```
#! /usr/bin/env ruby


#-----------------------------------------------#
#               Learn to Program                #
#                 by Chris Pine                 #
#             Copyright (c) 2003-2014           #
#                 chris@pine.fm                 #
#-----------------------------------------------#


require 'cgi'
require 'stringio'


LINK_ADDR    = '/LearnToProgram/'
FRLTP_ADDR   = 'http://www.pragprog.com/titles/ltp2/learn-to-program-2nd-edition'
RUBY_DL_ADDR = 'https://www.ruby-lang.org/en/downloads/'

TRANSLATIONS = [
# ['Japanese'      , 'Shin Nishiyama'    , 'http://www1.tf.chiba-
u.jp/~shin/tutorial/'                                ],
  ['French'        , 'Jean-Pierre ANGHEL', 'http://www.ruby-
doc.org/docs/ApprendreProgrammer/Apprendre_a_Programmer.pdf'],
  ['Russian'       , 'Mikhail Shokhirev' ,
'http://www.shokhirev.com/mikhail/ruby/ltp/title.html'                        ],
  ['Danish'        , 'Gunner Carstens'   ,
'http://gcarst.googlepages.com/komgodtigang'                                  ],
  ['Br. Portuguese', 'Fabio Akita et al.',
'http://aprendaaprogramar.rubyonrails.com.br/'                                ],
  ['Bosnian'       , 'Rusmir Gadžo'      ,
'http://sites.google.com/site/rubynabosanskom/'                               ],
  ['Turkish'       , 'Niyazi ATEŞ'       ,
'http://www.niyaziates.com.tr/rubyindex.html'                                 ],
  ['Greek'         , 'Ηλίας Μαργαριτίδης', 'http://www.imargar.gr/wp-
content/uploads/2011/06/egxeiridiometafrasi.pdf'   ],
  ['German'        , 'Anja Stiedl'       ,
'http://www.moccasoft.de/papers/ruby_tutorial/'                               ],
  ['Korean'        , 'engfordev'         , 'http://opentutorials.org/module/11'
],
  ['Spanish'       , 'rubysur'           , 'http://rubysur.org/aprende.a.programar/'
```

```
],
  ['Spanish'       , "David O' Rojo"     , 'http://goo.gl/3UcZi'
],
  ['Italian'       , 'Duccio Armenise'   , 'http://nemboweb.com/corsi/imparare-a-
programmare'                          ],
].sort


class LearnToProgramTutorial
  @@HLINE = '<div class="hline"> </div>'

  # If you decide to change coloring or formatting of this tutorial,
  # you'll want to change these to something appropriate.
  @@NUMBER_COLOR  = 'green'
  @@STRING_COLOR  = 'red'
  @@KEYWORD_COLOR = 'blue'
  @@INPUT_DESC    = 'dotted box'
  @@OUTPUT_DESC   = 'grey box'

  # Delimiter for input to sample code, since it comes out
  # right alongside the output.  Must not be html escapable.
  @@INPUT = "%%%'f'o'o'"


  def initialize
    @depth = 0
    @page  = []
  end

  def getChapter(method)
    CHAPTERS.each do |chap, title, meth|
      return chap if (method == meth)
    end
    'main'
  end

  def selfLink(chap = nil)
    LINK_ADDR+'chap_'+(chap ? getChapter(chap) : '')+'.html'
  end

  def makeLink(name, methodName)
    '<a href="'+selfLink(methodName)+'">'+name+'</a>'
  end

  def puts(string, escapeThis=false)
```

```
    if escapeThis
      string = escapeHTML string
    end
    @page << '  '*@depth+string
  end

  def escapeHTML(string)
    string.gsub(/[&\"<>]/, CGI::Util::TABLE_FOR_ESCAPE_HTML__)
  end

  def escapeOutputNotInput(output)
    md = /#{@@INPUT}.*?#{@@INPUT.reverse}/.match output
    if md
      escapeHTML(md.pre_match) +
      escapeHTML(md[0]).sub(/#{@@INPUT}/,'<span class="L2PinputOutput">').sub(/#
{@@INPUT.reverse}/,'</span>') +
      escapeOutputNotInput(md.post_match)
    else
      escapeHTML output
    end
  end

  def syntaxColor(str)  # str has probably already been html-escaped.
