Chapter 9

Expressions

So far we've been fairly cavalier in our use of expressions in Ruby. After all, `a = b + c` is pretty standard stuff. You could write a whole heap of Ruby code without reading any of this chapter.

But it wouldn't be as much fun ;-).

One of the first differences with Ruby is that anything that can reasonably return a value does: just about everything is an expression. What does this mean in practice?

Some obvious things include the ability to chain statements together:

```
a = b = c = 0                 # =>  0
[ 3, 1, 7, 0 ].sort.reverse   # =>  [7, 3, 1, 0]
```

Perhaps less obvious, things that are normally statements in C or Java are expressions in Ruby. For example, the `if` and `case` statements both return the value of the last expression executed:

```
song_type = if song.mp3_type == MP3::Jazz
              if song.written < Date.new(1935, 1, 1)
                Song::TradJazz
              else
                Song::Jazz
              end
            else
              Song::Other
            end
 rating = case votes_cast
          when 0...10    then Rating::SkipThisOne
          when 10...50   then Rating::CouldDoBetter
          else                Rating::Rave
          end
```

We'll talk more about `if` and `case` starting on page 156.

Operator Expressions

Ruby has the basic set of operators (+, -, *, /, and so on) as well as a few surprises. A complete list of the operators, and their precedences, is given in Table 22.4 on page 345.

In Ruby, many operators are actually implemented as method calls. For example, when you write `a*b+c`, you're actually asking the object referenced by `a` to execute the method `*`, passing in the parameter `b`. You then ask the object that results from that calculation to execute the `+` method, passing `c` as a parameter. This is equivalent to writing the following (perfectly valid) Ruby:

```
a, b, c = 1, 2, 3
a * b + c       # =>   5
(a.*(b)).+(c)   # =>   5
```

Because everything is an object and because you can redefine instance methods, you can always redefine basic arithmetic if you don't like the answers you're getting:

Download **samples/tutexpressions_4.rb**

```
class Fixnum
  alias old_plus +   # We can reference the original '+' as 'old_plus'

  def +(other)       # Redefine addition of Fixnums. This is a BAD IDEA!
    old_plus(other).succ
  end
end

1 + 2       # =>   4
a = 3
a += 4      # =>   8
a + a + a   # =>   26
```

More useful is that classes you write can participate in operator expressions just as if they were built-in objects. For example, the left shift operator, `<<`, is often used to mean *append to receiver*. Arrays support this:

```
a = [ 1, 2, 3 ]
a << 4   # =>   [1, 2, 3, 4]
```

You can add similar support to your classes:

Download **samples/tutexpressions_6.rb**

```
class ScoreKeeper
  def initialize
    @total_score = 0
    @count = 0
  end
  def <<(score)
    @total_score += score
    @count += 1
    self
  end
```

```
  def average
    fail "No scores" if @count == 0
    Float(@total_score) / @count
  end
end
scores = ScoreKeeper.new
scores << 10 << 20 << 40
puts "Average = #{scores.average}"
```

produces:

```
Average = 23.3333333333333
```

Note that there's a subtlety in this code—the << method explicitly returns self. It does this to allow the method chaining in the line scores << 10 << 20 << 40. Because each call to << returns the scores object, you can then call << again, passing in a new score.

As well as the obvious operators, such as +, *, and <<, indexing using square brackets is also implemented as a method call. When you write this:

```
some_obj[1,2,3]
```

you're actually calling a method named [] on some_obj, passing it three parameters. You'd define this method using this:

```
class SomeClass
  def [](p1, p2, p3)
    # ...
  end
end
```

Similarly, assignment to an element is implemented using the []= method. This method receives each object passed as an index as its first *n* parameters and the value of the assignment as its last parameter:

Download **samples/tutexpressions_9.rb**

```
class SomeClass
  def []=(*params)
    value = params.pop
    puts "Indexed with #{params.join(', ')}"
    puts "value = #{value.inspect}"
  end
end
s = SomeClass.new
s[1] = 2
s['cat', 'dog'] = 'enemies'
```

produces:

```
Indexed with 1
value = 2
Indexed with cat, dog
value = "enemies"
```

Miscellaneous Expressions

As well as the obvious operator expressions and method calls and the (perhaps) less obvious statement expressions (such as if and case), Ruby has a few more things that you can use in expressions.

