/) 

m[0] 

is 

m[1] 

i 

m[2] 

s 

The latter example demonstrates how you can parenthesize elements of the regular expression to separate their contents in the results. m[0] contains the full match, whereas m[1] onward matches each set of parentheses. This behavior is similar to that of $1, $2, $.. in Perl (note that these special variables also exist in Ruby, but their use is generally not preferred).

You can also scan through a string, returning each match for a regular expression:

"this is a test".scan(/[aeiou]/) 

['i', 'i', 'a', 'e'] 

"this is a test".scan(/\w+/) 

['this', 'is', 'a', 'test'] 

Methods such as split also accept regular expressions (as well as normal strings):

"this is a test".split(/\s/) 

['this', 'is', 'a', 'test'] 

Regular expressions are covered in more depth in Chapter 3, and are used throughout the book.

Numbers

Integers and floating point numbers are available in Ruby and operate mostly as you'd expect. Numbers support all common operators such as modulus (%), addition, subtraction, division, multiplication, and powers (**).

Note  You can produce roots easily by raising a number to the power of 1 divided by the root desired. For example, you can find the square (2) root of 25 with 25 ** 0.5.
A key consideration with numbers in Ruby is that unless you explicitly define a number as a floating point number, it won’t be one unless it contains a decimal point. For example:

\[
10 \div 3
\]

\[
3
\]

In this situation, 10 and 3 are both considered integers, so integer division is used. If integer division is what you’re after—and it might be in some cases—then you’re fine. But if you’re after floating point division, you need to do something to ensure that at least one of the values involved is recognized as a floating point number. You can generate a floating point value in one of three ways as follows:

- By invoking the `to_f` method, to convert an integer to its floating point equivalent
- By writing the number with a decimal point, even if you just add “.0” to the end
- By invoking the `Float()` initializer method to convert an integer to a floating point value

Here are some examples:

\[
10.0 / 3
\]

\[
3.33333333333335
\]

\[
10.0 / 3
\]

\[
3.33333333333335
\]

\[
10 / Float(3)
\]

\[
3.33333333333335
\]

Which method you choose to make the 10 be recognized as a `Float` object can be largely influenced by the situation, so it’s useful to see all your options.

Another useful feature in Ruby is that even though whole numbers are typically stored as 32-bit integers internally, Ruby automatically converts integer `Fixnum` objects into `Bignum` objects when the 32-bit barrier is breached. For example:

\[
(2 ** 24).class
\]

`Fixnum`

\[
(2 ** 30).class
\]

`Bignum`
Ruby appears to have no problem in dealing with numbers of tens of thousands of digits in length, certainly enough to solve any mathematical problems you might face! Numbers are covered in depth in Chapter 3.

Arrays

As in other programming languages, arrays act as ordered collections. However, in Ruby specifically, arrays are ordered collections of objects, because everything in Ruby is an object! Arrays can contain any combination of objects of any class.

At first sight, Ruby arrays work much like arrays in any other language, although note that you work on an array using methods, because an array itself is an object. The following example shows the invocation of the Array class's push method:

```ruby
a = []
a.push(10)
a.push('test')
a.push(30)
a<< 40

[10, 'test', 30, 40]
```

Notice the use of a different form of pushing objects to an array with the `<<` operator on the last line of the preceding example. You can then retrieve elements like so:

```ruby
puts a[0]
puts a[1]
puts a[2]
```

10
test
30

**Note** Although `[]` defines an empty literal array, you can also use `Array.new` to generate an empty array if you prefer to stick to object orientation all the way. Java and C++ developers might prefer this syntax initially.

Arrays are objects of class `Array` and support a plethora of useful methods, as covered in full in Chapter 3.
Hashes (Associative Arrays)

Hashes (also known as associative arrays) exist as a concept in many programming languages, such as Perl, Java, and Python (where they are called dictionaries). Hashes are data structures that let you associate keys with values.