    lines = str.split(/\n/)
    # L2Pcomment
    # L2Pstring
    # L2Pnumber
    # L2Pkeyword
    # L2Pdefinition

    lines.collect! do |line|
      #line += ' '  # for splitting... Do we need this?
      md = /'|#/.match line
      if md  # Comment or string.
        syntaxColor(md.pre_match) +
        if (md[0] == '#')
          '<span class="L2Pcomment">' + md[0] + md.post_match + '</span>'
        else  # Big string time...
          md2 = /(.*?)(^|[^\\])((\\\\)*)'/.match md.post_match
          if (md2)
            md[0] + '<span class="L2Pstring">' + $1 + $2 + $3 +
                    '</span>' + "'" + syntaxColor(md2.post_match)
          else
            md[0]
          end
```

```
      end
    else  # No comment nor string.
      keywords = %w[__FILE__ and end in or self unless __LINE__
                    begin ensure redo super until BEGIN break do
                    false next rescue then when END case else for
                    nil retry true while alias elsif if not
                    return undef yield]

      keywords.each do |keyword|
        line.gsub!(/(\W|^)(#{keyword})(\W|$)/) do
          $1+'<span class="L2Pkeyword">'+$2+'</span>'+$3
        end
      end

      ['def', 'class', 'module'].each do |keyword|
        line.gsub!(/(\W|^)(#{keyword}) +([\w?]+)/) do
          $1 + '<span class="L2Pkeyword">'    + $2 + '</span>' +
              ' <span class="L2Pdefinition">' + $3 + '</span>'
        end
      end

      line.gsub!(/(^|[-{\[( ^+%*\/?;])(\d+(\.\d+)?|\.\d+)/) do
        $1+'<span class="L2Pnumber">'+$2+'</span>'
      end

      line
    end
  end

  lines.join "\n"
end


def input(str)
  str = escapeHTML str
  str.gsub!(/ /, ' ')
  '<span class="L2Pinput">'+str+'</span>'
end

def code(str)
  str = escapeHTML str
  str.gsub!(/ /, ' ')
  str = syntaxColor str
  '<span class="L2Pcode">'+str+'</span>'
end
```

```ruby
def output(str)
  str = escapeHTML str
  str.gsub!(/ /, ' ')
  '<span class="L2Poutput">'+str+'</span>'
end

# This is the cool part...
def executeCode(code, input)
  # Wrap code to catch errors and to stop SystemExit.
  code = <<-END_CODE
    begin
      #{code}
    rescue SystemExit
    rescue Exception => error
      puts error.inspect
    end
  END_CODE

  strIO = StringIO.new

  if !input.empty?
    input = input.join("\n")+"\n"
    input = StringIO.new(input, "r")
    class << strIO; self; end.module_eval do
      ['gets', 'getc', 'read'].each do |meth|
        define_method(meth) do |*params|
          inStr = input.method(meth).call(*params)
          puts @@INPUT+inStr.chomp+(@@INPUT.reverse)  # Echo input.
          inStr
        end
      end
    end
  end

  # Pass these methods to strIO:
  kernelMethods = ['puts', 'putc', 'gets']

  # Swap out Kernel methods...
  kernelMethods.each do |meth|
    Kernel.module_eval "alias __temp__tutorial__#{meth}__ #{meth}"
    Kernel.module_eval do
      define_method(meth) do |*params|
        strIO.method(meth).call(*params)
      end
```

```
    end
  end

  begin
    strIO.instance_eval code
  rescue Exception => error  # Catch parse errors.
