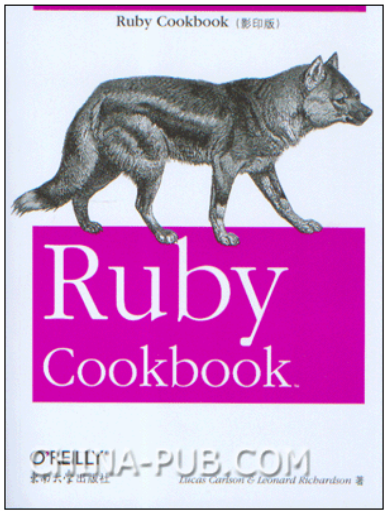
Abstract of Ruby Cookbook



The Ruby programming language is itself a wonderful time-saving tool. It makes you more productive than other programming languages because you spend more time making the computer do what you want, and less wrestling with the language.

String

Strings are dynamic, mutable, and flexible.

In Ruby, everything that can be assigned to a variable is an object. Here, the variable **string** points to an object of class **String**.

In Ruby, parentheses are almost always optional. They're especially optional in this case, since we're not passing any arguments into **String#length**. If you're passing arguments into a method, it's often more readable to enclose the argument list in parentheses.

Many programming languages (notably Java) treat a string as a series of characters. Ruby treats a string as a series of bytes. The French string contains 14 letters and 3 spaces, so you might think Ruby would say the length of the string is 17. But one of the letters (the e with acute accent) is represented as two bytes, and that's what Ruby counts is 18.

You can represent special characters in strings (like the binary data in the French string) with string escaping. Ruby does different types of string escaping depending on how you create the string. When you enclose a string in single quotes, the only special codes you can use are "\'" to get a literal single quote, and "\\" to get a literal backslash.

```
puts "This string\ncontains a newline"
puts 'it may look like this string contains a newline\nbut it doesn\'t'
puts 'Here is a backslash: \\'
```

Another useful way to initialize strings is with the " here documents" style:

```
long_string = <<EOF
Here is a long string
With many paragraphs
EOF
# => "Here is a long string\nWith many paragraphs\n"
```

But if you're coming from C, and you think of a string as an array of bytes, Ruby can accommodate you. Selecting a single byte from a string returns that byte as a number.

```
string = "My first string"
string[3].chr + string[4].chr + string[5].chr + string[6].chr + string[7].chr
# => "first"
```

Unlike in most programming languages, Ruby strings are mutable: you can change them after they are declared.

This is one of Ruby's syntactical conventions. "Dangerous" methods

(generally those that modify their object in place) usually have an exclamation mark at the end of their name. Another syntactical convention is that predicates, methods that return a true/false value, have a question mark at the end of their name (as in some varieties of Lisp).

This use of English punctuation to provide the programmer with information is an example of Matz's design philosophy: that Ruby is a language primarily for humans to read and write, and secondarily for computers to interpret.

In Ruby, though, strings are just as mutable as arrays. Just like arrays, they can expand as needed, without using much time or memory. The fastest solution to this problem in Ruby is usually to forgo a holding array and tack the substrings directly onto a base string. Sometimes using **Array#join** is faster, but it's usually pretty close, and the **<<** construction is generally easier to understand. If efficiency is important to you, don't build a new string when you can append items onto an existing string. Constructs like **str << 'a' + 'b'** or **str << "#{var1} #{var2}"** create new strings that are immediately subsumed into the larger string. This is exactly what you're trying to avoid. Use **str << var1 <<"<< var2** instead.

On the other hand, you shouldn't modify strings that aren't yours. Sometimes safety requires that you create a new string.

Ruby supports more complex string substitutions as well. Any text kept within the brackets of the special marker #{} (that is, #{text in here}) is interpreted as a Ruby expression. The result of that expression is substituted into the string that gets created. If the result of the expression is not a string, Ruby calls its **to_s** method and uses that instead.

You can use string interpolation to run even large chunks of Ruby code inside a string. You should never have any reason to do this, but it shows the power of this feature.

```
%{Here is #{class InstantClass
    def bar
        "some text"
    end
end
InstantClass.new.bar
}.}
# => "Here is some text."
```

If a string interpolation calls a method that has side effects, the side effects are triggered. If a string definition sets a variable, that variable is accessible afterwards.

To avoid triggering string interpolation, escape the hash characters or put the string in single quotes.

```
"\#{foo}"                       # => "\#{foo}"
```

```
'#{foo}'                        # => "\#{foo}"
```

The "here document" construct is an alternative to the **%{}** construct, which is sometimes more readable.

Ruby supports a printf-style string format like C's and Python's. Put printf directives into a string and it becomes a template. You can interpolate values into it later using the modulus operator:

```
'To 2 decimal places: %.2f' % Math::PI      # => "To 2 decimal places: 3.14"
```

An ERB template looks something like JSP or PHP code. Most of it is treated as a normal string, but certain control sequences are executed as Ruby code. The control sequence is replaced with either the output of the Ruby code, or the value of its last expression.

```
require 'erb'
template = ERB.new %q{Chunky <%= food %>!}
food = "bacon"
puts template.result(binding)                  # => "Chunky bacon!"
```

When you call **ERB#result**, or **ERB#run**, the template is executed according to the current values of those variables. You can omit the call to **Kernel#binding** if you're not in an **irb** session.

Because the regular expression **/(\s+)/** includes a set of parentheses, the separator strings themselves are included in the returned list.

```
"Three little words".split(/\s+/)   # => ["Three", "little", "words"]
"Three little words".split(/(\s+)/) # => ["Three", " ", "little", " ", "words"]
```

The most common unprintable characters (such as newline) have special mneumonic aliases consisting of a backslash and a letter.

```
"\a" == "\x07" # => true # ASCII 0x07 = BEL (Sound system bell)
"\b" == "\x08" # => true # ASCII 0x08 = BS (Backspace)
"\e" == "\x1b" # => true # ASCII 0x1B = ESC (Escape)
"\f" == "\x0c" # => true # ASCII 0x0C = FF (Form feed)
"\n" == "\x0a" # => true # ASCII 0x0A = LF (Newline/line feed)
"\r" == "\x0d" # => true # ASCII 0x0D = CR (Carriage return)
"\t" == "\x09" # => true # ASCII 0x09 = HT (Tab/horizontal tab)
"\v" == "\x0b" # => true # ASCII 0x0B = VT (Vertical tab)
```

Ruby stores a string as a sequence of bytes. It makes no difference whether those bytes are printable ASCII characters, binary characters, or a mix of the two.

When Ruby prints out a human-readable string representation of a binary character, it uses the character's \xxx octal representation. Characters with

special \x mneumonics are printed as the mneumonic. Printable characters are output as their printable representation, even if another representation was used to create the string.

```
p "\x48\145\x6c\x6c\157\x0a"    # => "Hello\n"
p "\x10\x11\xfe\xff"           # => "\020\021\376\377"
```

To avoid confusion with the mneumonic characters, a literal backslash in a string is represented by two backslashes.

```
p "\\".size                  # => 1
p "\\" == "\x5c"              # => true
p "\\n"[0] == ?\\             # => true
p "\\n"[1] == ?n              # => true
```

Ruby also provides special shortcuts for representing keyboard sequences like Control-C.

```
p "\C-a\C-b\C-c" #            => "\001\002\003"
p "\M-a\M-b\M-c" #            => "\341\342\343"
```

Shorthand representations of binary characters can be used whenever Ruby expects a character. For instance, you can get the decimal byte number of a special character by prefixing it with ?, and you can use shorthand representations in regular expression character ranges.

```
?\C-a                         # => 1
?\M-z                         # => 250
```

Special characters are only interpreted in strings delimited by double quotes, or strings created with **%{}** or **%Q{}**. They are not interpreted in strings delimited by single quotes, or strings created with **%q{}**. You can take advantage of this feature when you need to display special characters to the end-user, or create a string containing a lot of backslashes.

If you come to Ruby from Python, this feature can take advantage of you, making you wonder why the special characters in your single-quoted strings aren't treated as special. If you need to create a string with special characters and a lot of embedded double quotes, use the %{} construct.

To see the ASCII code for a specific character as an integer, use the ? operator:

```
puts ?a             # => 97
puts ?!             # => 33
puts ?\n            # => 10
```

To see the integer value of a particular in a string, access it as though it were an element of an array:

```
p 'a'[0]           # => 97
p 'bad sound'[1]    # => 97
```

To see the ASCII character corresponding to a given number, call its #chr method. This returns a string containing only one character:

```
p 97.chr           # => "a"
p 33.chr           # => "!"
p 10.chr           # => "\n"
p 0.chr            # => "\000"
p 256.chr          # RangeError: 256 out of char range
```

To turn a symbol into a string, use Symbol#to_s, or Symbol#id2name, for which to_s is an alias. You usually reference a symbol by just typing its name. If you're given a string in code and need to get the corresponding symbol, you can use **String.intern.**

```
p :AnotherSymbol.id2name             # => "AnotherSymbol"
p :"Yet another symbol!".to_s         # => "Yet another symbol!"

p :dodecahedron.object_id             # => 4565262
symbol_name = "dodecahedron"
p symbol_name.intern                 # => :dodecahedron
p symbol_name.intern.object_id        # => 4565262
```

A Symbol is about the most basic Ruby object you can create. It's just a name and an internal ID. Symbols are useful becase a given symbol name refers to the same object throughout a Ruby program.

Symbols are often more efficient than strings. Two strings with the same contents are two different objects (one of the strings might be modified later on, and become different), but for any given name there is only one Symbol object. This can save both time and memory.

```
p "string".object_id        # => 1503030
p "string".object_id        # => 1500330
p :symbol.object_id         # => 4569358
p :symbol.object_id         # => 4569358
```

It's also faster to compare two symbols than to compare two strings, because Ruby only has to check the object IDs.

Finally, to quote Ruby hacker Jim Weirich on when to use a string versus a symbol:

- If the contents of the object are important, use a string.
- If the identity of the object is important, use a symbol.

If you're processing an ASCII document, then each byte corresponds to one character. Use **String#each_byte** to yield each byte of a string as a number, which

you can turn into a one-character string.

Use **String#scan** to yield each character of a string as a new one-character string.

```
'foobar'.each_byte { |x| puts "#{x} = #{x.chr}" }
# 102 = f
# 111 = o
# 111 = o
# 98 = b
# 97 = a
# 114 = r

'foobar'.scan( /./ ) { |c| puts c }
# f
# o
# o
# b
# a
# r
```

String#each_byte is faster than String#scan, so if you're processing an ASCII file, you might want to use String#each_byte and convert to a string every number passed into the code block (as seen in the Solution).

String#scan works by applying a given regular expression to a string, and yielding each match to the code block you provide. The regular expression /./ matches every character in the string, in turn.

To change the case of specific letters while leaving the rest alone, you can use the **TR** or **TR!** methods, which translate one character into another.

```
p 'LOWERCASE ALL VOWELS'.tr('AEIOU', 'aeiou')
```

Use **strip** to remove whitespace from the beginning and end of a string. To remove whitespace from only one end of a string, use the **lstrip** or **rstrip** method.

```
p " \tWhitespace at beginning and end. \t\n\n".strip
s = "   Whitespace madness! "
p s.lstrip                    # => "Whitespace madness! "
p s.rstrip                    # => "   Whitespace madness!"
```

The **upcase** and **downcase** methods force all letters in the string to upper-or lowercase, respectively. The **swapcase** method transforms uppercase letters into lowercase letters and vice versa. The **capitalize** method makes the first character of the string uppercase, if it's a letter, and makes all other letters in the string lowercase. All four methods have corresponding methods that modify a string in place rather than creating a new one: **upcase!**, **downcase!**, **swapcase!**, and **capitalize!**.

Add whitespace to one or both ends of a string with **ljust, rjust**, and **center.**

```
s = "Some text."
p s.center(15)
p s.ljust(15)
p s.rjust(15)
```

Use the **gsub** method with a string or regular expression to make more complex changes, such as to replace one type of whitespace with another.

Whenever possible, you should treat objects according to the methods they define rather than the classes from which they inherit or the modules they include.

The idea to take to heart here is the general rule of duck typing: to see whether provided data implements a certain method, use **respond_to**? instead of checking the class.

To get the first portion of a string that matches a regular expression, pass the regular expression into **slice** or **[]**.

```
s = 'My kingdom for a string!'
p s[/.ing/]                    # => "king"
p s[/str.*/]                   # => "string!"
```

There's no reverse version of String#succ. Matz, and the community as a whole, think there's not enough demand for such a method to justify the work necessary to handle all the edge cases. If you need to iterate over a succession of strings in reverse, your best bet is to transform the range into an array and iterate over that in reverse:

```
("a.."e").to_a.reverse_each { |x| puts x }
# e
# d
# c
# b
# a
```

You can check a string against a series of regular expressions with a **case** statement.

```
string = "12f35"
case string
when /^[a-zA-Z]+$/
 "Letters"
when /^[0-9]+$/
 "Numbers"
else
 "Mixed"
```

```
end
# => "Numbers"
```

Ruby provides several ways of initializing regular expressions. The following are all equivalent and create equivalent Regexp objects:

```
/something/
Regexp.new("something")
Regexp.compile("something")
%r{something}
```

Regexp::IGNORECASE	i	Makes matches case-insensitive.
Regexp::MULTILINE	m	Normally, a regexp matches against a single line of a string. This will cause a regexp to treat line breaks like any other character.
Regexp::EXTENDED	x	This modifier lets you space out your regular expressions with whitespace and comments, making them more legible.

```
/something/mxi
Regexp.new('something',
          Regexp::EXTENDED + Regexp::IGNORECASE + Regexp::MULTILINE)
%r{something}mxi
```

Numbers

The first distinction is between small numbers and large ones. If you've used other programming languages, you probably know that you must use different data types to hold small numbers and large numbers (assuming that the language supports large numbers at all). Ruby has different classes for small numbers (Fixnum) and large numbers (Bignum), but you don't usually have to worry about the difference. When you type in a number, Ruby sees how big it is and creates an object of the appropriate class.

```
p 1000.class                    # => Fixnum
p 10000000000.class             # => Bignum
```

When you perform arithmetic, Ruby automatically does any needed conversions. You don't have to worry about the difference between small and large numbers.

Like all modern programming languages, Ruby implements the IEEE floating-point standard for representing fractional numbers. If you type a number that includes a decimal point, Ruby creates a Float object instead of a Fixnum or Bignum:

```
p 0.01.class                    # => Float
p 1.0.class                     # => Float
p 10000000000.00000000001.class # => Float
```

Use String#to_i to turn a string into an integer. Use String#to_f to turn a string into a floating-point number.

```
p '400'.to_i                    # => 400
p '3.14'.to_f                   # => 3.14
```

Unlike Perl and PHP, Ruby does not automatically make a number out of a string that contains a number. You must explicitly call a conversion method that tells Ruby how you want the string to be converted.

If you have a string that represents a hex or octal string, you can call **String#hex** or **String#oct** to get the decimal equivalent. This is the same as passing the base of the number into **to_i.**

```
p '405'.oct                     # => 261
p '405'.to_i(8)                 # => 261
p '405'.hex                     # => 1029
p '405'.to_i(16)                # => 1029
p 'fed'.hex                     # => 4077
p 'fed'.to_i(16)                # => 4077
```

If to_i, to_f, hex,or oct find a character that can't be part of the kind of number they're looking for, they stop processing the string at that character and

return the number so far. If the string's first character is unusable, the result is zero.

```
p "13: a baker's dozen".to_i                      # => 13
p '1001 Nights'.to_i                             # => 1001
p 'The 1000 Nights and a Night'.to_i               # => 0
p '60.50 Misc. Agricultural Equipment'.to_f         # => 60.5
p '$60.50'.to_f                                  # => 0.0
p 'Feed the monster!'.hex                          # => 65261
p 'I fed the monster at Canoga Park Waterslides'.hex # => 0
```

If you want an exception when a string can't be completely parsed as a number, use **Integer( )** or **Float( )**.

```
p Integer('1001')                                # => 1001
Integer('1001 nights')
# ArgumentError: invalid value for Integer: "1001 nights"

p Float('99.44')                                 # => 99.44
Float('99.44% pure')
# ArgumentError: invalid value for Float(): "99.44% pure"
```

Floating-point numbers are not suitable for exact comparison. Often, two numbers that should be equal are actually slightly different. You can avoid this problem altogether by using **BigDecimal** numbers instead of floats.

You can avoid this problem altogether by using **BigDecimal** numbers instead of floats. **BigDecimal** numbers are completely precise, and work as well as as floats for representing numbers that are relatively small and have few decimal places: everyday numbers like the prices of fruits. But math on **BigDecimal** numbers is much slower than math on floats.

A **BigDecimal** number can represent a real number to an arbitrary number of decimal places.

```
nm = "0.123456789012345678901234567890123456789"
p nm.to_f                     # => 0.123456789012346
p BigDecimal(nm).to_s
# => "0.123456789012345678901234567890123456789E0"
```

BigDecimal numbers store numbers in scientific notation format. A **BigDecimal** consists of a sign (positive or negative), an arbitrarily large decimal fraction, and an arbitrarily large exponent. You can use BigDecimal#split to split a BigDecimal object into the parts of its scientific-notation representation. It returns an array of four numbers: the sign (1 for positive numbers, -1 for negative numbers), the fraction (as a string), the base of the exponent (which is always 10), and the exponent itself.

```
p BigDecimal("105000").split
```

```
# => [1, "105", 10, 6]
# That is, 0.105*(10**6)

p BigDecimal("-0.005").split
# => [-1, "5", 10, -2]
# That is, -1 * (0.5*(10**-2))
```

Use a Rational object; it represents a rational number as an integer numerator and denominator. **Rational** objects can store numbers that can't be represented in any other form, and arithmetic on **Rational** objects is completely precise.

```
require 'rational'
rational = Rational(2, 3)          # => Rational(2, 3)
p rational.to_f                    # => 0.666666666666667
p rational * 100                   # => Rational(200, 3)
p rational * 100 / 42              # => Rational(100, 63)
```

The methods in Ruby's Math module implement operations like square root, which usually give irrational results. When you pass a Rational number into one of the methods in the Math module, you get a floating-point number back:

```
p Math::sqrt(Rational(25,1))       # => 5.0
p Math::log10(Rational(100, 1))    # => 2.0
```

The **mathn** library adds miscellaneous functionality to Ruby's math functions. Among other things, it modifies the **Math::sqrt** method so that if you pass in a square number, you get a **Fixnum** back instead of a **Float**.

```
require 'mathn'
p Math::sqrt(Rational(25,1))       # => 5
p Math::sqrt(25)                   # => 5
p Math::sqrt(25.0)                 # => 5.0
```

Pass in a single integer argument n to **Kernel#rand**, and it returns an integer between 0 and n-1.