Ruby’s implementation of hashes is straightforward and should be familiar to both Perl and Python developers, despite some minor syntax changes. For example:

```ruby
fred = {
  'name' => 'Fred Elliott',
  'age' => 63,
  'gender' => 'male',
  'favorite painters' => ['Monet', 'Constable', 'Da Vinci']
}
```

Fred refers to a basic hash that contains four elements that have keys of 'name', 'age', 'gender', and 'favorite painters'. You can refer back to each of these elements easily:

```ruby
puts fred['age']
```

63

```ruby
puts fred['gender']
```

male

```ruby
puts fred['favorite painters'].first
```

Monet

Hashes are objects of class Hash and come with a large number of helpful methods to make hashes easy to navigate and manipulate, much like regular arrays. It’s important to note that both hash element keys and values can be objects of any class themselves, as long as each element key is distinct. Otherwise, previously existing values will be overwritten. Hashes and associated methods and techniques are covered in detail in Chapter 3.

In Ruby 1.9 and above, an alternative style of defining hashes is available, and you may see it in production code. It would allow the previous example to be written like so:

```ruby
fred = {
  name: 'Fred Elliott',
  age: 63,
  gender: 'male',
  favorite_painters: ['Monet', 'Constable', 'Da Vinci']
}
```

You should note that this style of defining hashes is an alternative shorthand syntax and is still not considered to be the standard, even in Ruby 1.9 and above.
Complex Structures

Because hashes and arrays can contain other objects, it’s possible to create complex structures of data. Here’s a basic example of a hash containing other hashes (and another hash containing an array at one point):

```ruby
people = {
  'fred' => {
    'name' => 'Fred Elliott',
    'age' => 63,
    'gender' => 'male',
    'favorite painters' => ['Monet', 'Constable', 'Da Vinci']
  },
  'janet' => {
    'name' => 'Janet S Porter',
    'age' => 68,
    'gender' => 'female'
  }
}
puts people['fred']['age']
puts people['janet']['gender']
puts people['janet'].inspect
```

63
female
{"name"=>'Janet S Porter", "age"=>68, "gender"=>"female"}

This example presents a hash called `people` that contains two entries with keys of 'fred' and 'janet', each of which refers to another hash containing information about each person. These sorts of structures are common in Ruby (as well as in Perl and C++). They are covered in more depth in Chapter 3 and throughout this book. Typically, compared to other languages, the syntax is simple, and in Ruby, the simplest answer is usually the right one.

Input/Output

Ruby has powerful input/output (I/O) support, from the ability to create, read, and manipulate files, through to database support, external devices, and network connectivity. These topics are covered in full in this book (primarily in Chapters 9, 14, and 15), but this section presents a basic overview of the most important forms of I/O.

Files

Ruby’s support for file I/O is powerful compared to that of other languages. Although Ruby supports traditional techniques for reading and manipulating files, its object-oriented features and tight syntax offer more exciting possibilities. First, here is the traditional way you’d open and read a file (as when using a more procedural language):

```ruby
lines = []
file_handle = File.open("/file/name/here", "r")
```
while line = file_handle.gets
  lines<< line
end

file_handle.close

---

**Note** You would need to replace `/file/name/here` with a legitimate path for this to work as-is.

This example opens a file in read-only mode, and then uses the file handle to read the file line by line before pushing it into an array. This is a reasonably standard technique in, say, C or Pascal. Let's look at a Ruby-specific technique:

```ruby
lines = File.readlines('/file/name/here')
```

Ruby's file handling and manipulation support is particularly deep and extensive, so it's out of the scope of this chapter. However, the preceding examples should have provided a glimpse into what's possible, and files are covered in full in Chapter 9 of this book.

### Databases

There are several ways to connect to database systems such as MySQL, MongoDB, PostgreSQL, Oracle, SQLite, and Microsoft SQL Server from Ruby. Typically, a “driver” library is available for each of the main database systems, although these don’t come with Ruby by default. You typically install database driver libraries using the RubyGems Ruby library packaging system, or you might need to download and install them manually. Explaining how to use such libraries is beyond the scope of this appendix, but they are covered in full in Chapter 9.

Ruby also has libraries that can provide more standardized interfaces to various driver libraries. Consider looking at `sequel` for this.