Command Expansion

If you enclose a string in backquotes (sometimes called *backticks*) or use the delimited form prefixed by %x, it will (by default) be executed as a command by your underlying operating system. The value of the expression is the standard output of that command. Newlines will not be stripped, so it is likely that the value you get back will have a trailing return or linefeed character.

```
`date`                   # =>   "Mon Apr 13 13:25:58 CDT 2009\n"
`ls`.split[34]           # =>   "ext_c_win32ole.tip"
%x{echo "Hello there"}   # =>   "Hello there\n"
```

You can use expression expansion and all the usual escape sequences in the command string:

```
for i in 0..3
  status = `dbmanager status id=#{i}`
  # ...
end
```

The exit status of the command is available in the global variable $?.

Redefining Backquotes

In the description of the command output expression, we said that the string in backquotes would "by default" be executed as a command. In fact, the string is passed to the method called Kernel.` (a single backquote). If you want, you can override this. This example uses $?, which contains the status of the last external process run:

Download **samples/tutexpressions_12.rb**

```
alias old_backquote `
def `(cmd)
  result = old_backquote(cmd)
  if $? != 0
    puts "*** Command #{cmd} failed: status = #{$?.exitstatus}"
  end
  result
end
print `ls -l /etc/passwd`
print `ls -l /etc/wibble`
```

produces:

```
-rw-r--r--  1 root  wheel  2888 Sep 23  2007 /etc/passwd
ls: /etc/wibble: No such file or directory
*** Command ls -l /etc/wibble failed: status = 1
```

Assignment

Just about every example we've given so far in this book has featured assignment. Perhaps it's about time we said something about it.

An assignment statement sets the variable or attribute on its left side (the *lvalue*) to refer to the value on the right (the *rvalue*). It then returns that rvalue as the result of the assignment expression. This means you can chain assignments, and you can perform assignments in some unexpected places:

```
a = b = 1 + 2 + 3
a   # =>  6
b   # =>  6
a = (b = 1 + 2) + 3
a   # =>  6
b   # =>  3
File.open(name = gets.chomp)
```

Ruby has two basic forms of assignment. The first assigns an object reference to a variable or constant. This form of assignment is hardwired into the language:

```
instrument = "piano"
MIDDLE_A   = 440
```

The second form of assignment involves having an object attribute or element reference on the left side. These forms are special, because they are implemented by calling methods in the lvalues, which means you can override them.

We've already seen how to define a writable object attribute. Simply define a method name ending in an equals sign. This method receives as its parameter the assignment's rvalue. We've also seen that you can define [] as a method:

Download **samples/tutexpressions_15.rb**

```
class ProjectList
  def initialize
    @projects = []
  end
  def projects=(list)
    @projects = list.map(&:upcase)  # store list of names in uppercase
  end
  def [](offset)
    @projects[offset]
  end
end

list = ProjectList.new
list.projects = %w{ strip sand prime sand paint sand paint rub paint }
list[3]   # =>  "SAND"
list[4]   # =>  "PAINT"
```

As this example shows, these attribute-setting methods don't have to correspond with internal instance variables, and you don't need an attribute reader for every attribute writer (or vice versa).

In older Ruby versions, the result of the assignment was the value returned by the attribute-setting method. As of Ruby 1.8, the value of the assignment is *always* the value of the parameter; the return value of the method is discarded. In the code that follows, older versions of Ruby would set a to 99. Now a will be set to 2.