    return error.inspect
  end

  # ...and swap them back in.
  kernelMethods.each do |meth|
    Kernel.module_eval "alias #{meth} __temp__tutorial__#{meth}__"
  end

  strIO.string
end


# Tags (or similar)

def para(attributes = {}, &block)
  method_missing(:p, attributes, &block)
end

def prog(execute = [], remark = nil, fakeOutput = nil, &block)
  if !execute
    return progN(&block)
  end

  run = {:input => execute}
  run[:remark    ] = remark     if remark
  run[:fakeOutput] = fakeOutput if fakeOutput
  progN(run, &block)
end

def progN(*trialRuns)
  code = yield

  # Trim leading whitespace.
  lines = code.split $/
  numSpaces = lines[0].length - lines[0].sub(/ */, '').length
  lines.each do |line|
    line.sub!(/ {0,#{numSpaces}}/, '')
  end
  code = lines.join($/)
```

```
  prettyCode = syntaxColor(escapeHTML(code))

  # Spit it out.
  puts '<pre class="L2PcodeBlock">'+prettyCode+'</pre>'

  trialRuns.each do |run|
    if run[:fakeOutput]
      puts '<pre class="L2PoutputBlock">'+escapeHTML(run[:fakeOutput])+'</pre>'
    end
    if run[:remark]
      puts '<p>'+run[:remark]+'</p>'
    end
    output = escapeOutputNotInput(executeCode(code,run[:input]))
    puts '<pre class="L2PoutputBlock">'+$/+output+'</pre>'
  end
  nil
end

# Makes a tag.
def method_missing(methodSymbol, attributes = {})
  methodName = methodSymbol.to_s

  attribString = ''
  attributes.each do |key, val|
    raise methodName if (key.nil? || val.nil?)
    attribString += ' '+key.to_s+'="'+val+'"'
  end
  if (!block_given?)
    puts '<'+methodName+attribString+' />'
  else
    puts '<'+methodName+attribString+'>'
    @depth += 1
    blockReturn = yield
    puts blockReturn if (blockReturn.kind_of?(String))
    @depth -= 1
    puts '</'+methodName+'>'
  end
  nil
end

#
# TEST PAGE FOR FORMATTING
#
```

```
def generateFormattingPage
  h1 { 'Heading 1' }
  h2 { 'Heading 2' }
  h3 { 'Heading 3' }

  para {'Here\'s some code with fake output:'}
  prog [], '...just kidding, dude...', 'FUNKADELIC!' do <<-'END_CODE'
    # Here is some 'Ruby' code.
    # 5 is better than 6.
    # def wilma do end if not in, dude.
    #'

    '#This shouldn\'t cause any problems.'
    'Neither # should this\\'
    'do end if elsif else case when then while def class'
    'or and not next in'
    'to 3 or not to 3, that is 3.7'
    '' + 'x'+''+''+'..'
            '' +'o'

    8
    0.09
    9.9
    5.times {}

    puts 'I love chips.'   # yo 'g'
    puts 5.02 + 8 + 0.002  # s'up, muva

    jimmy = ['yoyoyo', 66]
    jimmy.each do |item|
      puts item.inspect
    end

    puts case 'pumpkin'
      when String then 'yep'
      when Fixnum then 'nope'
      else 'maybe'
    end

    def yummy
      if (4 <= 5)
        'Yummm!'
      elsif (4 == 5)
        'Huh?'
      else
```

```
        while (1 == 2)
          puts 'What?'
        end
      end
    end

    class JustSomeClass
      def initialize
        @var = 5
      end
    end

    puts Math::PI  # Should work.
    puts PI        # Shouldn't work.
    END_CODE
  end
  para {'Here\'s some code with input and output:'}
  prog ['"Chris"', '&26', '<red>'] do <<-END_CODE
    puts 'yoyo...wuddup?'
    puts 'NAME:'
    name = gets.chomp
    puts 'AGE:'
    age = gets.chomp
    puts 'FAVORITE COLOR'
    color = gets.chomp
    puts 'Hello, '+name+', the '+age+'-year-old '+color+' lover.'