### Web Access

Ruby comes with libraries that make accessing data on the Web incredibly easy. At a high level is the open-uri library, which makes it easy to access data from the Web. This example retrieves a web page and returns an array containing all the lines on that page:

```ruby
require 'open-uri'
open('http://www.rubyinside.com/').readlines
```

open-uri is a convenience library that provides an open method that allows you to load data from URLs. `open` returns a File handle (technically a Tempfile object) that works in the same way as any other File object, allowing you to use methods such as `readlines` to read all the lines of the data into an array. (This topic is covered in significantly more depth in Chapter 14.)
Ruby also provides lower-level libraries, such as net/http. Here’s an example of retrieving a file from a web site and displaying it on the screen:

```ruby
require 'net/http'
Net::HTTP.start('www.rubyinside.com') do |http|
  req = Net::HTTP::Get.new('/test.txt')
  puts http.request(req).body
end
```

Hello Beginning Ruby reader!

This example connects to the web server at www.rubyinside.com and performs an HTTP GET request for /test.txt. This file’s contents are then returned and displayed. The equivalent URL for this request is http://www.rubyinside.com/test.txt, and if you load that URL in your web browser, you’ll get the same response as this Ruby program.

net/http also lets you make requests using other HTTP verbs such as POST and DELETE, and it is the most flexible HTTP library for Ruby. As it’s included with the standard library, it’s usually the first choice for most Ruby developers. Refer to Chapter 14 for full information.

## Libraries

This section looks at how you can organize code into multiple files and manage libraries within Ruby.

### File Organization

Ruby libraries don’t need to be packaged in any special way (unlike, say, Java’s JAR archives). Ruby does have a library packaging system called RubyGems (covered in the next section), but its use is entirely optional. The simplest way to create a library is to create a Ruby file containing classes and methods and use require to load it. This technique is similar in Perl (using use) or C (using #include).

Let’s assume you have a file called `mylib.rb` containing the following:

```ruby
class MyLib
  def self.hello_world
    puts "Hello, world!"
  end
end
```

And then you have another file like so:

```ruby
require_relative 'mylib'
MyLib.hello_world
```

This program loads in `mylib.rb` and includes its classes, methods, and other particulars into the current runtime environment, meaning that `MyLib.hello_world` calls the correct routine.

Ruby searches through its library folders in a specific order (and usually the current directory, too, as in the previous example), as dictated by the special variable `$:`. This variable is an array that can be manipulated like any other array. You can push, pop, and otherwise change the order and directories in which your program searches for libraries.
This topic is covered in depth in Chapter 7, and demonstrations of several Ruby libraries are offered in Chapter 16. A basic Ruby library is also created from scratch in Chapter 12.

Packaging

RubyGems ([http://rubygems.org/](http://rubygems.org/)) is a packaging system for Ruby libraries and applications. Each package within the RubyGems universe is called a gem or RubyGem (in this book, both terms are used interchangeably). RubyGems makes it easier to distribute, update, install, and remove libraries and applications on your system. A further system called Bundler makes it possible to “bundle” together gems in the context of a single Ruby project that you might be working on.

RubyGems has been included by standard with Ruby since Ruby 1.9, but was previously an optional, third-party technology.

Before the advent of RubyGems, Ruby libraries and applications were distributed in a basic fashion in archive files, or even as source code to copy and paste from the Web. RubyGems makes it easier and more centralized, and also takes care of any prerequisites and dependencies required when installing a library. For example, here’s how to install the Ruby on Rails framework:

```ruby
gem install rails
```

---

**Note** On some platforms, `sudo gem install rails` would be required so as to install the libraries as a superuser.

This installs the gems that make up Rails along with all their dependencies. The gem application prompts at each step of the way so you know exactly what’s being installed (you can override this with command-line options). For example, `gem install rails -y` installs Rails and its dependencies without questioning you at all. Bundler provides an alternative whereby gems are defined within a special file and then the Bundler tool automatically installs the required dependencies for you.

You can uninstall gems in as simple a fashion:

```ruby
gem uninstall rails
```

If you have multiple versions of the same gem(s) installed, gem will ask you which version(s) you want to remove.

By default, gems are searched for in the default repository, hosted at RubyGems.org. There is documentation on the official RubyGems site if you want to create your own account to be able to release your own gems via the site.

Optionally you can run your own gems repository on your own web site or by using the RubyGems server software. This is less common and requires users of your gems to specify your server name at the same time as installing the gem. I would not advise this.