The Ruby interpreter initializes its random number generator on startup, using a seed derived from the current time and the process number. To reliably generate the same random numbers over and over again, you can set the random number seed manually by calling the **Kernel#srand** function with the integer argument of your choice.

```
#Some random numbers based on process number and current time
p rand(1000)                       # => 187
p rand(1000)                       # => 551
p rand(1000)                       # => 911
```

```
#Start the seed with the number 1
srand 1
p rand(1000)                        # => 37
p rand(1000)                        # => 235
p rand(1000)                        # => 908

#Reset the seed to its previous state
srand 1
p rand(1000)                        # => 37
p rand(1000)                        # => 235
p rand(1000)                        # => 908
```

You can also convert between decimal numbers and string representations of those numbers in any base from 2 to 36. Simply pass the base into **String#to_i** or **Integer#to_s**.

```
p "1045".to_i(10)                    # => 1045
p "-1001001".to_i(2)                 # => -73
p "abc".to_i(16)                     # => 2748
p 42.to_s(10)                       # => "42"
p -100.to_s(2)                       # => "-1100100"
p 255.to_s(16)                       # => "ff"
```

Write a generator function that yields each number in the sequence.

```
def fibonacci(limit = nil)
 seed1 = 0
 seed2 = 1
 while not limit or seed2 <= limit
     yield seed2
     seed1, seed2 = seed2, seed1 + seed2
 end
end

fibonacci(20) { |x| puts x }
# 1
# 1
# 2
# 3
# 5
# 8
# 13
```

Though integer sequences are the most common, any type of number can be used in a sequence. For instance, Float#step works just like Integer#step:

```
1.5.step(2.0, 0.25) { |x| puts x }
# => 1.5
# => 1.75
# => 2.0
```

Instantiate the **Prime** class to create a prime number generator. Call **Prime#succ** to get the next prime number in the sequence.

```
require 'mathn'
primes = Prime.new
primes.each { |x| puts x; break if x > 15; }
# 5
# 7
# 11
# 13
# 17
p primes.succ            # => 19
```

The best-known prime number algorithm is the Sieve of Eratosthenes, which finds all primes in a certain range by iterating over that range multiple times. On the first pass, it eliminates every even number greater than 2, on the second pass every third number after 3, on the third pass every fifth number after 5, and so on.

```
def sieve(max=100)
 sieve = []
 (2..max).each { |i| sieve[i] = i }
 (2..Math.sqrt(max)).each do |i|
     (i*i).step(max, i) { |j| sieve[j] = nil } if sieve[i]
 end
 sieve.compact
end

p sieve(30)
# => [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Date and Time

The **Time** class contains Ruby's interface to the C libraries, and it's all you need for most applications. The **Time** class has a lot of Ruby idiom attached to it, but most of its methods have strange unRuby-like names like **strftime** and **strptime**. This is for the benefit of people who are already used to the C library, or one of its other interfaces (like Perl or Python's).

The internal representation of a **Time** object is a number of seconds before or since "time zero." Time zero for Ruby is the Unix epoch: the first second GMT of January 1, 1970. You can get the current local time with **Time.now**, or create a **Time** object from seconds-since-epoch with **Time.at**.

```
Time.now                            # => Sat Mar 18 14:49:30 EST 2006
Time.at(0)                          # => Wed Dec 31 19:00:00 EST 1969
```

```
t = Time.at(0)
p t.sec                             # => 0
p t.min                             # => 0
p t.hour                            # => 19
p t.day                             # => 31
p t.month                           # => 12
p t.year                            # => 1969
p t.wday                            # => 3
```

Apart from the awkward method and member names, the biggest shortcoming of the **Time** class is that on a 32-bit system, its underlying implementation can't handle dates before December 1901 or after January 2037.

```
Time.local(1865, 4, 9)  # ArgumentError: time out of range
Time.local(2100, 1, 1)  # ArgumentError: time out of range
```

To represent those times, you'll need to turn to Ruby's other time implementation: the **Date** and **DateTime** classes. You can probably use **DateTime** for everything, and not use **Date** at all.

```
require 'date'
p DateTime.new.to_s                 # => "-4712-01-01T00:00:00Z"
p DateTime::now.to_s                # => "2006-03-18T14:53:18-0500"
```

Clearly DateTime is superior to Time for astronomical and historical applications, but you can use Time for most everyday programs. This table should give you a picture of the relative advantages of Time objects and DateTime objects.

	Time	DateTime
Date range	19012037 on 32-bit systems	Effectively infinite
Handles Daylight Saving Time	Yes	No
Handles calendar reform	No	Yes
Time zone conversion	Easy with the tz gem	Difficult unless you only work with time zone offsets
Common time formats like RFC822	Built-in	Write them yourself
Speed	Faster	Slower

```
puts Date.new(1582, 10, 4)
# => "1582-10-04"
puts Date.parse('2/9/2007')
# => "2007-02-09"
puts DateTime.parse('02-09-2007 12:30:44 AM')
# => "2007-09-02T00:30:44Z"
```

```
american_date = '%m/%d/%y'
puts Date.strptime('2/9/07', american_date)           # => "2007-02-09"
four_digit_year_date = '%m/%d/%Y'
puts Date.strptime('2/9/2007', four_digit_year_date)  # => "2007-02-09"
date_and_time = '%m-%d-%Y %H:%M:%S %Z'
puts DateTime.strptime('02-09-2007 12:30:44 EST', date_and_time)
# => "2007-02-09T12:30:44-0500"
```

```
(Date.new(1776, 7, 2)..Date.new(1776, 7, 4)).each { |x| puts x }
# 1776-07-02
# 1776-07-03
# 1776-07-04
```

```
the_first = Date.new(2004, 1, 1)
the_fifth = Date.new(2004, 1, 5)
```

```
the_first.upto(the_fifth) { |x| puts x }
# 2004-01-01
# 2004-01-02
# 2004-01-03
# 2004-01-04
# 2004-01-05
```

```
the_first.step(the_fifth, 2) { |x| puts x }
# 2004-01-01
# 2004-01-03
# 2004-01-05

puts the_fifth - the_first  #4
```

Use the built-in **timeout** library. The Timeout.timeout method takes a code block and a deadline (in seconds). If the code block finishes running in time, it returns true. If the deadline passes and the code block is still running, Timeout.timeout terminates the code block and raises an exception.

```
before = Time.now
begin
  status = Timeout.timeout(5) { sleep }
rescue Timeout::Error
  puts "I only slept for #{Time.now-before} seconds."
end
```

Array

If your array contains only strings, you may find it simpler to build your array by enclosing the strings in the **w{}** syntax, separated by whitespace.

If you want to produce a new array based on a transformation of some other array, use Enumerable#collect along with a block that takes one element and transforms it:

```
[1, 2, 3, 4].collect { |x| x ** 2 }          # => [1, 4, 9, 16]
```

Ruby supports **for** loops and the other iteration constructs found in most modern programming languages, but its prefered idiom is a code block fed to an method like **each** or **collect**.

Methods like **each** and **collect** are called generators or iterators: they iterate over a data structure, **yield**ing one element at a time to whatever code block you've attached. Once your code block completes, they continue the iteration and **yield** the next item in the data structure (according to whatever definition of "next" the generator supports).

In a method like **each**, the return value of the code block, if any, is ignored. Methods like **collect** take a more active role. After they **yield** an element of a data structure to a code block, they use the return value in some way. The **collect** method uses the return value of its attached block as an element in a new array.

If you need to have the array indexes along with the array elements, use Enumerable#each_with_index.

```
['a', 'b', 'c'].each_with_index do |item, index|
  puts "At position #{index}: #{item}"
end
# At position 0: a
# At position 1: b
# At position 2: c
```

Ruby's Array class also defines several generators not seen in Enumerable . For instance, to iterate over a list in reverse order, use the reverse_each method.

```
[1, 2, 3, 4].reverse_each { |x| puts x }
# 4
# 3
# 2
# 1
```

Enumerable#collect has a destructive equivalent: **Array# collect**!, also known

as **Arary#map**! (a helpful alias for Python programmers). This method acts just like **collect**, but instead of creating a new array to hold the return values of its calls to the code block, it replaces each item in the old array with the corresponding value from the code block. This saves memory and time, but it destroys the old array.

```
array = ['a', 'b', 'c']
array.collect! { |x| x.upcase }
p array                          # => ["A", "B", "C"]
array.map! { |x| x.downcase }
p array                          # => ["a", "b", "c"]
```

If you need to skip certain elements of an array, you can use the iterator methods Range#step and Integer#upto instead of Array#each. These methods generate a sequence of numbers that you can use as successive indexes into an array.

```
array = ['junk', 'junk', 'junk', 'val1', 'val2']
3.upto(array.length-1) { |i| puts "Value #{array[i]}" }
# Value val1
# Value val2

array = ['1', 'a', '2', 'b', '3', 'c']
(0..array.length-1).step(2) do |i|
 puts "Letter #{array[i]} is #{array[i+1]}"
end
# Letter 1 is a
# Letter 2 is b
# Letter 3 is c
```

You can even use the splat operator to extract items from the front of the array:

```
a, b, *c = [12, 14, 178, 89, 90]
a                              # => 12
b                              # => 14
c                              # => [178, 89, 90]
```

One final nugget of code that is interesting enough to mention even though it has no legitimate use in Ruby: it doesn't save enough memory to be useful, and it's slower than doing a swap with an assignment. It's possible to swap two integer variables using bitwise XOR, without using any additional storage space at all.

```
a, b = rand(1000), rand(1000)       # => [595, 742]
a = a ^ b                        # => 181
b = b ^ a                        # => 595
a = a ^ b                        # => 742
```

Use Array#uniq to create a new array, based on an existing array but with no duplicate elements. Array#uniq! strips duplicate elements from an existing array.

```
survey_results = [1, 2, 7, 1, 1, 5, 2, 5, 1]
distinct_answers = survey_results.uniq       # => [1, 2, 7, 5]
p distinct_answers
p survey_results.uniq!
```

To ensure that duplicate values never get into your list, use a Set instead of an array. If you try to add a duplicate element to a Set, nothing will happen.

```
require 'set'
survey_results = [1, 2, 7, 1, 1, 5, 2, 5, 1]
p survey_results.to_set
# => #<Set: {5, 1, 7, 2}>
```

Array#uniq preserves the original order of the array (that is, the first instance of an object remains in its original location), but a **Set** has no order, because its internal implementation is a hash.

Needing to strip all instances of a particular value from an array is a problem that often comes up. Ruby provides **Array#delete** for this task, and **Array#compact** for the special case of removing **nil** values.

```
a = [1, 2, nil, 3, 3, nil, nil, nil, 5]
p a.compact                # => [1, 2, 3, 3, 5]
a.delete(3)
p a                       # => [1, 2, nil, nil, nil, nil, 5]
```

To sort objects based on one of their data members, or by the results of a method call, use Array#sort_by. This code sorts an array of arrays by size, regardless of their contents:

```
arrays = [[1,2,3], [100], [10,20]]
p arrays.sort_by { |x| x.size }      # => [[100], [10, 20], [1, 2, 3]]
```

If there is one "canonical" way to sort a particular class of object, then you can have that class implement the <=> comparison operator. This is how Ruby automatically knows how to sort numbers in ascending order and strings in ascending ASCII order: Numeric and String both implement the comparison operator.

The **sort_by** method sorts an array using a Schwartzian transform. This is the most useful customized sort, because it's fast and easy to define.

If you pass a block into sort, Ruby calls the block to make comparisons instead of using the comparison operator. This is the most general possible sort,

and it's useful for cases where **sort_by** won't work.

```
result = [1, 100, 42, 23, 26, 10000].sort do |x, y|
  x == 42 ? 1 : x <=> y
end
p result
# => [1, 23, 26, 100, 10000, 42]
```

You want an alphabetical sort, regardless of case. Use Array#sort_by. This is both the fastest and the shortest solution.

```
list = ["Albania", "anteater", "zorilla", "Zaire"]
p list.sort_by { |x| x.downcase }
# => ["Albania", "anteater", "Zaire", "zorilla"]
```

The comparison operator and a **sort** code block both take one argument: an object against which to compare **self**. A call to **<=>** (or a **sort** code block) should return1 if **self** is "less than" the given object (and should therefore show up before it in a sorted list). It should return 1 if **self** is "greater than" the given object (and should show up after it in a sorted list), and 0 if the objects are "**equal**" (and it doesn't matter which one shows up first). You can usually avoid remembering this by delegating the return value to some other object's **<=>** implementation.

Although inject is the preferred way of summing over a collection, inject is generally a few times slower than each. The speed difference does not grow exponentially, so you don't need to always be worrying about it as you write code. But after the fact, it's a good idea to look for inject calls in crucial spots that you can change to use faster iteration methods like each.

```
collection = [1, 2, 3, 4, 5]
p    collection.inject(1) {|total, i| total * i}  # => 120
```

The simplest way to shuffle an array (in Ruby 1.8 and above) is to sort it randomly:

```
[1, 2, 3].sort_by { rand } # => [1, 3, 2]
```

To sort only the smallest elements, you can keep a sorted "stable" of champions, and kick the largest champion out of the stable whenever you find an element that's smaller. If you encounter a number that's too large to enter the stable, you can ignore it from that point on. This process rapidly cuts down on the number of elements you must consider, making this approach faster than doing a sort.

Array objects have overloaded arithmetic and logical operators to provide the three simplest set operations:

```
#Union
[1,2,3] | [1,4,5]                 # => [1, 2, 3, 4, 5]


#Intersection
[1,2,3] & [1,4,5]                 # => [1]

#Difference
[1,2,3] - [1,4,5]                 # => [2, 3]
```

Set objects overload the same operators, as well as the exclusive-or operator (**^**).

```
require 'set'
a = [1, 2, 3]
b = [3, 4, 5]
p a.to_set ^ b.to_set
# => #<Set: {5, 1, 2, 4}>
```

You want to partition a **Set** or array based on some attribute of its elements. All elements that go "together" in some code-specific sense should be grouped together in distinct data structures.

Use Set#divide, passing in a code block that returns the partition of the object it's passed. The result will be a new Set containing a number of partitioned subsets of your original Set.

```
require 'set'
s = Set.new((1..10).collect)
# => #<Set: {5, 6, 1, 7, 2, 8, 3, 9, 4, 10}>

p s.divide { |x| x < 5 }
# => #<Set: {#<Set: {5, 6, 7, 8, 9, 10}>, #<Set: {1, 2, 3, 4}>}>

p s.divide { |x| x % 2 }
# => #<Set: {#<Set: {6, 2, 8, 4, 10}>, #<Set: {5, 1, 7, 3, 9}>}>

s = Set.new([1, 2, 3, 'a', 'b', 'c', -1.0, -2.0, -3.0])
p s.divide { |x| x.class }
# => #<Set: {#<Set: {"a", "b", "c"}>,
# =>      #<Set: {1, 2, 3}>,
# =>      #<Set: {-1.0, -3.0, -2.0}>}>
```

Hash

You can create a Hash by calling Hash.new or by using one of the special sytaxes Hash[] or {}. With the Hash[] syntax, you pass in the initial elements as comma-separated object references. With the {} syntax, you pass in the initial contents as comma-separated key-value pairs.

```
empty = Hash.new                          # => {}
empty = {}                               # => {}
numbers = { 'two' => 2, 'eight' => 8} # => {"two"=>2, "eight"=>8}
numbers = Hash['two', 2, 'eight', 8]       # => {"two"=>2, "eight"=>8}
```

You can get an array containing the keys or values of a hash with **Hash#keys** or **Hash#values**. You can get the entire hash as an array with **Hash#to_a.**

```
numbers = { 'two' => 2, 'eight' => 8, 'ten' => 10}
p numbers.keys                # => ["two", "eight", "ten"]
p numbers.values              # => [2, 8, 10]
p numbers.to_a                # => [["two", 2], ["eight", 8], ["ten", 10]]
```

The main advantage of a hash is that it's often easier to find what you're looking for. Checking whether an array contains a certain value might require scanning the entire array. To see whether a hash contains a value for a certain key, you only need to look up that key. The **set** library (as seen in the previous chapter) exploits this behavior to implement a class that looks like an array, but has the performance characteristics of a hash.

A hash in Ruby is actually implemented as an array. When you look up a key in a hash (either to see what's associated with that key, or to associate a value with the key), Ruby calculates the hash code of the key by calling its **hash** method. The result is used as a numeric index in the array.

Except for strings and other built-in objects, most objects have a hash code equivalent to their internal object ID. As seen above, you can override **Object#hash** to change this, but the only time you should need to do this is if your class also overrides **Object#==**. If two objects are considered equal, they should also have the same hash code; otherwise, they will behave strangely when you put them into hashes.

Whenever you would otherwise use a quoted string, use a symbol instead. A symbol can be created by either using a colon in front of a word, like **:keyname**, or by transforming a string to a symbol using **String#intern**.

```
people = Hash.new
people[:nickname] = 'Matz'
people[:language] = 'Japanese'
```

```
people['last name'.intern] = 'Matsumoto'
p people[:nickname]                                  # => "Matz"
p people['nickname'.intern]                           # => "Matz"
p people['last name'.intern]                          # => "Matsumoto"
```

While '**name**' and '**name**' appear exactly identical, they're actually different. Each time you create a quoted string in Ruby, you create a unique object.

```
p 'name'.object_id                                   # => -605973716
p 'name'.object_id                                   # => -605976356
p 'name'.object_id                                   # => -605978996
```

By comparison, each instance of a symbol refers to a single object.

```
p :name.object_id                                   # => 878862
p :name.object_id                                   # => 878862
p 'name'.intern.object_id                            # => 878862
p 'name'.intern.object_id                            # => 878862
```

Using symbols instead of strings saves memory and time. It saves memory because there's only one symbol instance, instead of many string instances. If you have many hashes that contain the same keys, the memory savings adds up.

Using symbols as hash keys is faster because the hash value of a symbol is simply its object ID. If you use strings in a hash, Ruby must calculate the hash value of a string each time it's used as a hash key.

A normal hash has a default value of nil:

```
h = Hash.new
h[1]                                      # => nil
h['do you have this string?']                 # => nil
```

There are two ways of creating default values for hashes. If you want the default value to be the same object for every hash key, pass that value into the Hash constructor.

```
h = Hash.new("nope")
h[1]                                     # => "nope"
h['do you have this string?']            # => "nope"
```

If you want the default value for a missing key to depend on the key or the current state of the hash, pass a code block into the hash constructor. The block will be called each time someone requests a missing key.

```
h = Hash.new { |hash, key| (key.respond_to? :to_str) ? "nope" : nil }
h[1]                                      # => nil
h['do you have this string']              # => "nope"
```

When you use a string as a hash key, the string is transparently copied and the copy is frozen. This is to avoid confusion should you modify the string in place, then try to use its original form to do a hash lookup.

```
key = "Modify me if you can"
h = { key => 1 }
key.upcase!                    # => "MODIFY ME IF YOU CAN"
p h[key]                        # => nil
p h["Modify me if you can"]         # => 1

p h.keys                        # => ["Modify me if you can"]
h.keys[0].upcase!
# TypeError: can't modify frozen string
```

To add an array of key-value pairs to a hash, either iterate over the array with **Array#each**, or pass the hash into **Array#inject**. Using **inject** is slower but the code is more concise.

```
squares = [[1, 1], [2, 4], [3, 9]]
results = {}
squares.each { |k, v| results[k] = v }
p results
# => {1=>1, 2=>4, 3=>9}

p squares.inject({}) { |h, kv| h[kv[0]] = kv[1]; h }
# => {1=>1, 2=>4, 3=>9}
```

To insert into a hash every key-value from another hash, use **Hash#merge!**. If a key is present in both hashes when **a.merge!(b)** is called, the value in **b** takes precedence over the value in **a**. **Hash#merge!** also has a nondestructive version, **Hash#merge**, which creates a new **Hash** with elements from both parent hashes. Again, the hash passed in as an argument takes precedence.

```
squares = { 1 => 1, 2 => 4, 3 => 9}
cubes = { 3 => 27, 4 => 256, 5 => 3125}
squares.merge!(cubes)
p squares                   # =>{5=>3125, 1=>1, 2=>4, 3=>27, 4=>256}
p cubes                    # =>{5=>3125, 3=>27, 4=>256}
```

To completely replace the entire contents of one hash with the contents of another, use Hash#replace.

```
squares = { 1 => 1, 2 => 4, 3 => 9}
cubes = { 1 => 1, 2 => 8, 3 => 27}
p squares.replace(cubes)
p squares                   # => {1=>1, 2=>8, 3=>27}
```

Most of the time you want to remove a specific element of a hash. To do that,

pass the key into **Hash#delete**.