Download **samples/tutexpressions_16.rb**

```
class Test
  def val=(val)
    @val = val
    return 99
  end
end

t = Test.new
a = (t.val = 2)
a   # =>   2
```

Parallel Assignment

During your first week in a programming course (or the second semester if it was a party school), you may have had to write code to swap the values in two variables:

```
int a = 1;
int b = 2;
int temp;
temp = a;
a = b;
b = temp;
```

You can do this much more cleanly in Ruby:

```
a = 1
b = 2
a, b = b, a
```

Ruby lets you have a comma-separated list of rvalues (the things on the right of the assignment). Once Ruby sees more than one rvalue in an assignment, the rules of parallel assignment come into play. What follows is a description at the logical level: what happens inside the interpreter is somewhat hairier. Users of older versions of Ruby should note that these rules have changed in Ruby 1.9.

1.9

First, all the rvalues are evaluated, left to right, and collected into an array (unless they are already an array). This array will be the eventual value returned by the overall assignment.

Next, the left side (lhs) is inspected. If it contains a single element, the array is assigned to that element.

```
a = 1, 2, 3, 4     # a=[1, 2, 3, 4]
b = [1, 2, 3, 4]   # b=[1, 2, 3, 4]
```

If the lhs contains a comma, Ruby matches values on the rhs against successive elements on the lhs. Excess elements are discarded.

```
a, b = 1, 2, 3, 4   # a=1, b=2
c, = 1, 2, 3, 4     # c=1
```

Splats and Assignment

If Ruby sees any splats on the right side of an assignment (that is, rvalues preceded by an asterisk), each will be expanded inline into its constituent values during the evaluation of the rvalues and before the assignment to lvalues starts:

```
a, b, c, d, e = *(1..2), 3, *[4, 5]   # a=1, b=2, c=3, d=4, e=5
```

Exactly one lvalue may be a splat. This makes it greedy—it will end up being an array, and that array will contain as many of the corresponding rvalues as possible. So, if the splat is the last lvalue, it will soak up any rvalues that are left after assigning to previous lvalues:

```
a, *b = 1, 2, 3   # a=1, b=[2, 3]
a, *b = 1         # a=1, b=[]
```

If the splat is not the last lvalue, then Ruby ensures that the lvalues that follow it will all receive values from rvalues at the end of the right side of the assignment—the splat lvalue will soak up only enough rvalues to leave one for each of the remaining lvalues. (OK, that's a pretty tortuous explanation—some examples will help.)

```
*a, b = 1, 2, 3, 4            # a=[1, 2, 3], b=4
c, *d, e = 1, 2, 3, 4         # c=1, d=[2, 3], e=4
f, *g, h, i, j = 1, 2, 3, 4   # f=1, g=[], h=2, i=3, j=4
```

As with method parameters, you can use a raw asterisk to ignore some rvalues:

```
first, *, last = 1,2,3,4,5,6   # first=1, last=6
```

Nested Assignments

Parallel assignments have one more feature worth mentioning. The left side of an assignment may contain a parenthesized list of terms. Ruby treats these terms as if they were a nested assignment statement. It extracts the corresponding rvalue, assigning it to the parenthesized terms, before continuing with the higher-level assignment.

```
a, (b, c), d = 1,2,3,4       # a=1, b=2, c=nil, d=3
a, (b, c), d = [1,2,3,4]     # a=1, b=2, c=nil, d=3
a, (b, c), d = 1,[2,3],4     # a=1, b=2, c=3, d=4
a, (b, c), d = 1,[2,3,4],5   # a=1, b=2, c=3, d=5
a, (b,*c), d = 1,[2,3,4],5   # a=1, b=2, c=[3, 4], d=5
```

Other Forms of Assignment

In common with many other languages, Ruby has a syntactic shortcut: `a = a + 2` may be written as `a += 2`.