RubyGems and Bundler are covered in Chapter 7, and several RubyGems are documented in Chapter 16.
APPENDIX B

Useful Resources

This appendix provides links to useful Ruby resources that are available online, from web sites to chat rooms and mailing lists.

As the Internet is ever changing, some resources that were available at the time of writing may no longer be available to you. When you find that to be the case, it’s worth using a search engine to search for the keywords involved, as the site you’re looking for might have simply changed URLs.

Tutorials and Guides

The Internet is host to a significant number of tutorials and guides on how to use various features of Ruby and its libraries. Often there are multiple tutorials on how to do the same thing in different ways, and tutorials can appear quickly after libraries are released. This is why it’s worth subscribing to a few Ruby-related Twitter feeds and other news sources so that you can learn about the latest action as it happens.

However, in this section are links to a number of useful tutorials and guides that are more perennially useful.

General Ruby Tutorials

*Learn Ruby* ([http://rubylearning.com](http://rubylearning.com)): A collection of short tutorials and e-books on various aspects of Ruby, by Satish Talim. It’s ideal as a quick recap on various topics. Satish also runs Ruby-related online classes.


*Ruby in Twenty Minutes* ([http://www.ruby-lang.org/en/documentation/quickstart/](http://www.ruby-lang.org/en/documentation/quickstart/)): A basic primer to the bare essentials of Ruby. This guide won’t be of any use to readers of this book, but might be useful to forward to others who are interested in Ruby and want to get a quick look at the language from a beginner’s point of view.
Ruby on Rails

*Getting Started with Rails* ([http://guides.rubyonrails.org/getting_started.html](http://guides.rubyonrails.org/getting_started.html)): An excellent walkthrough of how to use Rails from a basic point of view. Covers creating a very basic application and provides links to further resources. Well worth reviewing after reading Chapter 13 of this book.

*The Rails Tutorial Book* ([https://www.railstutorial.org/book](https://www.railstutorial.org/book)): A book by Michael Hartl that is available to read in its entirely online. This is what I recommend if you want to learn Rails from scratch.

*Railscasts* ([http://railscasts.com/](http://railscasts.com/)): A site full of screencasts covering Rails- and Ruby-related topics. It has been abandoned since 2013 but still has hundreds of videos you can watch.

Other

*REXML Tutorial* ([http://www.germane-software.com/software/rexml/docs/tutorial.html](http://www.germane-software.com/software/rexml/docs/tutorial.html)): A tutorial giving lots of quick code snippets showing how to use the REXML XML-processing library that comes in the Ruby standard library.

*Using the Rake Build Language* ([http://www.martinfowler.com/articles/rake.html](http://www.martinfowler.com/articles/rake.html)): A comprehensive run through the Rake (Ruby Make) system, written by Martin Fowler. The Rake system was briefly mentioned in Chapter 13 of this book.

*SQL Tutorial* ([http://www.w3schools.com/sql/](http://www.w3schools.com/sql/)): A comprehensive SQL tutorial, expanding on what is covered in Chapter 9 of this book.

References

The resources covered in this section are general references to Ruby and Ruby on Rails. For specific tutorials and guides to doing certain things, you need to refer instead to the “Tutorials and Guides” section later on in this appendix.

Ruby


*Ruby-Doc.org* ([http://www.ruby-doc.org/](http://www.ruby-doc.org/)): A documentation site built by the Ruby community that features documentation for the core API, standard libraries, and other miscellaneous Ruby bits and pieces. Its primary maintainer is James Britt, who has been involved with Ruby documentation for many years.

*Ruby Core Documentation* ([http://www.ruby-doc.org/core/](http://www.ruby-doc.org/core/)): Documentation for the core elements of Ruby 2.3 (at the time of writing), such as the included classes (Array, Hash, and so on), as well as most of the standard library. This URL will redirect to the documentation for the latest production version of Ruby as it changes over time.
Ruby 2.0.0 Documentation (http://www.ruby-doc.org/core-2.0.0/): If, for some reason, you need to access documentation for older versions of Ruby, you can do by changing the version number in the URL, as in this case.


Ruby Standard Library Documentation (http://www.ruby-doc.org/stdlib/): Documentation for the Ruby standard libraries in Ruby 2.3 (at the time of writing). Each library is presented separately, making it easier to read than the core documentation.