```
h = {}
h[1] = 10
p h                                    # => {1=>10}
h.delete(1)
p h                                    # => {}
```

Don't try to delete an element from a hash by mapping it to **nil**. It's true that, by default, you get **nil** when you look up a key that's not in the hash, but there's a difference between a key that's missing from the hash and a key that's present but mapped to **nil.**

The easiest solution is to call the **Hash#rehash** method every time you modify one of the hash's keys. **Hash#rehash** will repair the broken treasure map defined above.

Ruby performs hash lookups using not the key object itself but the object's hash code (an integer obtained from the key by calling its **hash** method). The default implementation of **hash**, in **Object**, uses an object's internal ID as its hash code. **Array, Hash**, and **String** override this method to provide different behavior.

Since an object's internal ID never changes, the Object implementation is what you want to get reliable hashing. To get it back, you'll have to override or subclass the hash method of Array or Hash (depending on what type of key you're having trouble with).

Another solution is to freeze your hash keys. Any frozen object can be reliably used as a hash key, since you can't do anything to a frozen object that would cause its hash code to change. Ruby uses this solution: when you use a string as a hash key, Ruby copies the string, freezes the copy, and uses that as the actual hash key.

Most likely, the iterator you want is **Hash#each_pair** or **Hash#each**. These methods yield every key-value pair in the hash.

```
hash = { 1 => 'one', [1,2] => 'two', 'three' => 'three' }
hash.each_pair { |key, value| puts "#{key.inspect} maps to #{value}"}
```

Use **Hash#each_key** if you only need the keys of a hash.

Use **Hash#each_value** if you only need the values of a hash.

Don't iterate over Hash#each_value looking for a particular value: it's simpler and faster to use has_value? instead.

```
hash = {}
1.upto(10) { |x| hash[x] = x * x }
p hash.has_value? 49                # => true
```

```
p hash.has_value? 81                # => true
p hash.has_value? 50                # => false
```

Don't modify the keyset of a hash during an iteration, or you'll get undefined results and possibly a **RuntimeError.**

An alternative to using the hash iterators is to get an array of the keys, values, or key-value pairs in the hash, and then work on the array. You can do this with the **keys, values**, and **to_a** methods, respectively.

```
hash = {1 => 2, 2 => 2, 3 => 10}
p hash.keys                         # => [1, 2, 3]
p hash.values                       # => [2, 2, 10]
p hash.to_a                         # => [[1, 2], [2, 2], [3, 10]]
```

Hash#sort and **Hash#sort_by** turn a hash into an array of two-element subarrays (one for each key-value pair), then sort the array of arrays however you like.

```
to_do = {
  'Clean car' => 5,
  'Take kangaroo to vet' => 3,
  'Realign plasma conduit' => 3
}
to_do.sort_by { |task, priority| [priority, task] }.each { |k,v| puts k }
```

The easiest way to print a hash is to use **Kernel#p. Kernel#p** prints out the "inspected" version of its arguments: the string you get by calling **inspect** on the hash. The "inspected" version of an object often looks like Ruby source code for creating the object, so it's usually readable.

There are a number of ways of printing hash contents. The solution you choose depends on the complexity of the hash you're trying to print, where you're trying to print the hash, and your personal preferences. The best general-purpose solution is the **pp** library.

You can also print hashes by converting them into YAML with the **yaml** library. YAML is a human-readable markup language for describing data structures.

```
h = {}
h[:name] = "Robert"
h[:nickname] = "Bob"
h[:age] = 43
h[:email_addresses] = {
:home => "bob@example.com",
:work => "robert@example.com"
}
p h
```

```
require 'yaml'
puts h.to_yaml
```

You can use the **Hash#select** method to extract part of a hash that follows a certain rule. The array returned by Hash#select contains a number of key-value pairs as two-element arrays. The first element of one of these inner arrays is a key into the hash, and the second element is the corresponding value. This is similar to how Hash#each yields a succession of two-element arrays.

```
to_do = {
 'Clean car' => 5,
 'Take kangaroo to vet' => 3,
 'Realign plasma conduit' => 3
}
p to_do.select {|k, v| v < 4 }
# [["Take kangaroo to vet", 3], ["Realign plasma conduit", 3]]
```

If you want another hash instead of an array of key-value pairs, you can use **Hash#inject** instead of **Hash#select**.

```
res = to_do.inject({}) do |h, kv|
 k, v = kv
 h[k] = v if v < 4
 h
end
p res       # {"Take kangaroo to vet"=>3, "Realign plasma conduit"=>3}
```

File and Directories

Kernel#open is the simplest way to open a file. It returns a **Filel** object that you can read from or write to, depending on the "mode" constant you pass in.

To write data to a file, pass a mode of '**w**' to **open**. You can then write lines to the file with **File#puts**, just like printing to standard output with **Kernel#puts**.

```
open('beans.txt', "w") do |file|
  file.puts('lima beans')
  file.puts('pinto beans')
  file.puts('human beans')
end
```

To read data from a file, open it for read access by specifying a mode of 'r', or just omitting the mode. You can slurp the entire contents into a string with **File#read**, or process the file line-by-line with **File#each**.

```
open('beans.txt') do |file|
  file.each { |l| puts "A line from the file: #{l}" }
end
# A line from the file: lima beans
# A line from the file: pinto beans
# A line from the file: human beans
```

The **open** method creates a new **File** object, passes it to your code block, and closes the file automatically after your code block runs -- even if your code throws an exception. This saves you from having to remember to close the file after you're done with it. You could rely on the Ruby interpreter's garbage collection to close the file once it's no longer being used, but Ruby makes it easy to do things the right way.

Although this chapter focuses mainly on disk files, most of the methods of **File** are actually methods of its superclass, IO. You'll encounter many other classes that are also subclasses of **IO**, or just respond to the same methods. This means that most of the tricks described in this chapter are applicable to classes like the **Socket** class for Internet sockets and the infinitely useful **StringIO.**

```
File.file? filename                # => true
File.file? directory_name          # => false
File.exists? directory_name        # => true
File.directory? directory_name     # => true
File.directory? filename           # => false

File.blockdev? '/dev/hda1'              # => true
File.chardev? '/dev/tty1'               # => true
File.socket? '/var/run/mysqld/mysqld.sock' # => true
```

```
system('mkfifo named_pipe')
File.pipe? 'named_pipe'                    # => true

File.readable?('/bin/ls')       # => true
File.writable?('/bin/ls')       # => false
File.executable?('/bin/ls')     # => true

File.owned? 'test_file'      # => true
File.grpowned? 'test_file'    # => true
File.owned? '/bin/ls'        # => false
```

You can manipulate the constants defined above to get a new mode, then pass it in along with the filename to **File.chmod**.

The simplest way to do this is to use **File.lstat#mode** to get the file's current permission bitmap, then modify it with bit operators to add or remove permissions. You can pass the result into **File.chmod**.

If you're starting from a directory name, you can use **Dir.entries** to get an array of the items in the directory, or **Dir.foreach** to iterate over the items.

```
p Dir.entries('test')
Dir.foreach('test') { |x| puts x if x != "." && x != ".."}
```

You can also use **Dir[]** to pick up all files matching a certain pattern, using a format similar to the bash shell's glob format.

```
p Dir["test/**/*.java"]
p Dir["test/*.java"]
```

You can also open a directory handle with **Dir#open**, and treat it like any other Enumerable. Methods like **each,each_with_index, grep**, and **reject** will all work (but see below if you want to call them more than once). As with **File#open**, you should do your directory processing in a code block so that the directory handle will get closed once you're done with it.

Reading entries from a **Dir** object is more like reading data from a file than iterating over an array. If you call one of the **Dir** instance methods and then want to call another one on the same **Dir** object, you'll need to call **Dir#rewind** first to go back to the beginning of the directory listing.

```
d = Dir.open('test')
p d.reject { |f| f[0].chr == '.' }

#Now the Dir object is useless until we call Dir#rewind.
p d.entries.size        # => 0
d.rewind
p d.entries.size        # => 9
```

```
d.close
```

Methods for listing directories and looking for files return string pathnames instead of **File** and **Dir** objects. This is partly for efficiency, and partly because creating a **File** or **Dir** actually opens up a filehandle on that file or directory.

Globs make excellent shortcuts for finding files in a directory or a directory tree. Especially useful is the ** glob, which matches any number of directories. A glob is the easiest and fastest way to recursively process every file in a directory tree, although it loads all the filenames into an array in memory. For a less memoryintensive solution, see the **find** library.

```
p Dir["test/**/*"]
p Dir["test/{output, data}*"]
```

Open the file with **Kernel#open**, and pass in a code block that does the actual reading. To read the entire file into a single string, use **IO#read.**

```
# Put some stuff into a file.
open('sample_file', 'w') do |f|
  f.write("This is line one.\nThis is line two.")
end

# Then read it back out.
p open('sample_file') { |f| f.read }
# => "This is line one.\nThis is line two."
```

To read the file as an array of lines, use IO#readlines:

```
p open('sample_file') { |f| f.readlines }
# => ["This is line one.\n", "This is line two."]
```

To iterate over each line in the file, use IO#each. This technique loads only one line into memory at a time:

```
open('sample_file').each { |x| p x }
# "This is line one.\n"
# "This is line two."
```

If a certain string always marks the end of a chunk, you can pass that string into **IO#each** to get one chunk at a time, as a series of strings. This lets you process each full chunk as a string, and it uses less memory than reading the entire file.

```
# Create a file…
open('end_separated_records', 'w') do |f|
 f << %{This is record one.\nIt spans multiple lines.ENDThis is record two.END}
end
```

```
# And read it back in.
open('end_separated_records') { |f| f.each('END') { |record| p record } }
# "This is record one.\nIt spans multiple lines.END"
# "This is record two.END"
```

You can also pass a delimiter string into **IO#readlines** to get the entire file split into an array by the delimiter string. By default, **IO#each** and **IO#readlines** split the file by line.

```
# Create a file…
open('pipe_separated_records', 'w') do |f|
 f << "This is record one.|This is record two.|This is record three."
end

# And read it back in.
p open('pipe_separated_records') { |f| f.readlines('|') }
# => ["This is record one.|", "This is record two.|", "This is record three."]
```

IO#each and IO#readlines don't strip the delimiter strings from the end of the lines. Assuming the delimiter strings aren't useful to you, you'll have to strip them manually.

To strip delimiter characters from the end of a line, use the String#chomp or String#chomp! methods. By default, these methods will remove the last character or set of characters that can be construed as a newline.

```
p "This line has a Unix/Mac OS X newline.\n".chomp
# => "This line has a Unix/Mac OS X newline."

p "This line has a Windows newline.\r\n".chomp
# => "This line has a Windows newline."

p "This line has an old-style Macintosh newline.\r".chomp
# => "This line has an old-style Macintosh newline."

p "This string contains two newlines.\n\n".chomp
# "This string contains two newlines.\n"

p 'This is record two.END'.chomp('END')
# => "This is record two."

p 'This string contains no newline.'.chomp
# => "This string contains no newline."
```

You can **chomp** the delimiters as **IO#each** yields each record, or you can **chomp** each line returned by **IO#readlines.**

```
open('pipe_separated_records') do |f|
 f.each('|') { |l| puts l.chomp('|') }
end
# This is record one.
# This is record two.
# This is record three.

lines = open('pipe_separated_records') { |f| f.readlines('|') }
p lines
# => ["This is record one.|", "This is record two.|",
#    "This is record three."]
lines.each { |l| l.chomp!('|') }
p lines
# => ["This is record one.", "This is record two.", "This is record three."]
```

Use **IO#read** to read a certain number of bytes, or **IO#each_byte** to iterate over the **File** one byte at a time. The following code uses **IO#read** to continuously read uniformly sized chunks until it reaches end-of-file:

```
open("pipe_separated_records") do |f|
 f.each_chunk(15) { |chunk| puts chunk }
end
# This is record
# one.|This is re
# cord two.|This
# is record three
# .

open("pipe_separated_records") do |f|
 f.each_byte { |byte| puts byte.chr }
end
# T
# h
# i
# s
#
# i
# s
# ...
```

Open the file in write mode ('w'). The file will be created if it doesn't exist, and truncated to zero bytes if it does exist. You can then use IO#write or the << operator to write strings to the file, as though the file itself were a string and you were appending to it.

You can also use IO#puts or IO#p to write lines to the file, the same way

you can use Kernel#puts or Kernel#p to write lines to standard output.

```
open('output1', 'w') { |f| f << "This file contains great truths.\n" }
open('output1', 'w') do |f|
 f.puts 'The great truths have been overwritten with an advertisement.'
end

p open('output1') { |f| f.read }
# => "The great truths have been overwritten with an advertisement.\n"
```

To append to a file without overwriting its old contents, open the file in append mode ('a') instead of write mode:

```
open('output', "a") { |f| f.puts 'Buy Ruby(TM) brand soy sauce!' }
open('output') { |f| puts f.read }
# The great truths have been overwritten with an advertisement.
# Buy Ruby(TM) brand soy sauce!
```

There's no guarantee that data will be written to your file as soon as you call << or puts. Since disk writes are expensive, Ruby lets changes to a file pile up in a buffer. It occasionally flushes the buffer, sending the data to the operating system so it can be ritten to disk.

You can manually flush Ruby's buffer for a particular file by calling its **IO#flush** method. You can turn off Ruby's buffering altogether by setting **IO.sync** to false.

```
open('output2', 'w') do |f|
  f << 'This is going into the Ruby buffer.'
  f.flush # Now it's going into the OS buffer.
end
```

If two files differ, it's likely that their sizes also differ, so you can often solve the problem quickly by comparing sizes. If both files are regular files with the same size, you'll need to look at their contents.

This code does the cheap checks first:

1. If one file exists and the other does not, they're not the same.
2. If neither file exists, say they're the same.
3. If the files are the same file, they're the same.
4. If the files are of different types or sizes, they're not the same.
5. Otherwise, it compares the files contents, a block at a time.

```
def File.same_contents?(p1, p2)
 return false if File.exists?(p1) != File.exists?(p2)
 return true if !File.exists(p1)
 return true if File.expand_path(p1) == File.expand_path(p2)
 return false if File.ftype(p1) != File.ftype(p2) || File.size(p1) !=
File.size(p2)
 open(p1) do |f1|
```

```
        open(p2) do |f2|
            blocksize = f1.lstat.blksize
            same = true
            while same && !f1.eof? && !f2.eof?
                same = f1.read(blocksize) == f2.read(blocksize)
            end
            return same
        end
    end
    end
```

The simplest solution is to load all the files and directories into memory with a big recursive file glob, and iterate over the resulting array. This uses a lot of memory because all the filenames are loaded into memory at once. A more elegant solution is to use the find method in the Find module. It performs a depth-first traversal of a directory tree, and calls the given code block on each directory and file. The code block should take as an argument the full path to a directory or file.

```
require 'find'

Find.find('test') { |path| puts path }
#    test
#    test/_desktop.ini
#    test/Test.java
#    test/Test.class
#    test/subtest
#    test/subtest/_desktop.ini
#    test/subtest/Test.java
#    test/subtest/output.txt
#    test/subtest/Desktop_.ini
#    test/subtest/data.txt
#    test/output.txt
#    test/Desktop_.ini
#    test/data.txt
```

When your block is passed a path to a directory, you can prevent **Find.find** from recursing into a directory by calling **Find.prune**.

```
Find.find('test') do |path|
 Find.prune if File.basename(path) == 'data.txt'
 puts path
end
```

Find.find works by keeping a queue of files to process. When it finds a directory, it inserts that directory's files at the beginning of the queue. This gives it the characteristics of a depth-first traversal.

If you want to do a breadth-first traversal instead of a depth-first one, the simplest solution is to use a glob and sort the resulting array. Pathnames sort naturally in a way that simulates a breadth-first traversal:

```
Dir["test/**"].sort.each { |x| puts x }
```

Use **String#succ** to generate versioned suffixes for a filename until you find one that doesn't already exist:

```
def File.versioned_file(base, first_suffix='.0')
 suffix = nil
 filename = base
 while File.exists?(filename)
     suffix = (suffix ? suffix.succ : first_suffix)
     filename = base + suffix
 end
 return filename
end
5.times do |i|
 name = File.versioned_file('filename.txt')
 open(name, 'w') { |f| f << "Contents for run #{i}" }
 puts "Created #{name}"
end
# Created filename.txt
# Created filename.txt.0
# Created filename.txt.1
# Created filename.txt.2
# Created filename.txt.3
```

If you want to copy or move the original file to the versioned filename as a prelude to writing to the original file, include the **ftools** library to add the class methods **File**. **copy** and **File.move**.

```
def File.backup(filename, move=false)
    new_filename = nil
    if File.exists? filename
        new_filename = File.versioned_file(filename)
        File.send(move ? :move : :copy, filename, new_filename)
    end
    return new_filename
end
```

The StringIO class wraps a string in the interface of the IO class. You can treat it like a file, then get everything that's been "written" to it by calling its

string method.

```
require 'stringio'
s = StringIO.new %{I am the very model of a modern major general.
I've information vegetable, animal, and mineral.}

p s.pos                          # => 0
s.each_line { |x| puts x }
# I am the very model of a modern major general.
# I've information vegetable, animal, and mineral.
p s.eof?                         # => true
p s.pos                          # => 95
s.rewind
p s.pos                          # => 0
p s.grep /general/
# => ["I am the very model of a modern major general.\n"]

s = StringIO.new
s.write('Treat it like a file.')
s.rewind
s.write("Act like it's")
p s.string                         # => "Act like it's a file."

require 'yaml'
s = StringIO.new
YAML.dump(['A list of', 3, :items], s)
puts s.string
# ---
# - A list of
# - 3
# - :items
```

The Adapter is a common design pattern: to make an object acceptable as input to a method, it's wrapped in another object that presents the appropriate interface. The StringIO class is an Adapter between String and File (or IO), designed for use with methods that work on File or IO instances. With a StringIO, you can disguise a string as a file and use those methods without them ever knowing they haven't really been given a file.

For instance, if you want to write unit tests for a library that reads from a file, the simplest way is to pass in predefined StringIO objects that simulate files with various contents. If you need to modify the output of a method that writes to a file, a StringIO can capture the output, making it easy to modify and send on to its final destination.

StringIO-type functionality is less necessary in Ruby than in languages like Python, because in Ruby, strings and files implement a lot of the same methods

to begin with. Often you can get away with simply using these common methods.

Similarly, **File** and **String** both include the **Enumerable** mixin, so in a lot of cases you can read from an object without caring what type it is. This is a good example of Ruby's duck typing.

Just remember that, unlike a string, you can't iterate over a file multiple times without calling rewind.

StringIO object is always open for both reading and writing:

```
s = StringIO.new
s << "A string"
p s.read                # => ""
s << ", and more."
s.rewind
p s.read                # => "A string, and more."
```

Memory access is faster than disk access, but for large amounts of data (more than about 10 kilobytes), StringIO objects are slower than disk files. If speed is your aim, your best bet is to write to and read from temp files using the tempfile module. Or you can do what the open-uri library does: start off by writing to a StringIO and, if it gets too big, switch to using a temp file.

Create a Tempfile object. It has all the methods of a File object, and it will be in a location on disk guaranteed to be unique. A Tempfile object is opened for read-write access (mode w+), so you can write to it and then read from it without having to close and reopen it:

```
require 'tempfile'
out = Tempfile.new("tempfile")
p out.path  # => "C:/DOCUME~1/Andy/LOCALS~1/Temp/tempfile65956.0"
out << "Some text."
out.rewind
p out.read # => "Some text."
out.close
```

No matter where you create your temporary files, when your process exits, all of its temporary files are automatically destroyed. If you want the data you wrote to temporary files to live longer than your process, you should copy or move the temporary files to "real" files.

You can assign any **IO** object (a **File**, a **Socket**, or what have you) to the global variables $**stdin**, $**stdout**, or **$stderr**. You can then read from or write to those objects as though they were the originals.