The second form is converted internally to the first. This means that operators you have defined as methods in your own classes work as you'd expect:

Download samples/tutexpressions_26.rb

```
class Bowdlerize
  def initialize(string)
    @value = string.gsub(/[aeiou]/, '*')
  end
  def +(other)
    Bowdlerize.new(self.to_s + other.to_s)
  end
  def to_s
    @value
  end
end

a = Bowdlerize.new("damn ")   # =>  d*mn
a += "shame"                  # =>  d*mn sh*m*
```

Something you won't find in Ruby are the autoincrement (++) and autodecrement (--) operators of C and Java. Use the += and -= forms instead.

Conditional Execution

Ruby has several different mechanisms for conditional execution of code; most of them should feel familiar, and many have some neat twists. Before we get into them, though, we need to spend a short time looking at boolean expressions.

Boolean Expressions

Ruby has a simple definition of truth. Any value that is not nil or the constant false is true—"cat", 99, 0, and :a_song are all considered true.

In this book, when we want to talk about a general true or false value, we use regular Roman type: true and false. When we want to refer to the actual constants, we write true and false.

The fact that nil is considered to be false is convenient. For example, IO#gets, which returns the next line from a file, returns nil at the end of file, enabling you to write loops such as this:

```
while line = gets
  # process line
end
```

However, C, C++, and Perl programmers sometimes fall into a trap. The number zero is *not* interpreted as a false value. Neither is a zero-length string. This can be a tough habit to break.

And, Or, and Not

Ruby supports all the standard boolean operators. Both the keyword and and the operator && return their first argument if it is false. Otherwise, they evaluate and return their second argument (this is sometimes known as *shortcircuit* evaluation). The only difference in the two forms is precedence (and binds lower than &&).

```
nil   && 99   # =>   nil
false && 99   # =>   false
"cat" && 99   # =>   99
```

Thus, && and and both return a true value only if both of their arguments are true, as expected.

Similarly, both or and || return their first argument unless it is false, in which case they evaluate and return their second argument.

```
nil   || 99   # =>   99
false || 99   # =>   99
"cat" || 99   # =>   "cat"
```

As with and, the only difference between or and || is their precedence. To make life interesting, and and or have the same precedence, but && has a higher precedence than ||.

A common idiom is to use ||= to assign a value to a variable only if that variable isn't already set:

```
var ||= "default value"
```

This is almost, but not quite, the same as var = var || "default value". It differs in that no assignment is made at all if the variable is already set. In pseudocode, this might be written as var = "default value" unless var or as var || var = "default value".

not and ! return the opposite of their operand (false if the operand is true, and true if the operand is false). And, yes, not and ! differ only in precedence.

All these precedence rules are summarized in Table 22.4 on page 345.

defined?

The defined? operator returns nil if its argument (which can be an arbitrary expression) is not defined; otherwise, it returns a description of that argument. If the argument is yield, defined? returns the string "yield" if a code block is associated with the current context.

```
defined? 1          # =>   "expression"
defined? dummy      # =>   nil
defined? printf     # =>   "method"
defined? String     # =>   "constant"
defined? $_         # =>   "global-variable"
defined? Math::PI   # =>   "constant"
defined? a = 1      # =>   "assignment"
defined? 42.abs     # =>   "method"
defined? nil        # =>   "nil"
```

Comparing Objects

In addition to the boolean operators, Ruby objects support comparison using the methods ==, ===, <=>, =~, eql?, and equal? (see Table 9.1 on the following page). All but <=> are defined in class Object but are often overridden by descendants to provide appropriate semantics. For example, class Array redefines == so that two array objects are equal if they have the same number of elements and the corresponding elements are equal.

1.9

Both == and =~ have negated forms, != and !~. As of Ruby 1.9, the interpreter first looks for methods called != or !~, calling them if found. If not, it will then invoke either == or =~, negating the result.