Ruby Quickref (http://www.zenspider.com/Languages/Ruby/QuickRef.html): A quick-fire set of references and reminders that act as a cheat sheet for Ruby, listing reserved words, regular expression syntax, language constructions, special variables, and more.

Ruby on Rails

Official Rails Homepage (http://rubyonrails.org/): The official homepage for the Ruby on Rails framework. It features screencasts, tutorials, and links to many useful Rails references.

Rails API Documentation (http://api.rubyonrails.org/): API documentation for the entire Ruby on Rails framework in RDoc format. This is the most useful reference documentation for Ruby on Rails, as almost all Rails techniques and methods are covered.

Ruby on Rails Guides (http://guides.rubyonrails.org/): Well-written walkthrough guides for various Rails features, such as how to get started with Rails, and how to use the internationalization features, routing, and database migrations.

Ruby-Related Content

Aggregators and News

RubyFlow (http://www.rubyflow.com/): A community-driven link blog for all things related to Ruby and Rails. It’s very popular and a great way to keep up with the day-to-day Ruby news and to promote your own blog posts.

Ruby News (https://www.ruby-lang.org/en/news/): The official news site for the main implementation of Ruby. It is only updated sporadically and when there are key release or security announcements.
/r/ruby on Reddit (http://reddit.com/r/ruby): An area of the popular Reddit community discussion and bookmarking site dedicated to Ruby-related items.

Riding Rails (http://weblog.rubyonrails.org/): The official blog for Ruby on Rails, updated by several core Rails developers and activists. The blog focuses on sporadic announcements of interesting uses or deployments of Rails, as well as new Rails features.

Ruby Weekly (http://rubyweekly.com/): A weekly Ruby and Rails e-mail newsletter with almost 40,000 subscribers. It’s produced by your humble author and is highly recommended if you want to stay up to date with Ruby news on a frequent basis.

Podcasts

Ruby on Rails Podcast (http://5by5.tv/rubyonrails): A weekly podcast focusing on conversations around Rails and matters that affect Rails developers.

Ruby Rogues (http://devchat.tv/ruby-rogues/): A regular podcast oriented around a regular panel of developers who discuss Ruby and Rails related topics with a special guest.

Ruby5 (https://ruby5.codeschool.com/): A weekly podcast that quickly covers the latest Ruby news in several minutes each time.

Forums

Rails Forum (http://railsforum.com/): A popular Ruby on Rails help and discussion forum.

SitePoint Ruby Forum (https://www.sitepoint.com/community/c/ruby): A Ruby forum provided by the SitePoint webmaster resources site. Unlike with Rails Forum, all posts to the SitePoint Ruby forum are within a single category, making it easier to scan through.

Mailing Lists

Mailing lists are like forums, but based on e-mail. People subscribe to a “list,” and then all messages sent to that list are received by all the subscribers. There are also archives of e-mail lists available on the Web for reference or for those who don’t want to sign up for the list.


Ruby-Talk Mailing List: Ruby-Talk is the most popular Ruby mailing list, where all aspects of Ruby development are discussed. You can subscribe via the link above.

Ruby-Talk Mailing List Archives (http://blade.nagaokaut.ac.jp/ruby/ruby-talk/index.shtml): Offers web access to more than 400,000 posts made to the Ruby-Talk mailing list and includes a search feature.

ruby-core (http://blade.nagaokaut.ac.jp/ruby/ruby-core/index.shtml): A mailing list dedicated to discussing implementation details and the development of Ruby. Those who are developing the Ruby language use this list. However, it isn’t a list on which to ask general Ruby questions.

---

Note It’s important when using a mailing list that you look at the format and tone of other posts and don’t offend anyone. If your postings sound too demanding or are of the wrong tone, you might not get any responses.

---

Real-Time Chat

On the Internet, there are several ways you can discuss topics with other users in real time. For example, websites can contain Flash or Java chatrooms. Alternatively, you can use instant messenger or Internet Relay Chat (IRC) clients. Ruby is the primary focus of discussion in only a few real-time chat systems at present:

#rubyonrails (irc://irc.freenode.net/%23rubyonrails): #rubyonrails was the official Ruby on Rails IRC channel and still exists. You can ask questions about Ruby on Rails here, and most people are willing to help. The popularity of this channel has remained high, and it can be very busy much of the time.