```
new_stdout = StringIO.new
old_stdout = $stdout
```

```
puts "Hello, hello."
puts "I'm writing to standard output."
$stderr.puts "#{new_stdout.size} bytes written to standard ouput so far."
$stderr.puts "You haven't seen anything on the screen yet, but you soon will:"
$stderr.puts new_stdout.string
$stdout = STDOUT
puts "I'm writing to standard output again."
```

If you have any Unix experience, you know that when you run a Ruby script from the command line, you can make the shell redirect its standard input, output, and error streams to files or other programs. This technique lets you do the same thing from within a Ruby script.

You can use this as a quick and dirty way to write errors to a file, write output to a StringIO object (as seen above), or even read input from a socket. Within a script, you can programatically decide where to send your output, or receive standard input from multiple sources. These things are generally not possible from the command line without a lot of fancy shell scripting.

The redirection technique is especially useful when you've written or inherited a script that prints text to standard output, and you need to make it capable of printing to any file-like object. Rather than changing almost every line of your code, you can just set $stdout at the start of your program, and let it run as is. This isn't a perfect solution, but it's often good enough.

The original input and output streams for a process are always available as the constants STDIN, STDOUT, and STDERR. If you want to temporarily swap one IO stream for another, change back to the "standard" standard output by setting $stdin = STDIN. Keep in mind that since the $std objects are global variables, even a temporary change affects all threads in your script.

Since Ruby strings make no distinction between binary and text data, processing a binary file needn't be any different than processing a text file. Just make sure you add "b" to your file mode when you open a binary file on Windows.

```
open('binary', 'wb') do |f|
  (0..100).step(10) { |b| f << b.chr }
end
open('binary', 'rb') { |f| f.each_byte { |b| puts b } }
# 0
# 10
# 20
# ...
# 90
# 100
```

Removing a file is simple, with **File.delete:**

```
require 'fileutils'
FileUtils.touch "doomed_file"
p File.exists? "doomed_file"                # => true
File.delete "doomed_file"
p File.exists? "doomed_file"                # => false
```

Removing a directory tree is also fairly simple. The most confusing thing about it is the number of different methods Ruby provides to do it. The method you want is probably FileUtils.remove_dir, which recursively deletes the contents of a directory:

```
Dir.mkdir "doomed_directory"
p File.exists? "doomed_directory"          # => true
FileUtils.remove_dir "doomed_directory"
p File.exists? "doomed_directory"          # => false
```

Ruby provides several methods for removing directories, but you really only need remove_dir. Dir.delete and FileUtils.rmdir will only work if the directory is already empty. The rm_r and rm_rf defined in FileUtils are similar to remove_dir, but if you're a Unix user you may find their names more mneumonic.

The most common type of truncation is truncating a file to zero bytes, but the File.truncate method can truncate a file to any number of bytes, not just zero. You can also use the instance method, File#truncate, to truncate a file you've opened for writing:

```
filename = 'truncate.txt'
open(filename, 'w') do |f|
  f << 'These words will remain intact after the file is truncated.'
end
p File.size(filename) # => 59
File.truncate(filename, 30)
p File.size(filename)           # => 30
p open(filename) { |f| f.read }   # => "These words will remain intact"
```

These methods don't always make a file smaller. If the file starts out smaller than the size you give, they append zero-bytes (\000) to the end of file until the file reaches the specified size.

```
open(filename, "w") { |f| f << "Brevity is the soul of wit." }
p File.size(filename)              # => 27
File.truncate(filename, 30)
p File.size(filename)              # => 30
p open(filename) { |f| f.read }
# => "Brevity is the soul of wit.\000\000\000"
```

To find the current working directory, use Dir.getwd. To change the current working directory, use Dir.chdir.

```
puts Dir.getwd      # => D:/Backup/桌面
Dir.chdir("c:")
puts Dir.getwd      # => C:
```

The current working directory of a Ruby process starts out as the directory you were in when you started the Ruby interpreter. When you refer to a file without providing an absolute pathname, Ruby assumes you want a file by that name in the current working directory. Ruby also checks the current working directory when you require a library that can't be found anywhere else.

The current working directory is a useful default. If you're writing a Ruby script that operates on a directory tree, you might start from the current working directory if the user doesn't specify one.

You can change the working directory as often as necessary, but it's more reliable to use absolute pathnames, even though this can make your code less portable. This is especially true if you're writing multithreaded code.

The current working directory is global to a process. If multiple threads are running code that changes the working directory to different values, you'll never know for sure what the working directory is at any given moment.

Code Blocks and Iteration

In Ruby, a code block (or just "block") is an object that contains some Ruby code, and the context neccesary to execute it. Code blocks are the most visually distinctive aspect of Ruby, and also one of the most confusing to newcomers from other languages. Essentially, a Ruby code block is a method that has no name.

Most other languages have something like a Ruby code block: C's function pointers, C++'s function objects, Python's lambdas and list comprehensions, Perl's anonymous functions, Java's anonymous inner classes. These features live mostly in the corners of those languages, shunned by novice programmers. Ruby can't be written without code blocks. Of the major languages, only Lisp is more block-oriented. Unlike most other languages, Ruby makes code blocks easy to create and imposes few restrictions on them.

Keep in mind that the bracket syntax has a higher precedence than the **do..end** syntax.

Usually the code blocks passed into methods are anonymous objects, created on the spot. But you can instantiate a code block as a **Proc** object by calling **lambda**.

```
log = lambda { |str| puts "[LOG] #{str}" }
log.call("A test log message.")
# [LOG] A test log message.
```

Since Proc objects are created like other objects, you can create factory methods whose return values are customized pieces of executable Ruby code. Here's a simple factory method for code blocks that do multiplication:

```
def times_n(n)
    lambda { |x| x * n }
end
times_ten = times_n(10)
p times_ten.call(5)           # => 50
```

In Ruby, there is no difference between closures and blocks. Every Ruby block is also a closure. Basically, a Ruby block carries around the context in which it was defined. A block can reference the variables that were in scope when it was defined, even if those variables later go out of scope.

```
ceiling = 50
# Which of these numbers are less than the target?
p [1, 10, 49, 50.1, 200].select { |x| x < ceiling }
# => [1, 10, 49]
```

Nonetheless, the block can access **ceiling** from within **select**, because it

carries its context around with it. That's what makes it a closure.

There are several equivalent methods that take a block and return it as an object. The most favored method is **Kernel# lambda.**

You can also initialize blocks with the Proc constructor or the method Kernel#proc. The methods Kernel#lambda, Kernel#proc, and Proc.new all do basically the same thing. These three lines of code are nearly equivalent:

```
aBlock = Proc.new { |x| puts x }
aBlock = proc { |x| puts x }
aBlock = lambda { |x| puts x }
```

A block created with **lambda** acts like a Ruby method. If you don't specify the right number of arguments, you can't call the block. If you don't specify enough arguments when you call the block created with **Proc.new**, he rest of the arguments are given **nil**. In Ruby 1.8, **Kernel#proc** acts like **Kernel#lambda**. In Ruby 1.9, **Kernel#proc** acts like **Proc.new**, as better befits its name.

You don't have to do anything special to make your method capable of taking a code block. A caller can pass a code block into any Ruby method; it's just that there's no point in doing that if the method never invokes **yield**. The yield keyword acts like a special method, a stand-in for whatever code block was passed in. When you call it, it's exactly as the code block were a Proc object and you had invoked its call method.

You can pass in arguments to yield (they'll be passed to the block) and you can do things with the value of the yield statement (this is the value of the last statement in the block).

You can write a method that takes an optional code block by calling Kernel#block_given? from within your method. That method returns true only if the caller of your method passed in a code block. If it returns false, you can raise an exception, or you can fall back to behavior that doesn't need a block and never uses the yield keyword. If your method calls yield and the caller didn't pass in a code block, Ruby will throw an exception:

```
[1, 2, 3].each
# LocalJumpError: no block given
```

Put the name of the block variable at the end of the list of your method's arguments. Prefix it with an ampersand so that Ruby knows it's a block argument, not a regular argument.

```
def repeat(n)
 n.times { yield } if block_given?
end
repeat(2) { puts "Hello." }

def repeat(n, &block)
 n.times { yield } if block
```

```
end
repeat(2) { puts "Hello." }
```

If &foo is the name of a method's last argument, it means that the method accepts an optional block named foo. If the caller chooses to pass in a block, it will be made available as a Proc object bound to the variable foo. Since it is an optional argument, foo will be nil if no block is actually passed in. This frees you from having to call Kernel#block_given? to see whether or not you got a block.

A Ruby block contains references to the variable bindings, not copies of the values. If the variable changes later, the block will have access to the new value.

```
tax_percent = 6
position = lambda do
 p "I have always supported a #{tax_percent}% tax on imported limes."
end
position.call
# => "I have always supported a 6% tax on imported limes."

tax_percent = 7.25
position.call
# => "I have always supported a 7.25% tax on imported limes."
```

You want to use a data structure as an **Enumerable**, but the object's implementation of **#each** doesn't iterate the way you want. Fortunately, there's an elegant solution with no side effects: wrap the object in an **Enumerator**. This gives you a new object that acts like the old object would if you'd swapped out its **each** method.

```
array = %w{bob loves alice}
require 'enumerator'
reversed_array = array.to_enum(:reverse_each)
p reversed_array.collect { |x| x.capitalize }
reversed_array.each_with_index do |x, i|
 puts %{#{i}=>"#{x}"}
end
```

Note that you can't use the **Enumerator** for our array as though it were the actual array. Only the methods of **Enumerable** are supported

Whenever you're tempted to reimplement one of the methods of **Enumerable**, try using an **Enumerator** instead. It's like modifying an object's **each** method, but it doesn't affect the original object.

```
array_with_index = array.enum_with_index
array_with_index.each do |x, i|
 puts %{#{i}=>"#{x}"}
end
# 0=>"bob"
```

```
# 1=>"loves"
# 2=>"alice"
```

When you **require 'enumerator'**, **Enumerable** sprouts two extra enumeration methods, **each_cons** and **each_slice**. These make it easy to iterate over a data structure in chunks.

```
sentence = %w{Well, now I've seen everything!}

two_word_window = sentence.to_enum(:each_cons, 2)
two_word_window.each { |x| puts x.inspect }
# ["Well,", "now"]
# ["now", "I've"]
# ["I've", "seen"]
# ["seen", "everything!"]

two_words_at_a_time = sentence.to_enum(:each_slice, 2)
two_words_at_a_time.each { |x| puts x.inspect }
# ["Well,", "now"]
# ["I've", "seen"]
# ["everything!"]
```

In Ruby 1.9, the **Enumerable::Enumerator** class is part of the Ruby core; you don't need the **require** statement. Also, **each_cons** and **each_slice** are built-in methods of **Enumerable**.

If you're using a particular object and you wish its **collect** method used a different iterator, then you should turn the object into an **Enumerator** and call its **collect** method. But if you're writing a class and you want to expose a new **collect-like** method, you'll have to define a new method.[4] In that case, the best solution is probably to expose a method that returns a custom **Enumerator**: that way, your users can use all the methods of **Enumerable**, not just **collect**.

The simplest way to interrupt execution is to use break. A break statement will jump out of the closest enclosing loop defined in the current method:

```
1.upto(10) do |x|
  puts x
  break if x == 3
end
# 1
# 2
# 3
```

The **break** statement is simple but it has several limitations. You can't use **break** within a code block defined with **Proc.new** or (in Ruby 1.9 and up) **Kernel#proc**.

The **SyncEnumerator** class, defined in the **generator** library, makes it easy to iterate over a bunch of arrays or other **Enumerable** objects in parallel. Its **each** method yields a series of arrays, each array containing one item from each underlying **Enumerable** object.

```
require 'generator'

enumerator = SyncEnumerator.new(%w{Four seven}, %w{score years}, %w{and ago})
enumerator.each do |row|
 row.each { |word| puts word }
 puts '---'
end
# Four
# score
# and
# ---
# seven
# years
# ago
# ---
```

Any object that implements the each method can be wrapped in a Generator object. If you've used Java, think of a Generator as being like a Java Iterator object. It keeps track of where you are in a particular iteration over a data structure.

Normally, when you pass a block into an iterator method like each, that block gets called for every element in the iterator without interruption. No code outside the block will run until the iterator is done iterating. You can stop the iteration by writing a break statement inside the code block, but you can't restart a broken iteration later from the same placeunless you use a Generator.

In Ruby 1.8, the Generator class uses continuations to achieve this trick. It sets bookmarks for jumping out of an iteration and then back in. When you call Generator#next the generator "pumps" the iterator once (yielding a single element), sets a bookmark, and returns control back to your code. The next time you call Generator#next, the generator jumps back to its previously set bookmark and "pumps" the iterator once more.

Ruby 1.9 uses a more efficient implementation based on threads. This implementation calls each Enumerable object's each method (triggering the neverending stream of candy), but it does it in a separate thread for each object. After each piece of candy comes out, Ruby freezes time (pauses the thread) until the next time you call Generator#next.

```
l = ["junk1", 1, "junk2", 2, "junk3", "junk4", 3, "junk5"]
g = Generator.new(l)
p   g.next                      # => "junk1"
p   g.next                      # => 1
```

```
p    g.next                          # => "junk2"
```

Create a method that runs the setup code, yields to a code block (which contains the custom code), then runs the cleanup code. To make sure the cleanup code always runs, even if the custom code throws an exception, use a **begin/finally** block.

```
def between_setup_and_cleanup
 setup
     begin
     yield
 finally
     cleanup
 end
end
```

This useful technique shows up most often when there are scarce resources (such as file handles or database connections) that must be closed when you're done with them, lest they all get used up. A language that makes the programmer remember these resources tends to leak those resources, because programmers are lazy. Ruby makes it easy to be lazy and still do the right thing.

You've probably used this technique already, with the the Kernel#open and File#open methods for opening files on disk. These methods accept a code block that manipulates an already open file. They open the file, call your code block, and close the file once you're done:

```
open('output.txt', 'w') do |out|
   out.puts 'Sorry, the filesystem is also closed.'
end
```

Use a callback system, in which objects register code blocks with each other to be executed as needed. An object can call out to its registered callbacks when it needs something, or it can send notification to the callbacks when it does something.

To implement a callback system, write a "register" or "subscribe" method that accepts a code block. Store the registered code blocks as Proc objects in a data structure: probably an array (if you only have one type of callback) or a hash (if you have multiple types). When you need to call the callbacks, iterate over the data structure and call each of the registered code blocks.

```
module EventDispatcher
 def setup_listeners
     @event_dispatcher_listeners = {}
 end

 def subscribe(event, &callback)
     (@event_dispatcher_listeners[event] ||= []) << callback
 end
```

```
    protected
    def notify(event, *args)
        if @event_dispatcher_listeners[event]
            @event_dispatcher_listeners[event].each do |m|
                m.call(*args) if m.respond_to? :call
            end
        end
        return nil
    end
   end

   class Factory
   include EventDispatcher

    def initialize
        setup_listeners
    end

    def produce_widget(color)
        notify(:new_widget, color)
    end

   end

   class WidgetCounter
    def initialize(factory)
        @counts = Hash.new(0)
        factory.subscribe(:new_widget) do |color|
            @counts[color] += 1
            puts "#{@counts[color]} #{color} widget(s) created since I started
watching."
        end
    end
   end

   f = Factory.new
   WidgetCounter.new(f)
   f.produce_widget("red")
   # 1 red widget(s) created since I started watching.
   f.produce_widget("green")
   # 1 green widget(s) created since I started watching.
   f.produce_widget("red")
   # 2 red widget(s) created since I started watching.
```

Callbacks are an essential technique for making your code extensible. This technique has many names (callbacks, hook methods, plugins, publish/subscribe, etc.) but no matter what terminology is used, it's always the same. One object asks another to call a piece of code (the callback) when some condition is met. This technique works even when the two objects know almost nothing about each other. This makes it ideal for refactoring big, tightly integrated systems into smaller, loosely coupled systems.

In a pure listener system (like the one given in the Solution), the callbacks set up lines of communication that always move from the event dispatcher to the listeners. This is useful when you have a master object (like the **Factory**), from which numerous lackey objects (like the **WidgetCounter**) take all their cues. But in many loosely coupled systems, information moves both ways: the dispatcher calls the callbacks and then uses the return results.

Objects and Classes

Ruby is an object-oriented programming language; this chapter will show you what that really means. Like all modern languages, Ruby supports object-oriented notions like classes, inheiritance, and polymorphism. But Ruby goes further than other languages you may have used. Some languages are strict and some are permissive; Ruby is one of the most permissive languages around.

Strict languages enforce strong typing, usually at compile type: a variable defined as an array can't be used as another data type. If a method takes an array as an argument, you can't pass in an array-like object unless that object happens to be a subclass of the array class or can be converted into an array.

Ruby enforces dynamic typing, or duck typing ("if it quacks like a duck, it is a duck"). A strongly typed language enforces its typing everywhere, even when it's not needed. Ruby enforces its duck typing relative to a particular task. If a variable quacks like a duck, it is oneassuming you wanted to hear it quack. When you want "swims like a duck" instead, duck typing will enforce the swimming, and not the quacking.

Ruby's permissiveness is more a matter of attitude than a technical advancement. Python lets you reopen a class after its original definition and modify it after the fact, but the language syntax doesn't make many allowances for it. It's sort of a dirty little secret of the language. In Ruby, this behavior is not only allowed, it's encouraged. Some parts of the standard library add functionality to built-in classes when imported, just to make it easier for the programmer to write code. The Facets Core library adds dozens of convenience methods to Ruby's standard classes. Ruby is proud of this capability, and urges programmers to exploit it if it makes their lives easier.

Strict languages end up needing code generation tools that hide the restrictions and complexities of the language. Ruby has code generation tools built right into the language, saving you work while leaving complete control in your hands.

Within the code for the object's class, define a variable and prefix its name with an at sign (@). When an object runs the code, a variable by that name will be stored within the object.

If you want to make an instance variable readable from outside the object, call the **attr_reader** method on its symbol. If you want to make an instance variable readable from outside the object, call the **attr_reader** method on its symbol. Similarly, to make an instance variable readable and writable from outside the object, call the **attr_accessor** method on its symbol. There's also an **attr_writer** decorator method, which only generates a setter method, but you won't use it very often.

Some programming languages have complex rules about when one object can directly access to another object's instance variables. Ruby has one simple rule: it's never allowed. To get or set the value of an instance variable from outside the object that owns it, you need to call an explicitly defined getter or setter method.

Given the name for an instance variable, you can retrieve the value with Object#instance_variable_get, and set it with Object#instance_variable_set.