In the following example, Ruby calls the == method to perform both comparisons:

Download **samples/tutexpressions_32.rb**

```
class T
  def ==(other)
    puts "Comparing self == #{other}"
    other == "value"
  end
end
t = T.new
p(t == "value")
p(t != "value")
```

produces:

```
Comparing self == value
true
Comparing self == value
false
```

If instead we explicitly define !=, Ruby calls it instead:

Download **samples/tutexpressions_33.rb**

```
class T
  def ==(other)
    puts "Comparing self == #{other}"
    other == "value"
  end
  def !=(other)
    puts "Comparing self != #{other}"
    other != "value"
  end
end
t = T.new
p(t == "value")
p(t != "value")
```

produces:

```
Comparing self == value
true
Comparing self != value
false
```

Table 9.1. Common Comparison Operators

Operator	Meaning
==	Test for equal value.
===	Used to compare each of the items with the target in the when clause of a case statement.
<=>	General comparison operator. Returns -1, 0, or $+1$, depending on whether its receiver is less than, equal to, or greater than its argument.
<, <=, >=, >	Comparison operators for less than, less than or equal, greater than or equal, and greater than.
=~	Regular expression pattern match.
eql?	True if the receiver and argument have both the same type and equal values. 1 == 1.0 returns true, but 1.eql?(1.0) is false.
equal?	True if the receiver and argument have the same object ID.

You can use a Ruby range as a boolean expression. A range such as exp1..exp2 will evaluate as false until exp1 becomes true. The range will then evaluate as true until exp2 becomes true. Once this happens, the range resets, ready to fire again. We show some examples of this on page 160.

Prior to Ruby 1.8, you could use a bare regular expression as a boolean expression. This is now deprecated. You can still use the ~ operator (described on page 665) to match $_ against a pattern, but this will probably also disappear in the future.

If and Unless Expressions

An if expression in Ruby is pretty similar to if statements in other languages:

```
if artist == "Gillespie" then
  handle = "Dizzy"
elsif artist == "Parker" then
  handle = "Bird"
else
  handle = "unknown"
end
```

The then keyword is optional if you lay out your statements on multiple lines:

```
if song.artist == "Gillespie"
  handle = "Dizzy"
elsif song.artist == "Parker"
  handle = "Bird"
else
  handle = "unknown"
end
```

However, if you want to lay out your code more tightly, you must separate the boolean expression from the following statements with the then keyword:[1]

```
if artist == "Gillespie" then  handle = "Dizzy"
elsif artist == "Parker" then  handle = "Bird"
else  handle = "unknown"
end
```

You can have zero or more elsif clauses and an optional else clause. And notice that there's no e in the middle of elsif.

As we've said before, an if statement is an expression—it returns a value. You don't have to use the value of an if statement, but it can come in handy:

```
handle = if artist == "Gillespie"
           "Dizzy"
         elsif artist == "Parker"
           "Bird"
         else
           "unknown"
         end
```

Ruby also has a negated form of the if statement:

```
unless duration > 180
  listen_intently
end
```

The unless statement does support else, but most people seem to agree that it's clearer to switch to an if statement in these cases.

Finally, for the C fans out there, Ruby also supports the C-style conditional expression:

```
cost = duration > 180 ? 0.35 : 0.25
```

A conditional expression returns the value of either the expression before or the expression after the colon, depending on whether the boolean expression before the question mark is true or false. In the previous example, if the duration is greater than three minutes, the expression returns 0.35. For shorter durations, it returns 0.25. The result is then assigned to cost.