    END_CODE
  end

  para do <<-END_PARAGRAPH
    Hello there.  I love #{input 'inputting VARIOUS things'}.  I also get a kick
    out of #{code 'coding various things'}.  There's such a thrill in
    seeing all of the exciting #{output 'output you can get'} from a
    well-written program.
    END_PARAGRAPH
  end

  h2 {'A Few Things to Try'}

  ul do
    li {"Write a program which asks for a person's first name, then middle,
    then last.  Finally, it should greet the person using their full name."}
    li {"Write a program which asks for a person's favorite number.
    Have your program add one to the number, then suggest the result
    as a <em>bigger and better</em> favorite number.
```

```
      (Do be tactful about it, though.)"}
    end

    para do <<-END_PARAGRAPH
      Once you have finished those two programs (and any others you would like to
try),
      let's learn some more (and some more about) #{makeLink('methods',
:generateMethods)}.
      END_PARAGRAPH
    end
  end

  #
  # SETUP
  #

  def generateSetup
    para do <<-END_PARAGRAPH
      When you program a computer, you have to "speak" in a
      language your computer understands:  a programming
      language.  There are lots and lots of different
      languages out there, and many of them are excellent.
      In this tutorial I chose to use my favorite
      programming language, <em>Ruby</em>.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Aside from being my favorite,
      Ruby is also the easiest programming language I have seen
      (and I've seen quite a few).  In fact, that's the real
      reason I'm writing this tutorial:  I didn't decide to write
      a tutorial, and then choose Ruby because it's my favorite;
      instead, I found Ruby to be so easy that I decided there
      really ought to be a good beginner's tutorial which uses
      it.  It's Ruby's simplicity which prompted this tutorial,
      not the fact that it's my favorite.
      (Writing a similar tutorial using
      another language, like C++ or Java, would have required
      hundreds and hundreds of pages.)  But don't think that
      Ruby is a beginner's language just because it is easy!
      It is a powerful, professional-strength programming
      language if ever there was one.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
```

```
      When you write something in a human language, what is
      written is called text.  When you write something in a computer
      language, what is written is called <dfn>code</dfn>.  I
      have included lots of examples of Ruby code throughout
      this tutorial, most of them complete programs you can
      run on your own computer.  To make the code easier to
      read, I have colored parts of the code different
      colors.  (For example, numbers are always
      <span class="L2Pcode"><span class="L2Pnumber">#{@@NUMBER_COLOR}</span>
</span>.)
      Anything you are supposed to type in will be in a
      #{input @@INPUT_DESC}, and anything a program prints
      out will be in a #{output @@OUTPUT_DESC}.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      If you come across something you don't understand, or you
      have a question which wasn't answered, write it down and
      keep reading!  It's quite possible that the answer will
      come in a later chapter.  However, if your question was
      not answered by the last chapter, I will tell you where
      you can go to ask it.  There are lots of wonderful people
      out there more than willing to help; you just need to know
      where they are.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      But first we need to download and install Ruby onto your
      computer.
      END_PARAGRAPH
    end
    h2 {'Windows Installation'}
    para do <<-END_PARAGRAPH
      The Windows installation of Ruby is a breeze.  First, you
      need to <a href="#{RUBY_DL_ADDR}">download Ruby</a>.
      There might be a couple of versions to choose from; this tutorial
      is using version #{RUBY_VERSION}, so make sure what you download is at
      least as recent as that.  (I would just get the latest
      version available.)  Then simply run the installation program.
      It will ask you where you want to install Ruby.  Unless you have
      a good reason for it, I would just install it in the default
      location.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
```

```
    In order to program, you need to be able to write programs and
    to run programs.  To do this, you will need a text editor and a
    command line. My favorite text editor is
    <a href="http://www.sublimetext.com/">Sublime Text</a>.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    It would also be a good idea to create a folder somewhere to keep
    all of your programs.  Make sure that when you save a program,
    you save it into this folder.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    To get to your command line, select Command Prompt from the
    Accessories folder in your start menu.  You will want to
    navigate to the folder where you are keeping your programs.