```
class Frog
 def initialize(name)
     @name = name
 end

 def speak
     @speaks_english ||= @name.size > 6
     @speaks_english ? "Hi. I'm #{@name}, the talking frog." : 'Ribbit.'
 end
end

f = Frog.new('Leonard')
p f.speak
p f.instance_variable_get("@name")
f.instance_variable_set("@name", "Michael")
p f.instance_variable_get("@name")
```

Instance variables are prefixed by a single at sign; class variables are prefixed by two at signs. Class variables store information that's applicable to the class itself, or applicable to every instance of the class. They're often used to control, prevent, or react to the instantiation of the class. A class variable in Ruby acts like a static variable in Java.

Whenever possible, you should use duck typing (**Object#respond_to**?) in preference to class typing (**Object#is_a**?). Duck typing is one of the great strengths of Ruby, but it only works if everyone uses it. If you write a method that only accepts strings, instead of accepting anything that supports **to_str**, then you've broken the duck typing illusion for everyone who uses your code.

Ruby programmers use subclassing less frequently than they would in other languages, because it's often acceptable to simply reopen an existing class (even a built-in class) and attach a new method. We do this throughout this book, adding useful new methods to built-in classes rather than defining them in **Kernel**, or putting them in subclasses or utility classes. Libraries like Rails and Facets Core do the same.

A Ruby class can have only one method with a given name. Within that single method, though, you can put logic that branches depending on how many and what kinds of objects were passed in as arguments.

Ruby's loose typing eliminates most of the need for method overloading. Its default arguments, variable-length argument lists, and (simulated) keyword arguments eliminate most of the remaining cases.

Ruby never lets one object access another object's instance variables. All you can do is call methods. Ruby simulates instance variable access by making it easy to define getter and setter methods whose names are based on the names of instance variables. When you access object.my_var, you're actually calling a method called my_var, which (by default) just happens to return a reference to the instance variable my_var.

When you're defining a class, you can have Ruby generate a setter method for one of your instance variables by calling **Module#atttr_writer** or **Module#attr_accessor** on the symbol for that variable. This saves you from having to write code, but the default setter method lets anyone set the instance variable to any value at all.

Ruby accessor methods usually correspond to the names of the instance variables they access, but this is nothing more than a convention. Outside code has no way of knowing what your instance variables are called, or whether you have any at all, so you can create accessors for virtual attributes with no risk of outside code thinking they're backed by real instance variables.

```
class Arc
 attr_accessor :radians

 def degrees
     @radians * 180 / Math::PI
 end

 def degrees=(degrees)
     @radians = degrees * Math::PI / 180
 end
end

arc = Arc.new
arc.degrees = 180
p arc.radians                          # => 3.14159265358979
arc.radians = Math::PI / 2
p arc.degrees                          # => 90.0
```

By default, an object's inspect method works the same way as its to_s method. Unless your classes override inspect, inspecting one of your objects will

yield a boring and not terribly helpful string, containing only the object's class name, object_id, and instance variables:

```
class Dog
 def initialize(name, age)
     @name = name
     @age = age * 7 #Compensate for dog years
 end
end

spot = Dog.new("Spot", 2.1)
puts spot.inspect
p spot
# => "#<Dog:0xb7c16bec @name="Spot", @age=14.7>"
# => "#<Dog:0xb7c16bec @name="Spot", @age=14.7>"
```

A method can take a fixed number of "normal" arguments and then a variable number of "extra" arguments. When defining such a method, just make sure that the last argument is the one you prefix with the asterisk.

Use the super keyword to call the superclass implementation of the current method.

When you call super with no arguments, the arguments to your method are passed to the superclass method exactly as they were recieved by the subclass.

You can call super at any time in the body of a methodbefore, during, or after calling other code. This is in contrast to languages like Java, where you must call super in the method's first statement or never call it at all. If you need to, you can even call super multiple times within a single method.

Often you want to create a subclass method that exposes exactly the same interface as its parent. You can use the *args constructor to make the subclass method accept any arguments at all, then call super with no arguments to pass all those arguments (as well as any attached code block) into the superclass implementation. Let the superclass deal with any problems with the arguments.

If the subclass method takes arguments but the superclass method takes none, be sure to invoke **super** with an empty pair of parentheses. Usually you don't have to do this in Ruby, but **super** is not a real method call. If you invoke **super** without parentheses, it will pass all the subclass arguments into the superclass implementation, which won't be able to handle them.

You want to define a method of a class, but leave it for subclasses to fill in the actual implementations. Define the method normally, but have it do nothing except raise a **NotImplementedError.**

```
class Shape2D
 def area
   raise NotImplementedError.new("#{self.class}#area is an abstract method.")
```

```
  end
end

Shape2D.new.area
# NotImplementedError: Shape2D#area is an abstract method.
```

Ruby doesn't have a built-in notion of an abstract method or class, and though it has many built-in classes that might be considered "abstract," it doesn't enforce this abstractness the way C++ and Java do.

In general, this is in the spirit of Ruby. But it's sometimes useful to define a superclass method that every subclass is expected to implement. The **NotImplementedError** error is the standard way of conveying that a method is not there, whether it's abstract or just an unimplemented stub.

Unlike other programming languages, Ruby will let you instantiate a class that defines an abstract method. You won't have any problems until you actually call the abstract method; even then, you can catch the **NotImplementedError** and recover. If you want, you can make an entire class abstract by making its **initialize** method raise a **NotImplementedError**. Then no one will be able to create instances of your class.

```
class Shape2D
 def initialize
     raise NotImplementedError.new("#{self.class} is an abstract class.")
 end
end

Shape2D.new
# NotImplementedError: Shape2D is an abstract class.
```

When an object is frozen, its instance variables are permanently bound to their current values. The values themselves are not frozen: their instance variables can still be modified, to the extent they were modifiable before.

```
sequences = [[1, 2, 3], [1, 2, 4], [1, 4, 9]].freeze
sequences << [2, 3, 5]
# TypeError: can't modify frozen array
sequences[2] << 16                    # => [1, 4, 9, 16]
```

A frozen object cannot be unfrozen, and if cloned, the clone will also be frozen. Calling **Object#dup** (as opposed to **Object#clone**) on a frozen object yields an unfrozen object with the same instance variables.

```
frozen_string.clone.frozen?                # => true
frozen_string.dup.frozen?                  # => false
```

Constant objects are often frozen as a second line of defense against the object being modified in place. You can freeze an object whenever you need a permanent reference to an object; this is most commonly seen with strings:

```
API_KEY = "100f7vo4gg".freeze

API_KEY[0] = 4
# TypeError: can't modify frozen string

API_KEY = "400f7vo4gg"
# warning: already initialized constant API_KEY
```

Frozen objects are also useful in multithreaded code. For instance, Ruby's internal file operations work from a frozen copy of a filename instead of using the filename directly. If another thread modifies the original filename in the middle of an operation that's supposed to be atomic, there's no problem: Ruby wasn't relying on the original filename anyway. You can adopt this copy-and-freeze pattern in multithreaded code to prevent a data structure you're working on from being changed by another thread.

Another common programmer-level use of this feature is to freeze a class in order to prevent future modifications to it. This is not quite the same as the **final** construct in C# and Java, because you can still subclass a frozen class, and override methods in the subclass. Calling **freeze** only stops the in-place modification of a class. The simplest way to do it is to call **freeze** as the last statement in the class definition.

```
class MyClass
 def my_method
     puts "This is the only method allowed in MyClass."
 end
 freeze
end

class MyClass
 def my_method
     "I like this implementation of my_method better."
 end
end
# TypeError: can't modify frozen class
```

Ruby has two object-copy methods: a quick one and a thorough one. The quick one, Object#dup, creates a new instance of an object's class, then sets all of the new object's instance variables so that they reference the same objects as the original does. Finally, it makes the new object tainted if the old object was tainted.

The downside of dup is that it creates a new instance of the object's original class. If you open up a specific object and give it a singleton method, you implicitly create a metaclass, an anonymous subclass of the original class.

Calling dup on the object will yield a copy that lacks the singleton methods. The other object-copy method, Object#clone, makes a copy of the metaclass and instantiates the copy, instead of instantiating the object's original class.

```
material.definition
# The better half of velour.

'cotton'.definition
# NoMethodError: undefined method 'definition' for "cotton":String

material.clone.definition
# The better half of velour.

material.dup.definition
# NoMethodError: undefined method 'definition' for "cotton":String
```

Object#clone is also more strict about propagating Ruby's internal flags: it will propagate both an object's "tainted?" flag and its "frozen?" flag. If you want to make an unfrozen copy of a frozen object, you must use Object#dup.

Object#clone and Object#dup both perform shallow copies: they make copies of an object without also copying its instance variables. You'll end up with two objects whose instance variables point to the same objects.

If you want to do a deep copy, an easy (though not particularly quick) way is to serialize the object to a binary string with **Marshal**, then load a new object from the string.

```
class Object
 def deep_copy
     Marshal.load(Marshal.dump(self))
 end
end

class StringHolder
 attr_reader :string
 def initialize(string)
     @string = string
 end
end

s1 = StringHolder.new('string')
s2 = s1.deep_copy
s1.string[1] = 'p'
p s1.string                         # => "spring"
p s2.string                         # => "string"
```

Note that this will only work on an object that has no singleton methods.

	Object#clone	Object#dup	Deep copy with Marshal
Same instance variables?	New references to the same objects	New references to the same objects	New objects
Same metaclass?	Yes	No	Yes[5]
Same singleton methods?	Yes	No	N/A[6]
Same frozen state?	Yes	No	No
Same tainted state?	Yes	Yes	Yes

[5] Marshal can't serialize an object whose metaclass is different from its original class.
[6] Marshal can't serialize an object whose metaclass is different from its original class.

Declare the variable as a constant. You can't absolutely prohibit the variable from being assigned a different value, but you can make Ruby generate a warning whenever that happens.

```
A_CONSTANT = 3
A_CONSTANT = 10
# warning: already initialized constant A_CONSTANT
```

A constant variable is one whose name starts with a capital letter. By tradition, Ruby constant names consist entirely of capital letters, numbers, and underscores. Constants don't mesh well with Ruby's philosophy of unlimited changability: there's no way to absolutely prevent someone from changing your constant. However, they are a useful signal to the programmers who come after you, letting them know not to redefine a constant without a very good reason. You change the reference to point to a different object, you'll get a warning. Unfortunately, there's no way to tell Ruby to treat the redeclaration of a constant as an error.

Constants can occur anywhere in code. If they appear within a class or module, you can access them from outside the class or module with the double-colon operator (::).

However, you can use **Module#remove_const** as a sneaky way to "undeclare" a constant. You can then declare the constant again, without even triggering a warning.

```
module Math
 remove_const(:PI)
 PI = 3
end
```

```
p Math::PI      # => 3
```

If a constant points to a mutable object like an array or a string, the object itself can change without triggering the constant warning. Freezing operates on the object, not the reference. It does nothing to prevent a constant reference from being assigned to another object.

```
RGB_COLORS = [:red, :green, :blue]    # => [:red, :green, :blue]
RGB_COLORS << :purple          # => [:red, :green, :blue, :purple]

HOURS_PER_DAY = 24
HOURS_PER_DAY.freeze # This does nothing since Fixnums are already immutable.
HOURS_PER_DAY = 26
# warning: already initialized constant HOURS_PER_DAY
```

To define a class method, prefix the method name with the class name in the method definition. You can do this inside or outside of the class definition.

In Ruby, a singleton method is a method defined on one specific object, and not available to other instances of the same class. This is kind of analagous to the Singleton pattern, in which all access to a certain class goes through a single instance, but the name is more confusing than helpful.

Class methods are actually a special case of singleton methods. The object on which you define a new method is the Class object itself.

When you define a singleton method on an object other than a class, it's usually to redefine an existing method for a particular object, rather than to define a brand new method. This behavior is common in frameworks, such as GUIs, where each individual object has customized behavior. Singleton method definition is a cheap substitute for subclassing when you only need to customize the behavior of a single object.

When you define a method on a particular object, Ruby acts behind the scenes to transform the object into an anonymous subclass of its former class. This new class is the one that actually defines the new method or overrides the methods of its superclass.

```
class Button
 #A stub method to be overridden by subclasses or individual Button objects
   def pushed
   end
end

button_a = Button.new
def button_a.pushed
puts "You pushed me! I'm offended!"
end

button_b = Button.new
```

```
def button_b.pushed
  puts "You pushed me; that's okay."
end

Button.new.pushed       #
button_a.pushed         # You pushed me! I'm offended!
button_b.pushed         # You pushed me; that's okay.
```

Use **private** as a statement before a method definition, and the method will not be callable from outside the class that defined it. Unlike in many other programming languages, a private method in Ruby is accessible to subclasses of the class that defines it. In this way, Ruby's **private** works like Java's **protected**.

Like many parts of Ruby that look like special language features, Ruby's privacy keywords are actually methods. In this case, they're methods of Module. When you call private, protected, or public, the current module (remember that a class is just a special kind of module) changes the rules it applies to newly defined methods from that point on.

Most languages that support method privacy make you put a keyword before every method saying whether it's public, private, or protected. In Ruby, the special privacy methods act as toggles. When you call the private keyword, all methods you define after that point are declared as private, until the module definition ends or you call a different privacy method. This makes it easy to group methods of the same privacy levela good, general programming practice.

```
class MyClass
  def public_method1
  end

  def public_method2
  end

  protected

  def protected_method1
  end


 private

  def private_method1
  end

  def private_method2
  end
```

```
    end
```

Suppose further that you have two instances of the **Foo** class, A and B. In languages like Java, A and B can call each other's private methods. In Ruby, you need to use a **protected** method for that. This is the main difference between private and protected methods in Ruby.

Instance variables are always private: accessible by subclasses, but not from other objects, even other objects of the same class. If you want to make an instance variable accessible to the outside, you should define a getter method with the same name as the variable. This method can be either protected or public.

You can trick a class into calling a private method from outside by passing the method's symbol into Object#send (in Ruby 1.8) or Object#funcall (in Ruby 1.9). You'd better have a really good reason for doing this.

```
    s.send(:secret)
```

Modules and Namespaces

A Ruby module is nothing more than a grouping of objects under a single name. The objects may be constants, methods, classes, or other modules.

Modules have two uses. You can use a module as a convenient way to bundle objects together, or you can incorporate its contents into a class with Ruby's include statement.

When a module is used as a container for objects, it's called a **namespace**. Ruby's Math module is a good example of a namespace.

Modules are also used to package functionality for inclusion in classes. The **Enumerable** module isn't supposed to be used on its own: it adds functionality to a class like **Array** or **Hash**. The only thing you can't do with a module is instantiate an object from it.

A Ruby class can only have one superclass, but it can include any number of modules. These modules are called mixins. If you write a chunk of code that can add functionality to classes in general, it should go into a mixin module instead of a class.

If you come from Java, you might think of a module as being the combination of an interface and its implementation. By including a module, your class implements certain methods, and announces that since it implements those methods it can be treated a certain way.

When a class includes a module with the **include** keyword, all of the module's methods and constants are made available from within that class. They're not copied, the way a method is when you alias it. Rather, the class becomes aware of the methods of the module. If a module's methods are changed later (even during runtime), so are the methods of all the classes that include that module.

Your module can define an **initialize** method, and it will be called by a class whose constructor includes a **super** call.

If two modules define methods with the same name, and a single class includes both modules, the class will have only one implementation of that method: the one from the module that was included last. The method of the same name from the other module will simply not be available.

This rule sidesteps the fundamental problem of multiple inheritance by letting the programmer explicitly choose which ancestor they would like to inherit a particular method from. Nevertheless, it's good programming practice to give distinctive names to the methods in your modules. This reduces the risk of namespace collisions when a class mixes in more than one module. Collisions can occur, and the later module's method will take precedence, even if one or both methods are protected or private.

The **extend** method is used to mix a module's methods into an object, while **include** is used to mix a module's methods into a class. Within the class definition, **extend** is being called on the **Person** class itself: we could have also written **self.extend(SuperPowers)**. We're extending the **Person** class with the methods defined in **SuperPowers**. This means that the methods defined in the **SuperPowers** module have now become class methods of **Person**.

```
class Person
 extend SuperPowers
end

#which is equivalent to:
Person.extend(SuperPowers)
```

Split the big library into multiple files, and set up autoloading for the individual files by calling **Kernel#autoload**. The individual files will be loaded as they're referenced.

```
# functions.rb
autoload :Decidable, "decidable.rb"
autoload :Semidecidable, "semidecidable.rb"
# decidable.rb
module Decidable
 # … Many, many methods go here.
end
# semidecidable.rb
module Semidecidable
 # … Many, many methods go here.
End
```

Refactoring a library to consist of autoloadable components takes a little extra planning, but it's often worth it to improve performance for the people who use your library.

Each call to Kernel#autoload binds a symbol to the path of the Ruby file that's supposed to define that symbol. If the symbol is referenced, that file is loaded exactly as though it had been passed as an argument into require. If the symbol is never referenced, the user saves some memory.

Use include to copy a module's objects into the current namespace. You can then use them from the current namespace, without qualifying their names.

```
Instead of this:
 require 'rexml/document'


 REXML::Document.new(xml)
```

You might write this:

```
require 'rexml/document'
include REXML

Document.new(xml)
```

This is the exact same include statement you use to incorporate a mixin module into a class you're writing. It does the same thing here as when it includes a mixin: it copies the contents of a module into the current namespace.

Here, though, the point isn't to add new functionality to a class or module: it's to save you from having to do so much typing. This technique is especially useful with large library modules like Curses and the Rails libraries.

You can, of course, import a namespace that's nested within a namespace of its own. Instead of this:

```
require 'rexml/parsers/pullparser'

REXML::Parsers::PullParser.new("Some XML")
```

You might write this:

```
require 'rexml/parsers/pullparser'
include REXML::Parsers

PullParser.new("Some XML")
```

You can define and access instance variables within a module's instance methods, but you can't actually instantiate a module. A module's instance variables only exist within objects of a class that includes the module. However, classes don't usually need to know about the instance variables defined by the modules they include.

```
module Timeable
 attr_reader :time_created

 def initialize
     @time_created = Time.now
 end

 def age #in seconds
     Time.now - @time_created
 end
end

class Character
  include Timeable
  attr_reader :name
```

```
    def initialize( name )
        @name = name
        super() #calls Timeable's initialize
    end
end

c = Character.new "Fred"
p c.time_created
# => Mon Mar 27 18:34:31 EST 2006
```

Reflection and Metaprogramming

Probably the most interesting aspect of the Ruby programming philosophy is its use of reflection and metaprogramming to save the programmer from having to write repetitive code.

You can metaprogram in Ruby either by writing normal Ruby code that uses a lot of reflection, or by generating a string that contains Ruby code, and evaluating the string.

Use the Object#class method to get the class of an object as a Class object. Use Class#superclass to get the parent Class of a Class object:

```
p 'a string'.class                  # => String
p 'a string'.class.name             # => "String"
p 'a string'.class.superclass       # => Object
String.superclass                   # => Object
p String.class                      # => Class
p String.class.superclass           # => Module
'a string'.class.new                # => ""
```

A class can have only one superclass, but it may have any number of ancestors. The list returned by Module#ancestors contains the entire inheritance hierarchy (including the class itself), any modules the class includes, and the ever-present Kernel module, whose methods are accessible from anywhere because Object itself mixes it in.

```
p String.superclass     # => Object
p String.ancestors      # => [String, Enumerable, Comparable, Object, Kernel]
```

All Ruby objects implement the **Object#methods** method. It returns an array containing the names of the object's public instance methods.

```
p Object.methods
# => ["name", "private_class_method", "object_id", "new",
# "singleton_methods", "method_defined?", "equal?", ...]
```

To get a list of the singleton methods of some object (usually, but not always, a class), use **Object#singleton_methods.**

```
p Math.singleton_methods
# => ["atan2", "asinh", "cosh", "ldexp", "tan", "log", "acosh", "erfc", ...]
```

Ruby also defines some elementary predicates along the same lines. To see whether a class defines a certain instance method, call **method_defined?** on the class or **respond_to?** on an instance of the class. To see whether a class defines a certain class method, call **respond_to?** on the class.

methods, instance_methods, and **singleton_methods** will only return public

methods, and **method_defined?** will only return true if you give it the name of a public method. Ruby provides analagous methods for discovering protected and private methods, though these are less useful.

Goal	Public	Protected	Private
List the methods of an object	methods or public_methods	protected_methods	private_methods
List the instance methods defined by a class	instance_methods or public_instance_methods	protected_instance_methods	private_instance_methods
List the singleton methods defined by a class	singleton_methods	N/A	N/A
Does this class define such-and-such an instance method?	method_defined? or public_method_defined?	protected_method_defined?	private_method_defined?
Will this object respond to such-and-such an instance method?	respond_to?	N/A	N/A

You want to the name of a method into a reference to the method itself.Use the eponymous Object#method method:

```
s = 'A string'
length_method = s.method(:length) # => #<Method: String#length>
p length_method.arity             # => 0
p length_method.call              # => 8
```

In many programming languages a class, function, or method can't be modified after its initial definition. In other languages, this behavior is possible but not encouraged. For Ruby programmers, the ability to reprogram classes on the fly is just another technique for the toolbox, to be used when necessary. It's most commonly used to add new code to a class, but it can also be used to deploy a drop-in replacement for buggy or slow implementation of a method.