If and Unless Modifiers

Ruby shares a neat feature with Perl. Statement modifiers let you tack conditional statements onto the end of a normal statement:

```
mon, day, year = $1, $2, $3 if date =~ /(\d\d)-(\d\d)-(\d\d)/
puts "a = #{a}" if $DEBUG
print total unless total.zero?
```

1. Ruby 1.8 allowed you to use a colon character in place of the then keyword. This is no longer supported.

For an if modifier, the preceding expression will be evaluated only if the condition is true. unless works the other way around:

```
File.foreach("/etc/passwd") do |line|
  next if line =~ /^#/              # Skip comments
  parse(line) unless line =~ /^$/   # Don't parse empty lines
end
```

Because if itself is an expression, you can get really obscure with statements such as this:

```
if artist == "John Coltrane"
  artist = "'Trane"
end unless use_nicknames == "no"
```

This path leads to the gates of madness.

Case Expressions

The Ruby case expression is a powerful beast: a multiway if on steroids. And just to make it even more powerful, it comes in two flavors.

The first form is fairly close to a series of if statements; it lets you list a series of conditions and execute a statement corresponding to the first one that's true:

```
case
when song.name == "Misty"
  puts "Not again!"
when song.duration > 120
  puts "Too long!"
when Time.now.hour > 21
  puts "It's too late"
else
  song.play
end
```

The second form of the case statement is probably more common. You specify a target at the top of the case statement, and each when clause lists one or more comparisons:

```
case command
when "debug"
  dump_debug_info
  dump_symbols
when /p\s+(\w+)/
  dump_variable($1)
when "quit", "exit"
  exit
else
  print "Illegal command: #{command}"
end
```

As with if, case returns the value of the last expression executed, and you can use a then keyword if the expression is on the same line as the condition:[2]

```
kind = case year
       when 1850..1889 then "Blues"
       when 1890..1909 then "Ragtime"
       when 1910..1929 then "New Orleans Jazz"
       when 1930..1939 then "Swing"
       when 1940..1950 then "Bebop"
       else                 "Jazz"
       end
```

case operates by comparing the target (the expression after the keyword case) with each of the comparison expressions after the when keywords. This test is done using *comparison* === *target*. As long as a class defines meaningful semantics for === (and all the built-in classes do), objects of that class can be used in case expressions.

For example, regular expressions define === as a simple pattern match:

```
case line
when /title=(.*)/
  puts "Title is #$1"
when /track=(.*)/
  puts "Track is #$1"
when /artist=(.*)/
  puts "Artist is #$1"
end
```

Ruby classes are instances of class Class. The === operator is defined in Class to test whether the argument is an instance of the receiver or one of its superclasses. So (abandoning the benefits of polymorphism and bringing the gods of refactoring down around your ears), you can test the class of objects:

```
case shape
when Square, Rectangle
  # ...
when Circle
  # ...
when Triangle
  # ...
else
  # ...
end
```

1.9

2. Ruby 1.8 allowed you to use a colon character in place of the then keyword. As of Ruby 1.9, this is no longer supported.

Loops

Don't tell anyone, but Ruby has pretty primitive built-in looping constructs.

The while loop executes its body zero or more times as long as its condition is true. For example, this common idiom reads until the input is exhausted:

```
while line = gets
  # ...
end
```

The until loop is the opposite; it executes the body *until* the condition becomes true:

```
until play_list.duration > 60
  play_list.add(song_list.pop)
end
```

As with if and unless, you can use both of the loops as statement modifiers:

```
a = 1
a *= 2  while a < 100
a   # =>   128
a -= 10 until a < 100
a   # =>   98
```

On page 156, in the section on boolean expressions, we said that a range can be used as a kind of flip-flop, returning true when some event happens and then staying true until a second event occurs. This facility is normally used within loops. In the example that follows, we read a text file containing the first ten ordinal numbers ("first," "second," and so on) but print only the lines starting with the one that matches "third" and ending with the one that matches "fifth":

```
file = File.open("ordinal")
while line = file.gets
  puts(line)  if line =~ /third/ .. line =~ /fifth/
end
```

produces:

```
third
fourth
fifth
```

You may find folks who come from Perl writing the previous example slightly differently:

```
file = File.open("ordinal")
while file.gets
  print  if ~/third/ .. ~/fifth/
end
```

produces:

```
third
fourth
fifth
```

This uses some behind-the-scenes magic behavior: gets assigns the last line read to the global variable $_, the ~ operator does a regular expression match against $_, and print with no arguments prints $_. This kind of code is falling out of fashion in the Ruby community and may end up being removed from the language.