#ruby (irc://irc.freenode.net/%23ruby): #ruby used to be a quieter, more generic Ruby channel, but since the #ruby-lang channel went invite only, it has become the de facto official channel for discussing Ruby generally and has a significant number of participants.

You can also take part in these channels by using Freenode’s web-based client at https://webchat.freenode.net/.

---

Note If you aren’t familiar with IRC, visit http://en.wikipedia.org/wiki/Internet_Relay_Chat to learn more.
Index

A
ActionMailer, 357
ActiveRecord and model relationships, 327–328
add_accessor method, 246
attr_accessor method, 246

B
Base64
Binary Data conversion, 381–382
Compressio, 382
installation, 381
Benchmarking and Profiling
Benchmark.measure do, 181
bmbm method, 182
profiling code, 182
Benchmarking C vs. Ruby, 264
Benchmark module, 383–384
Bots
basic client, 297–298
Code Listing, 295–297
contact information management tool, 270
data structure, 283
ELIZA-type, 269
external data storage, 284–286
fred.bot, 292–294
history, 269
preprocessing—tasks, 271
processing language, 271
pronoun-switching feature, 274–275
sentences method, 271–272
testing and documentation, 271
Text Files, 298
Web connection, 299–300
Bot-to-Bot conversations, 301–302
Bundler, 159

C
CentOS, 310
chroniclibrary, 389–390
class_eval method, 245
Class methods vs. instance methods, 114–116
Command line arguments, 230
Common Gateway Interface (CGI), 236
  basic CGI script, 385
cgilibrary, 385
cookies, 387–388
Passed Data, 386
sessions, 388–389

D
Debugging
  exceptions and error handling
    Catch and Throw, 172
    commands prompt, 173
dobugtest.rb, 173
  handling exceptions, 170
  raised exceptions, 169
receive exceptions, 171
Deleting directory, 203
Digest, 390–393
Directory creation, 202
Documentation, RDoc
  basic formatting, 165
  command-line option, 168
  creation, 163
 nodoc Modifier, 167
  processing comments, 167
techniques, 165
  web browser, 164
Document Object Model (DOM), 256
Dynamic code Execution
  add_accessor method, 246
Dynamic code Execution (cont.)
  attr_accessor method, 246
  bindings, 244
  class_eval method, 245
  instance_eval method, 246

E
  E-mail, 354
  encode method, 266
  English library, 393–394
  Environment variables, 228
  ERB library, 395–397
  Extensible Markup Language (XML), REXML, 352–354

F
  Fibers
    cooperative scheduling, 259
    data passing, 260
    disadvantage, 261
    Fiber.yield method, 260
    implementation, 260
    resume method, 260
  Fiber.yield method, 260
  File I/O
    character encodings, 196
    deleting file, 197
    directory, 201
    file modes, 195
    Opening and Reading Files, 189
    operations, 197
    pos method, 194
    techniques, 190
    writing to files, 194
  Files and databases
    I/O streams
      File I/O. File I/O
      keyboard input, 187
    objects and data structures
      PStore, 207
      YAML, 208
    text file
      CSV data, 205
      saving data, 206
  FTP system, 358–359

G, H
  Generic sentence-splitting method, 271
  getbinaryfile, 360
  Global interpreter lock (GIL), 257
  GNOME Editor (gedit), 74
  Greenwich Mean Time (GMT), 62
  GServer, 372–374
  GServer-Based Chat Server, 374–377

I
  initialize method, 286
  instance_eval method, 246
  Interactive Ruby interpreter, 420
  Interactive Ruby (irb), 12
  Internet
    Basic Authentication scheme, 346
    HTTP, 343
    net/http Library, 343–345
    Proxying, 347–348
    Internet Protocol (IP), 363
    Internet Relay Chat (IRC), 100
    IronRuby, 97