```
class Multiplier
  def double_your_pleasure(pleasure)
    return pleasure * 3 # FIXME: Actually triples your pleasure.
  end
```

```
end

class Multiplier
  alias :double_your_pleasure_BUGGY :double_your_pleasure
  def double_your_pleasure(pleasure)
    return pleasure * 2
  end
end
```

Define the class methods **method_added, method_removed**, and/or **method_undefined**. Whenever the class gets a method added, removed, or undefined, Ruby will pass its symbol into the appropriate callback method.

```
class Tracker
 def important
     "This is an important method!"
 end

 def self.method_added(sym)
     if sym == :important
         raise 'The "important" method has been redefined!'
     else
         puts %{Method "#{sym}" was (re)defined.}
     end
 end

 def self.method_removed(sym)
     if sym == :important
         raise 'The "important" method has been removed!'
     else
         puts %{Method "#{sym}" was removed.}
  end
 end

 def self.method_undefined(sym)
     if sym == :important
         raise 'The "important" method has been undefined!'
     else
         puts %{Method "#{sym}" was removed.}
     end
 end
end

class Tracker
  def new_method
```

```
      'This is a new method.'
    end
  end
  # Method "new_method" was (re)defined.
  class Tracker
    undef :important
  end
  # RuntimeError: The "important" method has been undefined!
```

Define a **method_missing** method for your class. Whenever anyone calls a method that would otherwise result in a **NoMethodError**, the **method_missing** method is called instead. It is passed the symbol of the nonexistent method, and any arguments that were passed in.

```
  class MyClass
   def defined_method
       puts 'This method is defined.'
   end

   def method_missing(m, *args)
       puts "Sorry, I don't know about any #{m} method."
   end
  end

  o = MyClass.new
  o.defined_method
  # => "This method is defined."
  o.undefined_method
  # => "Sorry, I don't know about any undefined_method method."
```

The **method_missing** technique is frequently found in delegation scenarios, when one object needs to implement all of the methods of another object. Rather than defining each method, a class implements **method_missing** as a catch-all, and uses **send** to delegate the "missing" method calls to other objects.

```
  class Library < Array
   def method_missing(m, *args)
       search_by_author_or_title(m.to_s)
   end

   def add_book(author, title)
       self << [author, title]
   end

   def search_by_author(key)
```

```
        reject { |b| !match(b, 0, key) }
    end

    def search_by_author_or_title(key)
        reject { |b| !match(b, 0, key) && !match(b, 1, key) }
    end

    :private
    def private_method

    end

    def match(b, index, key)
        b[index].index(key) != nil
    end
    end

    l = Library.new
    l.add_book("James Joyce", "Ulysses")
    l.add_book("James Joyce", "Finnegans Wake")
    l.add_book("John le Carre", "The Little Drummer Boy")
    l.add_book("John Rawls", "A Theory of Justice")

    p l.search_by_author("John")
    # => [["John le Carre", "The Little Drummer Boy"],
    # ["John Rawls", "A Theory of Justice"]]

    p l.search_by_author_or_title("oy")
    # => [["James Joyce", "Ulysses"], ["James Joyce", "Finnegans Wake"],
    #    ["John le Carre", "The Little Drummer Boy"]]

    p l.oy
    # => [["James Joyce", "Ulysses"], ["James Joyce", "Finnegans Wake"],
    #    ["John le Carre", "The Little Drummer Boy"]]

    p l.James
    # => [["James Joyce", "Ulysses"], ["James Joyce", "Finnegans Wake"]]
```

You can also define a **method_missing** method on a class. This is useful for adding syntactic sugar to factory classes.

```
    class StringFactory
    def StringFactory.method_missing(m, *args)
        return String.new(m.to_s, *args)
    end
```

```
end

p StringFactory.a_string                    # => "a_string"
p StringFactory.another_string               # => "another_string"
```

The method_missing method intercepts all calls to undefined methods, including the mistyped names of calls to "real" methods. This is a common source of bugs. If you run into trouble using your class, the first thing you should do is add debug statements to method_missing, or comment it out altogether.

Programmers have always preferred writing new code to cranking out variations on old code. From lex and yacc to modern programs like Hibernate and Cog, we've always used tools to generate code that would be tedious to write out manually.

Instead of generating code with an external tool, Ruby programmers do it from within Ruby. The nicer technique is to use **define_method** to create a method whose implementation can use the local variables available at the time it was defined.

The built-in decorator methods we've already seen use metaprogramming. The **attr_reader** method takes a string as an argument, and defines a method whose name and implementation is based on that string.

```
class Numeric
 [
     ["add", "+"], ["subtract", "-"], ["multiply", "*",], ["divide", "/"]
 ].each do |method, operator|
     define_method("#{method}_2") do
         method(operator).call(2)
     end
 end
end

p 4.add_2                                   # => 6
p 10.divide_2                                # => 5
```

The most common evalutation method used for metaprogramming is **Module#module_eval**. This method executes a string as Ruby code, within the context of a class or module. Any methods or class variables you define within the string will be attached to the class or module, just as if you'd typed the string within the class or module definition.

```
class String
 define_method('last') do |n|
     self[-n, n]
 end
end
```

```
p "Here's a string.".last(7)                # => "string."

class String
 module_eval %{def last(n)
                 self[-n, n]
               end}
end
p "Here's a string.".last(7)                # => "string."

String.module_eval %{def last(n)
                       self[-n, n]
                     end}

p "Here's a string.".last(7)                # => "string."
```

The **eval** method can execute a string of Ruby code as though you had written in some other part of your application. This magic is made possible by **Binding** objects. You can get a **Binding** at any time by calling **Kernel#binding**, and pass it in to **eval** to recreate your original environment where it wouldn't otherwise be available.

A Binding object is a bookmark of the Ruby interpreter's state. It tracks the values of any local variables you have defined, whether you are inside a class or method definition, and so on.

Once you have a Binding object, you can pass it into eval to run code in the same context as when you created the Binding. All the local variables you had back then will be available. If you called Kernel#binding within a class definition, you'll also be able to define new methods of that class, and set class and instance variables.

```
vice_grips = 10
print_variable('vice_grips', binding)
# The value of vice_grips is 10

class String
 eval %{def last(n)
          self[-n, n]
        end}, binding
end
p "Here's a string.".last(7)                # => "string."
```

Since a **Binding** object contains references to all the objects that were in scope when it was created, those objects can't be garbage-collected until both they and the **Binding** object have gone out of scope.

You can only use **remove_method** to remove a method from a class or

module that explicitly defines it. You'll get an error if you try to remove a method from a class that merely inherits that method. To make a subclass stop responding to an inherited method, you should undefine the method with **undef_method**.

The alias command doesn't make a single method respond to two names, or create a shell method that delegates to the "real" method. It makes an entirely separate copy of the old method under the new name. If you then modify the original method, the alias will not be affected.

This may seem wasteful, but it's frequently useful to Ruby programmers, who love to redefine methods that aren't working the way they'd like. When you redefine a method, it's good practice to first alias the old method to a different name, usually the original name with an _old suffix. This way, the old functionality isn't lost.

```
class Array
 alias :len :length
end

p [1, 2, 3, 4].len                    # => 4
```

Aspect-oriented programming lets you permanently add these aspects to previously defined methods, without having to change any of the code that calls them. It's a good way to modularize your code, and to modify existing code without having to do a lot of metaprogramming yourself. Though less mature, the AspectR library has the same basic features of Java's AspectJ.

```
require 'aspectr'
class Verbose < AspectR::Aspect

 def describe(method_sym, object, *args)
     "#{object.inspect}.#{method_sym}(#{args.join(",")})"
 end

 def before(method_sym, object, return_value, *args)
     puts "About to call #{describe(method_sym, object, *args)}."
 end

 def after(method_sym, object, return_value, *args)
     puts  "#{describe(method_sym,  object,  *args)}  has  returned  "  +
 return_value.inspect + '.'
 end
end

verbose = Verbose.new
stack = []
```

```
verbose.wrap(stack, :before, :after, :push, :pop)

stack.push(10)
# About to call [].push(10).
# [10].push(10) has returned [[10]].

stack.push(4)
# About to call [10].push(4).
# [10, 4].push(4) has returned [[10, 4]].

stack.pop
# About to call [10, 4].pop().
# [10].pop() has returned [4].
```

The **Aspect#wrap** method modifies the methods of some other object or class. You can also undo the effects of a **wrap** call with **Aspect#unwrap**.

Persistence

Simple persistence mechanisms like YAML let you write Ruby data structures to disk and load them back later. This is great for simple programs that don't handle much data. Your program can store its entire state in a disk file, and load the file on its next invocation to pick up where it left off. If you never keep more data than can fit into memory, the simplest way to make it permanent is to store it with YAML, Marshal, or Madeleine, and reload it later.

The simplest way is to use the built-in **yaml** library. When you **require yaml**, all Ruby objects sprout **to_ yaml** methods that convert them to the YAML serialization format.

```
require 'yaml'

p 10.to_yaml            # => "--- 10\n"
p 'ten'.to_yaml          # => "--- ten\n"
puts %w{Brush up your Shakespeare}.to_yaml
# --
# - Brush
# - up
# - your
# - Shakespeare

puts ({ 'star' => 'hydrogen', 'gold bar' => 'gold' }).to_yaml
# --
# star: hydrogen
# gold bar: gold

users = [{:name => 'Bob', :permissions => ['Read']},
        {:name => 'Alice', :permissions => ['Read', 'Write']}]

# Serialize
open('users.txt', 'w') { |f| YAML.dump(users, f) }

# And deserialize
users2 = open("users.txt") { |f| YAML.load(f) }
p users2
# => [{:permissions=>["Read"], :name=>"Bob"},
#    {:permissions=>["Read", "Write"], :name=>"Alice"}]
```

YAML is a human-readable and somewhat cross-language serialization standard. Its format describes the simple data structures common to all modern programming languages. YAML can serialize and deserialize any combination of strings, booleans, numbers, dates and times, arrays (possibly nested arrays),

and hashes (again, possibly nested ones).

You can also use YAML to serialize Ruby-specific objects: symbols, ranges, and regular expressions. Indeed, you can use YAML to serialize instances of custom classes: YAML serializes the class of the object and the values of its instance variables. There's no guarantee, though, that other programming languages will understand what you mean.

Before you get drunk with power, you should know that YAML shares the limitations of other serialization schemes. Most obviously, you can only deserialize objects in an environment like the one in which you serialized them.

```
require 'yaml'
require 'set'
set = Set.new([1, 2, 3])
open("set.txt", "w") { |f| YAML.dump(set, f) }
set2 = open("set.txt") { |f| YAML.load(f) }
p set2
# => #<Set: {1, 2, 3}>
```

YAML can only serialize data; it can't serialize Ruby code or system resources (such as filehandles or open sockets). This means some objects can't be fully converted to YAML. Objects that contain Ruby code will lose their code when dumped to YAML. This means that Proc and Binding objects will turn up empty. Objects with singleton methods will be dumped without them. Classes can't be dumped to YAML at all. But these are all edge cases.

Most data structures, even complex ones, can be serialized to YAML and stay readable to boot.

Use the **Marshal** module, built into Ruby. It works more or less like **YAML**, but it's much faster. The **Marshal.dump** method transforms a data structure into a binary string, which you can write to a file and reconstitute later with **Marshal.load**.

```
p Marshal.load(Marshal.dump(%w{Brush up your Shakespeare}))
# => ["Brush", "up", "your", "Shakespeare"]
```

Marshal is what most programmers coming from other languages expect from a serializer. It's fast (much faster than yaml), and it produces unreadable blobs of binary data. It can serialize almost anything that yaml can and it can also handle a few cases that yaml can't. For instance, you can use Marshal to serialize a reference to a class.

```
NewSet = Marshal.load(Marshal.dump(Set))
p NewSet.new([1, 2, 3])
# => #<Set: {1, 2, 3}>
```

Like YAML, Marshal depends on the presence of the original classes, and you

can't deserialize a reference to a class you don't have.

Like YAML, Marshal only serializes data structures. It can't serialize Ruby code (like Proc objects), or resources allocated by other processes (like filehandles or database connections). However, the two libraries differ in their error handling. YAML tends to serialize as much as it can: it can serialize a File object, but when you deserialize it, you get an object that doesn't point to any actual file. Marshal just gives you an error when you try to serialize a file:

```
open('output', 'w') { |f| Marshal.dump(f) }
# TypeError: can't dump File
```

Run the code only if the global variable **$DEBUG** is true. You can trigger debug mode by passing in the **--debug** switch to the Ruby interpreter, or you can set the variable **$DEBUG** to true within your code.

An exception is an object, and the **Kernel#raise** method creates an instance of an exception class. By default, **Kernel#raise** creates an exception of **RuntimeError** class, which is a subclass of **StandardError**. This in turn is a subclass of **Exception**, the superclass of all exception classes.

```
ObjectSpace.each_object(Class) { |x| puts x if x.name =~ /Error$/ }
# SystemStackError
# LocalJumpError
# EOFError
# IOError
# RegexpError
# ...
```

Rescue the exception with a **begin/rescue** block. The code you put into the **rescue** clause should handle the exception and allow the program to continue executing.

By default, a **rescue** clause rescues exceptions of class **StandardError** or its subclasses. Mentioning a specific class in a **rescue** statement will make it rescue exceptions of that class and its subclasses.

If the code cannot be run (because it's not valid Ruby), **eval** raises an exceptiona **SyntaxError**. This exception is not a subclass of **StandardError**; it's a subclass of **ScriptError**, which is a subclass of **Exception**.

```
def do_it(code)
  eval(code)
rescue SyntaxError
  puts "Cannot do it!"
end

do_it('puts 1 +')
# Cannot do it!
```

If you want to interrogate a rescued exception, you can map the **Exception** object to a variable within the **rescue** clause.

```
begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
end
```

You can also use the special variable $! within a **rescue** block to refer to the most recently raised **Exception**. If you do a **require 'English'**, you can use the **$ERROR_INFO** variable, which is easier to remember.

```
require 'English'

begin
 raise 'Another test exception.'
rescue Exception
 puts $!.message
 puts $ERROR_INFO.message
end
# Another test exception.
# Another test exception.
```

Since $! is a global variable, and might be changed at any time by another thread, it's safer to map each **Exception** object you rescue to an object.

Retry the code that failed by executing a **retry** statement within a **rescue** clause of a code block. **retry** reruns the block from the beginning.

```
def rescue_and_retry
 error_fixed = false
 begin
     puts 'I am before the raise in the begin block.'
     raise 'An error has occurred!' unless error_fixed
     puts 'I am after the raise in the begin block.'
 rescue
     puts 'An exception was thrown! Retrying...'
     error_fixed = true
     retry
 end
 puts 'I am after the begin block.'
end
rescue_and_retry
# I am before the raise in the begin block.
# An exception was thrown! Retrying...
# I am before the raise in the begin block.
# I am after the raise in the begin block.
# I am after the begin block.
```

Use the **logger** library in the Ruby standard library. Use its **Logger** class to send logging data to a file or other output stream.

```
require 'logger'

$LOG = Logger.new($stderr)
#$LOG.level = Logger::ERROR
```

```
$LOG.level = Logger::INFO
#$LOG.level = Logger::WARN
$LOG.info "Error in division!"
$LOG.warn "Error in division!"
$LOG.error "Error in division!"
```

If your log is being stored in a file, you can have Logger rotate or replace the log file when it get too big, or once a certain amount of time has elapsed:

```
# Keep data for the current month only
Logger.new('this_month.log', 'monthly')

# Keep data for today and the past 20 days.
Logger.new('application.log', 20, 'daily')

# Start the log over whenever the log exceeds 100 megabytes in size.
Logger.new('application.log', 0, 100 * 1024 * 1024)
```

If the default log entries are too verbose for you, you have a couple of options. The simplest is to set **datetime_format** to a more concise date format.

```
$LOG.datetime_format = '%Y-%m-%d %H:%M:%S'
$LOG.error('This is a little shorter.')
# E, [2006-03-31T19:35:01#17339] ERROR -- : This is a little shorter.
```

There are lots of assertion methods besides the assert_equal and assert_raise method used in the test classes above: assert_not_equal, assert_nil, and more exotic methods like assert_respond_to. All the assertion methods are defined in the Test::Unit::Assertions module, which is mixed into the Test::Unit::TestCase class.

It all happens behind the scenes. When we required the Test::Unit framework, it passed a block into the method method Kernel#at_exit. This block is guaranteed to be called before the Ruby interpreter exits. It looks like this:

```
$ tail -5 /usr/local/lib/ruby/1.8/test/unit.rb
at_exit do
  unless $! || Test::Unit.run?
    exit Test::Unit::AutoRunner.run
  end
end
```

Once you **require 'breakpoint'**, you can call the **breakpoint** method from anywhere in your application. When the execution hits the **breakpoint** call, the application turns into an interactive Ruby session.

```
require 'breakpoint'

class Foo
 def initialize(init_value)
     @instance_var = init_value
 end

 def bar
     test_var = @instance_var
     puts 'About to hit the breakpoint!'
     breakpoint
     puts 'HERE ARE SOME VARIABLES:'
     puts "test_var: #{test_var}, @instance_var: #{@instance_var}"
 end
end

f = Foo.new('When in the course')
f.bar
```

It's good programming practice to preface each of your methods, classes, and modules with a comment that lets the reader know what's going on. Ruby rewards this behavior by making it easy to transform those comments into a set of HTML pages that document your code. This is similar to Java's JavaDoc, Python's PyDoc, and Perl's Pod.

```
# Takes any number of numeric terms and returns the sum.
#   sum(1, 2, 3)                        # => 6
#   sum(1, -1, 10)                       # => 10
#   sum(1.5, 0.2, 0.3, 1)                 # => 3.0
def sum(*terms)
 terms.inject(0) { |sum, term| sum + term}
end
```

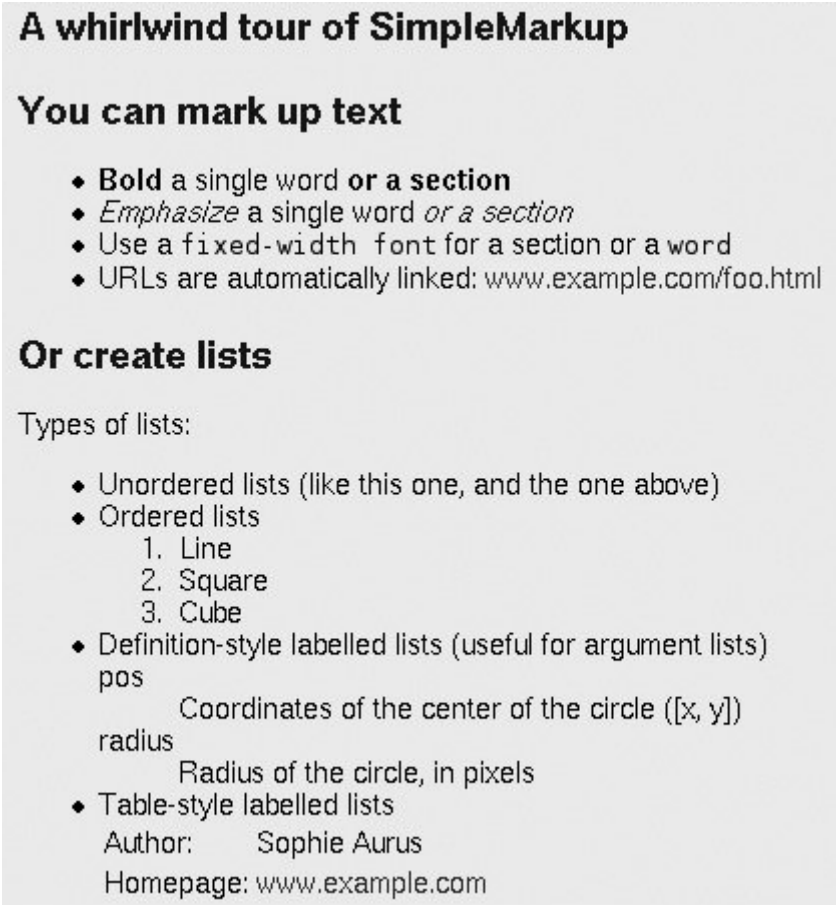
The rdoc command creates a doc/ subdirectory beneath the current directory. It parses every Ruby file it can find in or below the current directory, and generates HTML files from the Ruby code and the comments that document it.

RDoc parses a set of Ruby files, cross-references them, and generates a web site that captures the class and module structure, and the comments you wrote while you were coding.

Generated RDoc makes for a useful reference to your classes and methods, but it's not a substitute for handwritten examples or tutorials. Of course, RDoc comments can contain handwritten examples or tutorials. This will help your users and also help you keep your documentation together with your code.