The start and end of a range used in a boolean expression can themselves be expressions. These are evaluated each time the overall boolean expression is evaluated. For example, the following code uses the fact that the variable $. contains the current input line number to display line numbers 1 through 3 as well as those between a match of /eig/ and /nin/:

```
File.foreach("ordinal") do |line|
  if (($. == 1) || line =~ /eig/) .. (($. == 3) || line =~ /nin/)
    print line
  end
end
```

produces:

```
first
second
third
eighth
ninth
```

You'll come across a wrinkle when you use while and until as statement modifiers. If the statement they are modifying is a begin/end block, the code in the block will always execute at least one time, regardless of the value of the boolean expression:

```
print "Hello\n" while false
begin
  print "Goodbye\n"
end while false
```

produces:

```
Goodbye
```

Iterators

If you read the beginning of the previous section, you may have been discouraged. "Ruby has pretty primitive built-in looping constructs," it said. Don't despair, gentle reader, for we have good news. Ruby doesn't need any sophisticated built-in loops, because all the fun stuff is implemented using Ruby iterators.

For example, Ruby doesn't have a for loop—at least not the kind you'd find in C, C++, and Java. Instead, Ruby uses methods defined in various built-in classes to provide equivalent, but less error-prone, functionality.

Let's look at some examples:

```
3.times do
  print "Ho! "
end
```

produces:

```
Ho! Ho! Ho!
```

It's easy to avoid fence-post and off-by-one errors; this loop will execute three times, period. In addition to times, integers can loop over specific ranges by calling downto and upto, and all numbers can loop using step. For instance, a traditional "for" loop that runs from 0 to 9 (something like i=0; i < 10; i++) is written as follows:

```
0.upto(9) do |x|
  print x, " "
end
```

produces:

```
0 1 2 3 4 5 6 7 8 9
```

A loop from 0 to 12 by 3 can be written as follows:

```
0.step(12, 3) {|x| print x, " " }
```

produces:

```
0 3 6 9 12
```

Similarly, iterating over arrays and other containers is easy if you use their each method:

```
[ 1, 1, 2, 3, 5 ].each {|val| print val, " " }
```

produces:

```
1 1 2 3 5
```

And once a class supports each, the additional methods in the Enumerable module (documented beginning on page 487 and summarized on page 100) become available. For example, the File class provides an each method, which returns each line of a file in turn. Using the grep method in Enumerable, we could iterate over only those lines that end with a d:

```
File.open("ordinal").grep(/d$/) do |line|
  puts line
end
```

produces:

```
second
third
```

Last, and probably least, is the most basic loop of all. Ruby provides a built-in iterator called loop:

```
loop do
  # block ...
end
```

The loop iterator calls the associated block forever (or at least until you break out of the loop, but you'll have to read ahead to find out how to do that).

For . . . In

Earlier we said that the only built-in Ruby looping primitives were while and until. What's this for thing, then? Well, for is almost a lump of syntactic sugar.

When you write this:

```
for song in playlist
  song.play
end
```

Ruby translates it into something like this:

```
playlist.each do |song|
  song.play
end
```

The only difference between the for loop and the each form is the scope of local variables that are defined in the body. This is discussed on page 165.