J
  join method, 48
  JRuby, 97
  JavaScript Object Notation (JSON), 397–398

K
  KDE Advanced Text Editor (Kate), 74
  Kernel methods, 22

L, M
  length method, 80
  Local area network (LAN), 363
  Logger, 399–401

N
  Navigating directory, 201
  Networking
    description, 363
    DNS queries, 365–366
    IP Addresses and DNS, 364
    machine and service availability, 364–365
    TCP and UDP, 363
    TCP Server connection, 367–368
  Nokogiri, 351–352, 401–402

O
  Object orientation
    classes and methods, 110, 427–429
    class variables, 113
    Comparable module, 135–136
    concepts and processes, 107
    constants, 128
    encapsulation, 121, 123–125
    Enumerable module, 132–134
    global variables, 112
inheritance, 116–118
instant variables, 112
iteration process, 132
local variables, 111–112
Mix-Ins, Namespaces and Classes, 137
modules, 129
multiple inheritance, 131–132
namespaces, 129–130
Nested Classes, 127
objects, 110, 427
override existing classes and methods, 118–119
polymorphism, 109, 125–126
procedural programmer coding, 107
procedural style of programming, 107
Reflection, 120–121
Open source culture, 99
open-uri library, 349–350

P, Q
Popen method, 249
Post Office Protocol 3 (POP3), 354–356
ppis, 402–404
Program's lifecycle, 281–282