```
# =A whirlwind tour of SimpleMarkup
#
```

```
# ==You can mark up text
#
# * *Bold* a single word <b>or a section</b>
# * _Emphasize_ a single word <i>or a section</i>
# * Use a <tt>fixed-width font</tt> for a section or a +word+
# * URLs are automatically linked: https://www.example.com/foo.html
#
# ==Or create lists
#
# Types of lists:
# * Unordered lists (like this one, and the one above)
# * Ordered lists
#   1. Line
#   2. Square
#   3. Cube
# * Definition-style labelled lists (useful for argument lists)
#   [pos] Coordinates of the center of the circle ([x, y])
#   [radius] Radius of the circle, in pixels
# * Table-style labelled lists
#   Author:: Sophie Aurus
#   Homepage:: http://www.example.com
```



A whirlwind tour of SimpleMarkup

You can mark up text

- **Bold** a single word **or a section**
- *Emphasize* a single word *or a section*
- Use a `fixed-width font` for a section or a `word`
- URLs are automatically linked: www.example.com/foo.html

Or create lists

Types of lists:

- Unordered lists (like this one, and the one above)
- Ordered lists
  1. Line
  2. Square
  3. Cube
- Definition-style labelled lists (useful for argument lists)

  pos
  : Coordinates of the center of the circle ([x, y])

  radius
  : Radius of the circle, in pixels
- Table-style labelled lists

  Author: Sophie Aurus

  Homepage: www.example.com

There are also several special RDoc directives that go into comments on the same line as a method, class, or module definition. The most common is :nodoc:, which is used if you want to hide something from RDoc. You can and should put an RDoc-style comment even on a :nodoc: method or class, so that people reading your Ruby code will know what it does.

```
# This class and its contents are hidden from RDoc; here's what it does:
class HiddenClass # :nodoc:


end
```

Private methods don't show up in RDoc generated documentationthat would usually just mean clutter. If you want one particular private method to show up in the documentation (probably for the benefit of people subclassing your class), use the **:doc:** directive; it's the opposite of the **:nodoc:** directive.

```
class MyClass
  private

  def hidden_method
  end

  def visible_method # :doc:
  end
end
```

Include the Ruby profiler in your application with **include 'profile'** and the profiler will start tracking and timing every subsequent method call. When the application exits, the profiler will print a report to your program's standard error stream.

The profiler results are ordered with the most time-consuming method calls first. To optimize your code, go from the top of the profiler results and address each call in turn. See why your script led to so many calls of that method, and what you can do about it. Either change the underlying code path so it doesn't call that method so many times, or optimize the method itself. If the method is one you wrote, you can optimize it by profiling it in isolation.

The Ruby profiler sets the interpreter's trace function (by passing a code block into **Kernel#set_trace_func)**, so if your program uses a trace function of its own, using the profiler will overwrite the old function. This probably won't affect you, because the trace function is mainly used by profilers and other analysis tools.

Use the **benchmark** library to time the tasks you want to run. The **Benchmark.bm** method gives you an object that can report on how long it takes for code blocks to run.

```
RANGE = (0..1000)
```

```
array = RANGE.to_a
hash = RANGE.inject({}) { |h,i| h[i] = true; h }

def test_member?(data)
  RANGE.each { |i| data.member? i }
end

require 'benchmark'

Benchmark.bm(10) do |timer|
  timer.report('Array') { test_member?(array) }
  timer.report('Hash') { test_member?(hash) }
end
#              user     system      total        real
#Array      0.381000   0.000000   0.381000 (  0.411000)
#Hash       0.000000   0.000000   0.000000 (  0.000000)
```

The real time is "wall clock" time: the number of seconds that passed in the real world between the start of the test and its completion. This time is actually not very useful, because it includes time during which the CPU was running some other process. If your system is operating under a heavy load, the Ruby interpreter will get less of the CPU's attention and the real times won't reflect the actual performance of your benchmarks. You only need real times when you're measuring user-visible performance on a running system.

The user time is time actually spent running the Ruby interpreter, and the system time is time spent in system calls spawned by the interpreter. If your test does a lot of I/O, its system time will tend to be large; if it does a lot of processing, its user time will tend to be large. The most useful time is probably total, the sum of the user and system times.

Recall that the **real** time can be distorted by the CPU doing things other than running your Ruby process. The **user** and **system** times can also be distorted by the Ruby interpreter doing things besides running your program. To get around these problems, use the Benchmark.bmbm method. It runs each of your timing tests twice. The first time is just a rehearsal to get the interpreter into a stable state. Nothing can completely isolate the time spent running benchmarks from other tasks of the Ruby interpreter, but bmbm should be good enough for most purposes.

If you want to measure one operation instead of comparing several operations to each other, use **Benchmark#measure**. It returns an object that you can interrogate to get the times, or print out to get a listing in the same format as **Benchmark.bm**.

```
def write_to_file
```

```
    File.open('out.txt', 'w') { |f| f.write('a' * 10000) }
end
puts Benchmark.measure { 1000.times { write_to_file } }
# =>   1.251000   0.871000   2.122000 (  3.195000)
```

User Interface

The advantage of this simple interface is that you can use Unix shell tools like redirection and pipes to connect these programs to each other. Instead of manually typing a Ruby program into the interpreter's standard input, you can send it a file with the Unix command **ruby < file.rb**. If you've got another program that generates Ruby code and prints it to standard output, you can pipe the generated code into the interpreter with **generator | ruby**.

The disadvantage is that these programs are not very user-friendly. Libraries like Curses, Readline, and HighLine can add color and sophistication to your terminal programs. The **irb** interactive interpreter uses Readline to offer interactive line editing instead of the simpler interface offered by the Unix shell.

The graphical user interface is the most common interface in the world. Even a web interface is usually interpreted within a GUI on the client end. However, there's not much that's Ruby-specific about GUI programming. All the common GUI libraries (like Tk, GTK, and QT) are written in C, and Ruby's bindings to them look a lot like the bindings for other dynamic languages such as Perl and Python.

HighLine, written by James Edward Gray II and Gregory Brown, is available as the **highline** gem. The Curses and Readline libraries come preinstalled with Ruby (even on Windows, if you use the one-click installer).

The Tk binding for Ruby comes preinstalled with Ruby, assuming you've installed Tk itself.

The HighLine library requires that you install a gem (**highline**), but it makes sophisticated line-oriented input much easier.

```
def confirmation_hearings
 answers = {}
 answers[:name] = ask('What is your name? ')
 answers[:age] = ask('How old are you? ', Integer) { |q| q.in = 0..120 }
 answers[:why] = ask('Why would you like to be Secretary of the Treasury? ')
 puts "Okay, you're confirmed!"
 return answers
end

confirmation_hearings
# What is your name?                              # <= Leonard Richardson
# How old are you?                                # <= twenty-seven
# You must enter a valid Integer.
# ?                                               # <= 200
# Your answer isn't within the expected range (included in 0..120)
# ?                                               # <= 27
# …
```

The **OptionParser** class can parse any command-line arguments you're likely to need, and it includes a lot of Unix know-how that would take a long time to write yourself. All you have to do is define the set of arguments your script accepts, and write code that reacts to the presence of each argument on the command line.

```
    require 'optparse'

    class CatArguments < Hash
     def initialize(args)
         super()
         self[:show_ends] = ''
         self[:number_lines] = false

         opts = OptionParser.new do |opts|
             opts.banner = "Usage: #$0 [options]"
             opts.on('-E', '--show-ends [STRING]',  'display [STRING] at end of
each line') do |string|
                 self[:show_ends] = string || '$'
             end

             opts.on('-n', '--number', 'number all output lines') do
                 self[:number_lines] = true
             end

             opts.on_tail('-h', '--help', 'display this help and exit') do
                 puts opts
                 exit
             end
         end

         opts.parse!(args)
     end
    end

    arguments = CatArguments.new(ARGV)
    p arguments
```

The HighLine library is available as a gem, and it can do this almost as well. You just have to turn off the terminal echo feature:

```
    require 'highline/import'

    def get_password(prompt='Password: ')
      ask(prompt) { |q| q.echo = false}
    end
```

```
get_password("What's your password? ")
# What's your password?
# => "buddy"
```

Use the readline library. Instead of reading directly from standard input, pass a prompt string into Readline.readline. The user will be able to edit their input using the same shortcut keys you can use in the irb Ruby interpreter (assuming their terminal supports those keys). On Windows, this isn't necessary because the **cmd shell** provides any console program with many of **readline**'s features.

```
require 'readline'
vegetable = Readline.readline("What's your favorite vegetable?> ")
puts "#{vegetable.capitalize}? Are you crazy?"
```

Note that you don't have to **chomp** the result of **Readline.readline.**

With the backspace key, you can correct errors one character at a time. But what if you want to insert text into the middle of a line, or delete the whole thing and start over? That's where readline comes in. It's a Ruby interface to the Readline library used by many Unix programs, and it recognizes many control characters besides the backspace.

The **readline** library also supports command history: that's the feature of **irb** that lets you revisit commands you've already typed. To add this feature to your program, pass **true** as the second argument to **Readline.readline**. When the user enters a line, her input will be added to the command history. The next time your code calls **Readline.readline**, the user can hit the up arrow key to recall previous lines of input.

```
line = 0
loop do
  Readline.readline('%.3d> ' % line, true)
  line += 1
end
```

IO.popen is a good way to run noninteractive commandscommands that read all their standard input at once and produce some output.

```
def run(command, input='')
 IO.popen(command, 'r+') do |io|
     io.puts input
     io.close_write
     return io.read.chomp
 end
end

run 'wc -w', 'How many words are in this string?'     # => "7"
```

Utils

```
require 'ftools'

class NumberParser
 @@number_regexps = {
     :to_i   => /([+-]?[0-9]+)/,
     :to_f   => /([+-]?([0-9]*\.)?[0-9]+(e[+-]?[0-9+])?)/i,
     :oct    => /([+-]?[0-7]+)/,
     :hex    => /\b([+-]?(0x)?[0-9a-f]+)\b/i
 }

 def NumberParser.re(parsing_method=:to_i)
     re = @@number_regexps[parsing_method]
     raise ArgumentError, "No regexp for #{parsing_method.inpect}!" unless re
     return re
 end

 def extract(s, parsing_method=:to_i)
     numbers = []
     s.scan(NumberParser.re(parsing_method)) do |match|
         numbers << match[0].send(parsing_method)
     end
     numbers
 end
end

class SortedArray < Array
 def initialize(*args, &sort_by)
     @sort_by = sort_by || Proc.new { |x, y| x <=> y }
     super(*args)
     sort! &sort_by
 end

 def insert(i, v)
     insert_before = index(find { |x| @sort_by.call(x, v) == 1 })
     super(insert_before ? insert_before : -1, v)
 end

 def <<(v)
     insert(0, v)
 end

 alias push <<
```

```
    alias unshift <<

    ["collect!", "  flatten!", "[]="].each do |method_name|
        module_eval %{
            def #{method_name}(*args)
                super
                sort! &@sort_by
            end
        }
    end

    def reverse!
        raise "Reversing the array would disorder it!"
    end

    def +(other_array)
        SortedArray.new(super)
    end

    end

    class String
    def margin!
        self.gsub!(/^\s*\|/m,"")   # Remove leading characters
    end

    def rot13
        self.tr("A-Ma-mN-Zn-z","N-Zn-zA-Ma-m")
    end

    def ip?
        num = "(\\d|[01]?\\d\\d|2[0-4]\\d|25[0-5])"
        ip_pat = Regexp.new("^(#{num}\.){3}#{num}$")
        if self =~ ip_pat
            true
        else
            false
        end
    end

    def utf8?
        unpack('U*') rescue return false
        true
    end
```

```
    def ascii?
        self.split(/./).all? {|ch| ch < 128 }
    end

    def word_count
        frequencies = Hash.new(0)
        downcase.scan(/(\w+([-'.]\w+)*)/) { |word, ignore| frequencies[word] +=
1 }
        return frequencies
    end

    def capitalize_first_letter
        self[0].chr.capitalize + self[1, size]
    end

    def capitalize_first_letter!
        unless self[0] == (c = self[0,1].upcase[0])
            self[0] = c
        end
        self
    end

    def mgsub(key_value_pairs=[].freeze)
        regexp_fragments = key_value_pairs.collect { |k,v| k }
        gsub(Regexp.union(*regexp_fragments)) do |match|
            key_value_pairs.detect{|k,v| k =~ match}[1]
        end
    end

    def email?
        valid = '[A-Za-z\d.+-_]+'
        (self =~ /#{valid}@#{valid}\.#{valid}/) == 0
    end

    def last(n)
        self[-n, n]
    end

   end

   class Float

    def equals?(x, tolerance)
```

```
        (self - x).abs < tolerance
    end

    def            approx?(other,            relative_epsilon=Float::EPSILON,
epsilon=Float::EPSILON)
        difference = other - self
        return true if difference.abs <= epsilon
        relative_error = (difference / (self > other ? self : other)).abs
        return relative_error <= relative_epsilon
    end

   end

   class Fixnum
    def to_str
        self.to_s
    end

    def commas
        str = self.to_s.reverse
        str.gsub!(/([0-9]{3})/,"\\1,")
        str.gsub(/,$/,"").reverse
    end
   end

   class Object
    def accessor?(sym)
        return (self.respond_to?(sym) and self.respond_to?(sym + "="))
    end
   end

   class Array
    include Comparable
    def ^(other)
        (self | other) - (self & other)
    end

    def subset?(other)
        self.each  do |x|
            return false if !(other.include? x)
        end
        true
    end
```

```
    def superset?(other)
        other.subset?(self)
    end

    def randomize
        self.sort_by { rand }
    end

    def randomize!
        self.replace(self.randomize)
    end

    def pick_random
        self[rand(self.length)]
    end

    def shuffle!
        each_index do |i|
            j = rand(length-i) + i
            self[j], self[i] = self[i], self[j]
        end
    end

    def shuffle
        dup.shuffle!
    end

    def to_hash(default=nil)
        if block_given?
          Hash[*inject([]) { |a, value| a.push value, yield(value) }]
        elsif default.nil?
            unless size % 2 == 0
          raise StandardError, "Expected array with even number of elements"
        end
        Hash[*self]
        else
          Hash[*inject([]) { |a, value| a.push value, default }]
        end
    end

    def extract!
        ary = self.dup
        self.reject! { |x| yield x }
        ary - self
```

```
    end

    def extract
        ary = self.reject { |x| yield x }
        self - ary
    end

    def grep_extract!(re)
        extract! { |x| re.match(x) }
    end

    def grep_extract(re)
        extract { |x| re.match(x) }
    end

    def strip_values_at!(*args)
        values = []
        dummy = Object.new
        args.each do |i|
            if i < size
                values << self[i]
                self[i] = dummy
            end
        end
        delete(dummy)
        return values
    end

    def classify
        require 'set'
        h = {}
        each do |i|
            x = yield(i)
            (h[x] ||= []) << i
        end
        h
    end

    def divide(&block)
        Set.new(classify(&block).values)
    end

   end
```

```
    module Enumerable
     def sort_by_frequency
         histogram = inject(Hash.new(0)) { |hash, x| hash[x] += 1; hash }
         sort_by { |x| [histogram[x], x] }
     end

     def sort_by_frequency_descending
         histogram = inject(Hash.new(0)) { |hash, x| hash[x] += 1; hash }
         sort_by { |x| [histogram[x] * -1, x] }
     end

     def sort_distinct_by_frequency
         histogram = inject(Hash.new(0)) { |hash, x| hash[x] += 1; hash }
         histogram.keys.sort_by { |x| [histogram[x], x] }
     end

     def sort_distinct_by_frequency_descending
         histogram = inject(Hash.new(0)) { |hash, x| hash[x] += 1; hash }
         histogram.keys.sort_by { |x| [histogram[x] * -1, x] }
     end

     def min_n(n, &block)
         block = Proc.new { |x, y| x <=> y } if block == nil
         stable = SortedArray.new(&block)
         each do |x|
             stable << x if stable.size < n or block.call(x, stable[-1]) == -1
             stable.pop until stable.size <= n          # Pop the old max if
necessary
         end
         return stable
     end

     def max_n(n, &block)
         block = Proc.new { |x, y| x <=> y } if block == nil
         stable = SortedArray.new(&block)
         each do |x|
             stable << x if stable.size < n or block.call(x, stable[0]) == 1

             stable.shift until stable.size <= n        # Shift the old min if
necessary
         end
         return stable
     end
```

```
    def to_histogram
        inject(Hash.new(0)) { |h, x| h[x] += 1; h }
    end

    def find_no_more_than(limit)
        inject([]) do |a, e|
            a << e if yield e
            return a if a.size >= limit
            a
        end
    end

   end

   class Hash
    def delete_value(value)
        delete_if { |k, v| v == value }
    end

    def remove_hash(other_hash)
        delete_if { |k, v| other_hash[k] == v }
    end

    def safe_invert
        new_hash = {}
        self.each do |k, v|
            if v.is_a? Array
                v.each { |x| new_hash.add_or_append(x, k) }
            else
                new_hash.add_or_append(v, k)
            end
        end
        return new_hash
    end

    def add_or_append(key, value)
        if has_key?(key)
            self[key] = [value, self[key]].flatten
        else
            self[key] = value
        end
    end

    def tied_with(hash)
```

```
        new_hash = {}
        each do |k, v|
            new_hash[hash[k]] = v;
        end.delete_if { |key, value| key.nil? || value.nil? }
        new_hash
    end

    def grep(pattern)
        inject([]) do |res, kv|
            res << kv if kv[0] =~ pattern or kv[1] =~ pattern
            res
        end
    end

    def grep_by_value(pattern)
        inject([]) do |res, kv|
            res << kv if kv[1] =~ pattern
            res
        end
    end

    def grep_by_key(pattern)
        inject([]) do |res, kv|
            res << kv if kv[0] =~ pattern
            res
        end
    end

end

class File
    U_R = 0400
    U_W = 0200
    U_X = 0100
    G_R = 0040
    G_W = 0020
    G_X = 0010
    O_R = 0004
    O_W = 0002
    O_X = 0001