You can use for to iterate over any object that responds to the method each, such as an Array or a Range:

```
for i in ['fee', 'fi', 'fo', 'fum']
  print i, " "
end
for i in 1..3
  print i, " "
end
for i in File.open("ordinal").find_all {|line| line =~ /d$/}
  print i.chomp, " "
end
```

produces:

```
fee fi fo fum 1 2 3 second third
```

As long as your class defines a sensible each method, you can use a for loop to traverse its objects:

Download **samples/tutexpressions_64.rb**

```
class Periods
  def each
    yield "Classical"
    yield "Jazz"
    yield "Rock"
  end
end
periods = Periods.new
for genre in periods
  print genre, " "
end
```

produces:

```
Classical Jazz Rock
```

Break, Redo, and Next

The loop control constructs break, redo, and next let you alter the normal flow through a loop or iterator.[3]

break terminates the immediately enclosing loop; control resumes at the statement following the block. redo repeats current iteration of the loop from the start but without reevaluating the condition or fetching the next element (in an iterator). next skips to the end of the loop, effectively starting the next iteration:

```
while line = gets
  next  if line =~ /^\s*#/  # skip comments
  break if line =~ /^END/   # stop at end
  # substitute stuff in backticks and try again
  redo if line.gsub!(/`(.*?)`/) { eval($1) }
  # process line ...
end
```

These keywords can also be used within blocks. Although you can use them with any block, they typically make the most sense when the block is being used for iteration:

```
i=0
loop do
  i += 1
  next if i < 3
  print i
  break if i > 4
end
```

produces:

```
345
```

A value may be passed to break and next. When used in conventional loops, it probably makes sense only to do this with break, where it sets the value returned by the loop. (Any value given to next is effectively lost.) If a conventional loop doesn't execute a break, its value is nil.

```
result = while line = gets
           break(line) if line =~ /answer/
         end
process_answer(result) if result
```

If you want the nitty-gritty details of how break and next work with blocks and procs, take a look at the reference description starting on page 365. If you are looking for a way of exiting from nested blocks or loops, take a look at Kernel.catch, described on pages 369 and 567.

3. Prior versions of Ruby also supported the retry keyword as a looping mechanism. This has been removed in Ruby 1.9.

Variable Scope, Loops, and Blocks

The while, until, and for loops are built into the language and do not introduce new scope; previously existing locals can be used in the loop, and any new locals created will be available afterward.

The blocks used by iterators (such as loop and each) are a little different. Normally, the local variables created in these blocks are not accessible outside the block:

```
[ 1, 2, 3 ].each do |x|
  y = x + 1
end
[ x, y ]
```

produces:

```
prog.rb:4:in `<main>': undefined local variable or method `x' for
 main:Object (NameError)
```

1.9 However, if at the time the block executes a local variable already exists with the same name as that of a variable in the block, the existing local variable will be used in the block. Its value will therefore be available after the block finishes. As the following example shows, this applies to normal variables in the block but not to the block's parameters:

```
x = "initial value"
y = "another value"
[ 1, 2, 3 ].each do |x|
  y = x + 1
end
[ x, y ]   # =>   ["initial value", 4]
```

Note that the assignment to the variable doesn't have to be executed; the Ruby interpreter just needs to have seen that the variable exists on the left side of an assignment:

```
if false
  a = "never used"
end
3.times {|i| a = i }

a   # =>   2
```

1.9 Ruby 1.9 introduced the concept of block-local variables. These are listed in the block's parameter list, preceded by a semicolon. Contrast this code, which does not use block-locals:

```
square = "yes"
total = 0
[ 1, 2, 3 ].each do |val|
  square = val * val
  total += square
end
puts "Total = #{total}"
puts "Square = #{square}"
```

produces:

```
Total = 14
Square = 9
```

with the following code, which uses a block-local variable, so square in the outer scope is not affected by a variable of the same name within the block:

```
square = "yes"
total = 0
[ 1, 2, 3 ].each do |val; square|
  square = val * val
  total += square
end
puts "Total = #{total}"
puts "Square = #{square}"
```

produces:

```
Total = 14
Square = yes
```

If you are concerned about the scoping of variables with blocks, turn on Ruby warnings, and declare your block-local variables explicitly.