R
Rails
books, 332
Command-Line Tool, 311–313
Controller Actions, 322–324
Controllers, 307
CRUD mechanisms, 310
database configuration, 315, 317
database considerations, 310
database migrations, 318–322
files and directories, 313–314
Getting Started with Rails, 332
installation, 309
layouts, 330–331
models, 307
model-view-controller (MVC) architectural pattern, 307
Official Ruby on Rails API, 332
open source web application, 306
Ruby on Rails guides, 332
Ruby's killer app, 306
scaffolding, 317–318
sessions and filters, 329–330
Views, 307
readlines method, 80
RedCarpet, 404–405
Red Hat, 310
Reflective language, Ruby, 429–430
Relational Databases and SQL
ActiveRecord, 222
add data, 213
concepts, 210
CREATE TABLE SQL statement, 217
database connection code, 219
definition, 212
DELETE command, 216
gem install sequel, 221
INSERT command, 214
install SQLite3, 212
MySQL, PostgreSQL and SQLite, 211
retrieve data, 218
SELECT command, 214
Sequel.connect, 221
SQLiteRuby gem, 217
UPDATE command, 216
response_to Method
best_sentence, 289
matching phrases, 289–290
phrase substitutions, 287
possible_responses, 290–291
preprocessing word, 287
resume method, 260
REXML, 352–354
Rubinius, 97
Ruby
ARGV array, 89–90
arrays, 436
coection, 421
comparison expressions, 422–423
complex structures, 438
conditional execution, 423–424
definition and concepts, 417–419
direction method, 428
expressions, 420–421
File Organization, 440
forking, 248
hashes (associative arrays), 437
input/output (I/O) support
database systems, 439
files, 438–439
integers and floating point numbers, 434–435
Internet (see Internet)
Interpreter, 419
method visibility, 431–432
Popen method, 249
regular expressions, 433–434
safe levels, 253
source code
creation, 75
Linux, 76
Mac OS X, 76
UNIX-based systems, 76
windows, 75
source code files, 73
Ruby (cont.)
- stop words, 85
- strings, 433
- summarization features, 87
- system method, 248
- tainted method, 250
- ternary operator, 425–426
- test file creation
  - Linux, 74
  - visual studio code, 74
- web access, 439
- Windows users, 227
RubyGems, 235, 309, 441
- bundler, 234–235
- creation, 231
- distribution, 235
- gem build, 234
- string_extend.gemspec, 232
- structuring, 232
RubyInline
- benchmarking, 263
- C function, 263
- C language, 262
- definition, 262
- factorial method, 262–263
- installation, 262
Ruby installation
- Atari ST, 9
- BeOS, 9
- definition, 3
- Linux
  - package managers, 8
  - source code, 8
- Mac OS X, 6
- MS-DOS, 9
- Windows, 4
Ruby libraries
- abbrev library, 379–380
- definition, 153
- gems
  - bundler, 159
  - command list, 156
  - installation, 157
  - loading, 157
  - Uninstallation, 158
  - upgrading, 158
- net/http, 154
- OpenStruct, 155
- RubyGems, 156
- RUBY_PLATFORM, 228
Ruby programming, 225
- attributes, 20
- classes, 16, 19
- experimentation, 14
- irb, 12
- Kernel methods, 22
- natural language, 13
- non object oriented, 26
- object-oriented, 15, 18, 21
- parameter passing methods, 23
- String class, 25
- syntax, 12
- testing, 14
- variables, 17
Ruby, projects
- basic file inclusion, 149
- definition, 149
- directories, 152
- logic and including code, 152
- nested inclusions, 153
Ruby resources
- aggregators and news, 445
- forums, 446
- guides, 444
- internet, 443
- mailing lists, 446
- podcasts, 446
- rails, 444–445
- real time chat, 447
- tutorials, 443–444
Ruby’s building blocks
- array addition and concatenation, 51
- array iteration, 50
- array order reversal, 52
- array subtraction, 51
- comparison operators, 32
- constants, 36
- date/time Attributes, 65
- empty array, 51
- expressions, 29
- first and last methods, 52
- floating point numbers, 35
- flow control
  - code blocks, 61
  - else if and case, 58
  - if and unless, 56
  - ternary operator, 57
  - while and until, 60
- hashes
  - definition, 53
  - DELETE method, 54–55
  - Fred’s hash, 56
  - iteration, 54
  - retrieving keys, 54
- include? method, 52
- inspect method, 49
- interpolation, 39
- large numbers, 65
- loop implementation, 33
object conversion, 69
ranges, 67
regular expression
iteration, 43
matching, 46
substitutions, 42
split method, 49
string expressions, 37
string literal, 36
string methods, 40
symbols, 68
variables, 30
Ruby’s ecosystem
code contribution, 102
community, 102
documentation, 101
forums, 101
history of, 93
homepage 2000, 95
homepage 2015, 96
implementations, 97
IRC, 100
mailing lists, 100
open source, 99
ruby-lang.org, 95
ruby-talk, 95
sites and podcasts, 103
Web 2.0, 98
Ruby’s networking
CGI script, 236
CGI variables, 237
HTTP servers, 238
Ruby threading
cooperative multitasking, 259
join method, 258
run method, 259
system-based threads, 257
Thread.list, 258
S
scan method, 82
Secure Socket Layer (SSL), 348
SELECT method, 86
Sentence separations, 277
Shebang Line, 226
Simple Mail Transfer Protocol (SMTP), 356
Sinatra
Error Handling, 340–341
external templates and layouts, 338–339
Extreme Simplicity, 333
Halting, 340
Inline and In-File Templates, 335, 337
layouts, 337
URL routing and parameter
matching, 334–335
Sockets
GServer, 372–374
Multi-Client TCP Servers, 372
TCP servers, 370–371
UDP Client and Server, 368–370
split method, 81
Square.count class method, 116
StringScanner, 405–407
switch_pronouns method, 278
T
Temporary files (tempfile), 408–410
Testing
assertions types, 180
behavior-driven development, 176
test-driven development, 175
unit testing, 177
Text adventure/virtual dungeon
classes, 142
concepts, 138
go method, 145
Initial Classes, 138–139
show_current_description method, 143–144
 structs, 139–140
Text analyzer
analyser.rb, 90
average calculations, 84
basic statistics, 77
counting characters, 80
counting lines, 79
counting paragraphs, 83
counting sentences, 82
counting words, 81
key steps, 78
loading text files, 79
source code, 84
test data, 78
Text editors vs. source code editors, 76
Transmission Control Protocol (TCP), 363
U
UDP Client and Server, 368–370
Unicode Transformation Format-8 (UTF-8)
character encodings, 265
source code, 266
strings, 265
unicode, 265
Uniform Resource Identifiers (URIs), 410–413
Uploading files, FTP, 360–361
User Datagram Protocol (UDP), 363
Views and Embedded Ruby (ERB), 324–327
vowels method, 231

Web/HTTP Servers, 377
WEBrick server, 239
Windows API, 253
Windows Automation, 256

Word Matching, 273
WordPlay.best_sentence method, 277–278
WordPlay library
  test_wordplay.rb, 280
  wordplay.rb, 279
  words method, 277

zlibis, 414–415