    A_R = 0444
    A_W = 0222
    A_X = 0111
```

```
    def File.fancy_chmod(permisson_string, file)
        mode = File.lstat(file).mode
        permisson_string.scan(/[ugoa][+-=][rwx]+/) do |setting|
            who = setting[0..0]
            setting[2..setting.size].each_byte do |perm|
                perm = perm.chr.upcase
                mask = eval("File::#{who.upcase}_#{perm}")
                (setting[1] == ?+) ? mode |= mask : mode ^= mask
            end
        end
        File.chmod(mode)
    end

    def File.from_dir(dir, name)
        dir = dir.path if dir.is_a? Dir
        path = File.join(dir, name)
        (File.directory?(path) ? Dir : File).open(path) { |f| yield f }
    end

    def each_chunk(chunk_size=1024)
        yield read(chunk_size) until eof?
    end

    def File.same_contents?(p1, p2)
        return false if File.exists?(p1) != File.exists?(p2)
        return true if !File.exists?(p1)
        return true if File.expand_path(p1) == File.expand_path(p2)
        return false if File.ftype(p1) != File.ftype(p2) || File.size(p1) !=
File.size(p2)
        open(p1) do |f1|
            open(p2) do |f2|
                blocksize = f1.lstat.blksize
                same = true
                while same && !f1.eof? && !f2.eof?
                    same = f1.read(blocksize) == f2.read(blocksize)
                end
                return same
            end
        end
    end

    def File.versioned_file(base, first_suffix='.0')
        suffix = nil
```

```
        filename = base
        while File.exists?(filename)
            suffix = (suffix ? suffix.succ : first_suffix)
            filename = base + suffix
        end
        return filename
    end

    def File.backup(filename, move=false)
        new_filename = nil
        if File.exists? filename
            new_filename = File.versioned_file(filename)
            File.send(move ? :move : :copy, filename, new_filename)
        end
        return new_filename
    end

    end

    module EventDispatcher
    def setup_listeners
        @event_dispatcher_listeners = {}
    end

    def subscribe(event, &callback)
        (@event_dispatcher_listeners[event] ||= []) << callback
    end

    protected
    def notify(event, *args)
        if @event_dispatcher_listeners[event]
            @event_dispatcher_listeners[event].each do |m|
                m.call(*args) if m.respond_to? :call
            end
        end
        return nil
    end
    end

    class Class

    def abstract(*args)
        args.each do |method_name|
            define_method(method_name) do |*args|
```

```
                if method_name == :initialize
                    msg = "#{self.class.name} is an abstract class."
                else
                    msg = "#{self.class.name}##{method_name} is an abstract
method."
                end
                raise NotImplementedError.new(msg)
            end
        end
    end

    def hierarchy
        (superclass ? superclass.hierarchy : []) << self
    end

   end

   class Object
    def deep_copy
        Marshal.load(Marshal.dump(self))
    end

    def my_methods_only
        my_super = self.class.superclass
        return my_super ? methods - my_super.instance_methods : methods
    end

    def must_support(*args)
        args.each do |method|
            unless respond_to? method
                raise ArgumentError, %{Must support "#{method}"}
            end
        end
    end
   end

   class Regexp
    def Regexp.valid?(str)
        valid = false
        begin
            compile(str)
            valid = true
        rescue RegexpError
            valid = false
```

```
        end
        valid
    end
    end

    module Contract
    def get_contract(input)
        if @user_defined and @user_defined[input]
            @user_defined[input]
        else
            case input
            when :number
                lambda { |x| x.is_a? Numeric }
            when :string
                lambda { |x| x.respond_to? :to_str }
            when :anything
                lambda { |x| true }
            else
                lambda { |x| false }
            end
        end
    end

    class ContractViolation < StandardError
    end

    def define_contract(inputs={}.freeze)
        @user_defined ||= {}
        inputs.each do |name, contract|
            @user_defined[name] = contract if contract.respond_to? :call
        end
    end

    def contract(method, *inputs)
        @contracts ||= {}
        @contracts[method] = inputs
        contract_added(method)
    end

    def setup_contract(method, inputs)
        @contracts[method] = nil
        method_renamed = "__#{method}".intern
        conditions = ""
```

```
        inputs.flatten.each_with_index do |input, i|
            conditions << %{
             if not self.class.get_contract(#{input.inspect}).call(args[#{i}])
                    raise  ContractViolation,  "argument  #{i+1}  of  method
'#{method}' must " +
                                    "satisfy the '#{input}' contract", caller
                end
            }
        end

        class_eval %{
             alias_method #{method_renamed.inspect}, #{method.inspect}
             def #{method}(*args)
                 #{conditions}
                 return #{method_renamed}(*args)
             end
         }
    end

    def contract_added(method)
        inputs = @contracts[method]
        setup_contract(method, inputs) if inputs
    end

   end

   class ScriptLines

    attr_reader :name
    attr_accessor :bytes, :lines, :lines_of_code, :comment_lines

    LINE_FORMAT = '%10s %10s %10s %10s    %s'

    def self.headline
        sprintf LINE_FORMAT, "BYTES", "LINES", "LOC", "COMMENTS", "FILE"
    end

    def initialize(name)
        @name = name
        @bytes = 0
        @lines = 0
        @lines_of_code = 0
        @comment_lines = 0
    end
```

```
    def read(io)
        in_multiline_comment = false
        io.each { |line|
            @lines += 1
            @bytes += line.size
            case line
            when /^=begin(\s|$)/
                in_multiline_comment = true
                @comment_lines += 1
            when /^=end(\s|$)/
                in_multiline_comment = false
                @comment_lines += 1
            when /^\s*#/
                @comment_lines += 1
            else
                if in_multiline_comment
                    @comment_lines += 1
                else
                    @lines_of_code += 1
                end
            end
        }
    end

    def +(other)
        sum = self.dup
        sum.bytes += other.bytes
        sum.lines += other.lines
        sum.lines_of_code += other.lines_of_code
        sum.comment_lines += other.comment_lines
        sum
    end

    def to_s
        sprintf LINE_FORMAT, @bytes, @lines, @lines_of_code, @comment_lines,
@name
    end

    def self.print_for(*files)
        puts self.headline

        sum = self.new("TOTAL (#{files.size} file(s))")
```

```
        # Print stats for each file.
        files.each do |fn|
            File.open(fn) do |file|
                script_lines = self.new(fn)
                script_lines.read(file)
                sum += script_lines
                puts script_lines
            end
        end

        # Print total stats.
        puts sum
    end
    end

    module Find
    def match(*paths)
        matched = []
        find(*paths) { |path| matched << path if yield path }
        return matched
    end

    def find_bigger_than(bytes, *paths)
        Find.match(*paths) { |p| File.lstat(p).size > bytes }
    end

    def find_smaller_than(bytes, *paths)
        Find.match(*paths) { |p| File.lstat(p).size < bytes }
    end

    def rename(*paths)
        unrenamable = []
        find(*paths) do |file|
            next unless File.file? file # Skip directories, etc.
            path, name = File.split(file)
            new_name = yield name
            if new_name and new_name != name
                new_path = File.join(path, new_name)
                if File.exists? new_path
                    unrenamable << file
                else
                    puts "Renaming #{file} to #{new_path}" if $DEBUG
                    File.rename(file, new_path)
                end
```

```
            end
        end
        return unrenamable
    end

    module_function :match, :find_bigger_than, :find_smaller_than, :rename
end

class Numeric
    def to_degrees
        self * 180 / Math::PI
    end

    def to_radians
        self * Math::PI / 180
    end
end



def debug(if_level)
    yield if ($DEBUG == true) ||($DEBUG && $DEBUG >= if_level)
end

def pdebug(str, if_level=1)
    debug(if_level) { $stderr.puts "DEBUG: " + str }
end

def run(command, input='')
    IO.popen(command, 'r+') do |io|
        io.puts input
        io.close_write
        return io.read.chomp
    end
end
```

Unit Test for Utils

```
require File.join(File.dirname(__FILE__), 'util')
require 'test/unit'
require 'set'
require 'Find'

class UtilTest < Test::Unit::TestCase
 def test_NumberParser
     pw = "Today's numbers are 104 and 391"
     p = NumberParser.new
     assert_equal ["104"], NumberParser.re(:to_i).match(pw).captures
     assert_equal [104, 391], p.extract(pw, :to_i)
     assert_equal [1000], p.extract('The 1000 nights and a night')
     assert_equal [15], p.extract('In octal, fifteen is 017.', :oct)
     assert_equal [60.5], p.extract('The price is $60.50', :to_f)
 end

 def test_SortedArray
     a = SortedArray.new([10, 6, 4, -4, 200, 100])
     assert_equal [-4, 4, 6, 10, 100, 200], a
     a[3] = 25
     assert_equal [-4, 4, 6, 25, 100, 200], a
 end

 def test_String
     str = <<-EOF.margin!
         |This here-document has a "left margin"
         |at the vertical bar on each line.
         |
         |  We can do inset quotations,
         |  hanging indentions, and so on.
     EOF

     expect_str = <<-EOF
This here-document has a "left margin"
at the vertical bar on each line.

  We can do inset quotations,
  hanging indentions, and so on.
EOF
     assert_equal expect_str, str

     joke = "Y2K bug"
```

```
        joke13 = joke.rot13
        assert_equal "L2X oht", joke13

        ip1 = "9.53.97.102"
        ip2 = "0.0.0.0"
        assert_equal(true, ip1.ip?)
        assert_equal(true, ip2.ip?)

        sword = "\303\251p\303\251e"
        assert_equal(true, sword.utf8?)

        text = "abcdefg"
        assert_equal(true, text.ascii?)

        expect = {"the" => 1,"he's" => 1, "man-about-town" => 1, "quite" => 1}
        assert_equal expect, %{He's quite the man-about-town.}.word_count

        assert_equal "I told", "i told".capitalize_first_letter
        assert_equal "I told", "i told".capitalize_first_letter!
        assert_equal "I told", "I told".capitalize_first_letter
        assert_equal "I told", "I told".capitalize_first_letter!

        result = "GO HOME!".mgsub([[/.*GO/i, 'Home'], [/home/i, 'is where the
heart is']])
        assert_equal "Home is where the heart is!", result
        result = "Here is number #123".mgsub([[/[a-z]/i, '#'], [/#/, 'P']])
        assert_equal "#### ## ###### P123", result

        assert_equal true, 'joe+ruby-mail@example.com'.email?
        assert_equal true, 'joe_bloggs@mail.example.com'.email?

        assert_equal "string.", "Here's a string.".last(7)

    end

    def test_Fixnum
        str = "The number is " + 345
        assert_equal "The number is 345", str
        assert_equal("12,345", 12345.commas)
        assert_equal("1,234,589", 1234589.commas)
    end

    def test_Float
        assert_equal true, (3.1416).equals?(Math::PI, 0.001)
```

```
        assert_equal true, 100.2.approx?(100.1 + 0.1)
        assert_equal true, 10e10.approx?(10e10 + 1e-5)
        assert_equal false, 100.0.approx?(100 + 1e-5)
    end

    def test_Array
        assert_equal(true, [1, 2, 3] < [1, 2, 3, 4])
        assert_equal(true, [1, 2, 3] <= [1, 2, 3, 4])
        assert_equal(true, [1, 2, 3] == [1, 2, 3])
        assert_equal(true, [1, 2, 3, 4] > [1, 2, 3])
        assert_equal(true, [1, 2, 3, 4] >= [1, 2, 3])
        x = [1, 2, 3, 4, 5]
        y = [3, 4, 5, 6, 7]
        assert_equal([1, 2, 6, 7], x ^ y)

        a = [1, 2, 3, 4]
        b = [2, 3]
        c = [2, 3, 4, 5]
        assert_equal(false, c.subset?(a))
        assert_equal(true, b.subset?(a))
        assert_equal(true, c.superset?(b))

        expect = {1 => true, 2 => true, 3 => true}
        assert_equal expect, [1, 2, 3].to_hash(true)

        expect = {1 => [-1, 2], 2 => [-2, 4], 3 => [-3, 6]}
        assert_equal expect, [1, 2, 3].to_hash { |value| [value * -1, value * 2] }

        expect = {1 => 1, 2 => 3, 4 => 9}
        assert_equal expect, [1, 1, 2, 3, 4, 9].to_hash
        assert_raises(StandardError) {[1, 1, 2, 3, 4].to_hash}

        a = ("a".."h").to_a
        assert_equal ["a", "c", "d"], a.extract! { |x| x < "e" && x != "b" }
        assert_equal ["b", "e", "f", "g", "h"], a

        a = ("a".."h").to_a
        assert_equal ["a", "c", "d"], a.extract { |x| x < "e" && x != "b" }
        assert_equal ["a", "b", "c", "d", "e", "f", "g", "h"], a

        a = ("a".."h").to_a
        assert_equal ["a", "e"], a.grep_extract!(/[aeiou]/)
        assert_equal ["b", "c", "d", "f", "g", "h"], a
```

```
        a = ("a".."h").to_a
        assert_equal ["a", "e"], a.grep_extract(/[aeiou]/)
        assert_equal ["a", "b", "c", "d", "e", "f", "g", "h"], a

        a = ("a".."h").to_a
        assert_equal ["b", "a", "g"], a.strip_values_at!(1, 0, -2)
        assert_equal ["c", "d", "e", "f", "h"], a

        expect = Set.new([[2, 6, 6], [1, 1, 7, 101]])
        assert_equal expect, [1, 1, 2, 6, 6, 7, 101].divide { |x| x % 2 }

    end

    def test_Enumerable
        assert_equal  [3,  8,  9,  16,  2,  2,  1,  1,  1,  4,  4,  4],
[1,2,3,4,1,2,4,8,1,4,9,16].sort_by_frequency
        assert_equal  [1,  1,  1,  4,  4,  4,  2,  2,  3,  8,  9,  16],
[1,2,3,4,1,2,4,8,1,4,9,16].sort_by_frequency_descending
        assert_equal     [3,     8,     9,     16,     2,     1,     4],
[1,2,3,4,1,2,4,8,1,4,9,16].sort_distinct_by_frequency
        assert_equal     [1,     4,     2,     3,     8,     9,     16],
[1,2,3,4,1,2,4,8,1,4,9,16].sort_distinct_by_frequency_descending

        l = [1, 60, 21, 100, -5, 20, 60, 22, 85, 91, 4, 66]
        assert_equal [60, 66, 85, 91, 100], l.max_n(5)
        assert_equal [-5, 1, 4, 20, 21], l.min_n(5)
        assert_equal [1, 4, -5, 20, 21], l.min_n(5) { |x, y| x.abs <=> y.abs }

        a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        assert_equal [2, 4, 6], a.find_no_more_than(3) { |x| x % 2 == 0 }

    end

    def test_Hash
        h = {'apple' => 'green','potato' => 'red','sun' => 'yellow','katydid' =>
'green'}
        h.delete_value('green')
        expect = {'potato' => 'red', 'sun' => 'yellow'}
        assert_equal expect, h

        squares = { 1 => 1, 2 => 4, 3 => 9 }
        doubles = { 1 => 2, 2 => 4, 3 => 6 }
        squares.remove_hash(doubles)
        expect = {1 => 1, 3 => 9}
```

```
        assert_equal expect, squares

        phone_directory = {
            'Alice' => '555-1212',
            'Bob' => '555-1313',
            'Carol' => '555-1313',
            'Mallory' => '111-1111',
            'Ted' => '555-1212',
        }
        expect = {
            "111-1111" => "Mallory",
            "555-1212" => ["Ted", "Alice"],
            "555-1313" => ["Bob", "Carol"]
        }
        assert_equal expect, phone_directory.safe_invert
        assert_equal phone_directory, phone_directory.safe_invert.safe_invert

        expect = {1 => 1, 2 => 3, 3 => 2}
        assert_equal expect, [1, 2, 2, 2, 3, 3].to_histogram

        expect = {"a" => 2, "b" => 2, "c" => 1, nil => 2}
        assert_equal expect, ["a", "b", nil, "c", "b", nil, "a"].to_histogram

        a = {1 => 2, 3 => 4}
        b = {1 => 'foo', 3 => 'bar'}
        expect = {"foo" => 2, "bar" => 4}
        assert_equal expect, a.tied_with(b)

        expect = {2 => "foo", 4 => "bar"}
        assert_equal expect, b.tied_with(a)

        h = {
            "Apple tree" => "plant",
            "Ficus" => "plant",
       "Shrew" => "animal",
            "Plesiosaur" => "animal",
        }
        expect = [["Ficus", "plant"], ["Apple tree", "plant"], ["Plesiosaur",
"animal"]]
        assert_equal expect, h.grep(/p/i)

        expect = [["Ficus", "plant"], ["Apple tree", "plant"]]
        assert_equal expect, h.grep_by_value(/p/i)
```

```
        expect = [["Apple tree", "plant"], ["Plesiosaur", "animal"]]
        assert_equal expect, h.grep_by_key(/p/i)

    end

    def test_File
        assert_equal 0644, File::U_R | File::U_W | File::G_R | File::O_R
        assert_equal 0644, File::A_R | File::U_W

        expect = "My path is test/subtest/Test.java."
        str = ""
        File.from_dir("test", "subtest") do |subdir|
            File.from_dir(subdir, "Test.java") do |file|
                str = %{My path is #{file.path}.}
            end
        end
        assert_equal expect, str

        assert_equal        false,         File.same_contents?("noexist.txt",
"test/data.txt")
        assert_equal       false,        File.same_contents?("test/data.txt",
"noexist.txt")
        assert_equal true, File.same_contents?("noexist.txt", "noexist.txt")
        assert_equal        true,        File.same_contents?("test/data.txt",
"test/data.txt")
        assert_equal      false,       File.same_contents?("test/output.txt",
"test/data.txt")
        assert_equal    true,    File.same_contents?("test/subtest/data.txt",
"test/data.txt")

    end

    def test_Class
        assert_raise(NotImplementedError) { Shape2D.new }
        assert_raise(NotImplementedError) { Shape.new.area }
        assert_raise(NotImplementedError) { Shape.new.length }

        assert_equal [Object, Array], Array.hierarchy
        assert_equal [Object, Shape2D], Shape2D.hierarchy

    end

    def test_Object
        s1 = StringHolder.new('string')
```

```
        s2 = s1.deep_copy
        s1.string[1] = 'p'
        assert_equal "spring", s1.string
        assert_equal "string", s2.string

        assert_equal ["string"], s1.my_methods_only
        assert_equal ["my_string_method"], MyString.new.my_methods_only

        obj = "a string"
        assert_raise(ArgumentError) { obj.must_support(:-) }

    end

    def test_Regexp
        assert_equal true, Regexp.valid?("\\d+(\\s+)e?")
        assert_equal false, Regexp.valid?("The)horror")
    end

    def test_Contract
        tc = TestContract.new
        assert_raise(Contract::ContractViolation)  {  tc.hello(-2,  'world',
$stdout) }
        assert_raise(Contract::ContractViolation) { tc.hello(-2, 123, $stdout) }
        assert_raise(Contract::ContractViolation) { tc.hello(-2, 123, $stdin) }
    end

    def test_Find
        expect = ["test/Test.java", "test/subtest/Test.java"]
        assert_equal expect , Find.match("test") { |p| p =~ /java/}
    end

   end

   class Shape2D
    abstract :initialize
   end

   class Shape
    abstract :area, :length
   end

   class StringHolder
    attr_reader :string
    def initialize(string)
```

```
        @string = string
    end
   end

   class MyString < String
    def my_string_method
    end
   end

   class TestContract
    extend Contract

    writable_and_open = lambda do |x|
        x.respond_to?('write') and x.respond_to?('closed?') and not x.closed?
    end

    define_contract(:writable => writable_and_open, :positive => lambda {|x| x
>= 0 })

    def hello(n, s, f)
        n.times { f.write "hello #{s}!\n" }
    end

    contract :hello, [:positive, :string, :writable]
   end
```