    Typing #{input 'cd ..'} will take you up one folder, and
    #{input 'cd foldername'} would put you inside the folder
    named <kbd>foldername</kbd>.  To see all of the folders
    in your current folder, type #{input 'dir /ad'}.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    And that's it!  You're all set to
    #{makeLink 'learn to program', :generateNumbers}.
    END_PARAGRAPH
  end
  h2 {'Macintosh Installation'}
  para do <<-END_PARAGRAPH
    If you have Mac OS X 10.2 (Jaguar) or later, then you already have
    Ruby on your system!  What could be easier?
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    In order to program, you need to be able to write programs and
    to run programs.  To do this, you will need a text editor and a
    command line.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    Your command line is accessible through the Terminal
    application (found in Applications/Utilities).
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
```

```
  For a text editor, you can use whatever one you are familiar
  or comfortable with.  My favorite text editor is
  <a href="http://www.sublimetext.com/">Sublime Text</a>.
  If you use TextEdit, however, make sure
  you save your programs as text-only!  Otherwise your programs
  <em>will not work</em>.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  And that's it!  You're all set to
  #{makeLink 'learn to program', :generateNumbers}.
  END_PARAGRAPH
end
h2 {'Linux Installation'}
para do <<-END_PARAGRAPH
  First, you will want to check and see if you have Ruby installed
  already.  Type #{input 'which ruby'}.  If it says something like
  #{output '/usr/bin/which: no ruby in (...)'}, then you need to
  <a href="#{RUBY_DL_ADDR}">download Ruby</a>,
  otherwise see what version of Ruby you are running with
  #{input 'ruby -v'}.  If it is older than the latest stable build
  on the above download page, you might want to upgrade.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  If you are the root user, then you probably don't need any
  instructions for installing Ruby.  If you aren't, you might want
  to ask your system administrator to install it for you.  (That way
  everyone on that system could use Ruby.)
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Otherwise, you can just install it so that only you can use it.
  Move the file you downloaded to a temporary directory, like
  <kbd>$HOME/tmp</kbd>.  If the name of the file is
  <kbd>ruby-1.6.7.tar.gz</kbd>, you can open it with
  #{input 'tar zxvf ruby-1.6.7.tar.gz'}.  Change directory
  to the directory you just created (in this example,
  #{input 'cd ruby-1.6.7'}).
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Configure your installation by typing
  #{input './configure --prefix=$HOME'}).  Next type
  #{input 'make'}, which will build your Ruby interpreter.
```

```
      This might take a few minutes.  After that is done, type
      #{input 'make install'} to install it.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Next, you'll want to add <kbd>$HOME/bin</kbd> to your
      command search path by editing your <kbd>$HOME/.bashrc</kbd>
      file.  (You might have to log out and back in again for
      this to take effect.)  After you do that, test your installation:
      #{input 'ruby -v'}.  If that tells you what version of Ruby you
      have, you can now delete the files
      in <kbd>$HOME/tmp</kbd> (or wherever you put them).
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      And that's it!  You're all set to
      #{makeLink 'learn to program', :generateNumbers}.
      END_PARAGRAPH
    end
  end

  #
  # NUMBERS
  #

  def generateNumbers
    para do <<-END_PARAGRAPH
      Now that you've gotten everything #{makeLink('setup', :generateSetup)},
      let's write a program!  Open up your favorite text
      editor and type in the following:
      END_PARAGRAPH
    end
    prog false do <<-END_CODE
      puts 1+2
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Save your program (yes, that's a program!) as #{input 'calc.rb'}
      (the <strong>.rb</strong> is what we usually put at the end of
      programs written in Ruby).  Now run your program by typing #{input 'ruby
calc.rb'}
      into your command line.  It should have put a #{output '3'} on your screen.
      See, programming isn't so hard, now is it?
      END_PARAGRAPH
    end
```

```
h2 {'Introduction to '+(code 'puts')}
para do <<-END_PARAGRAPH
  So what's going on in that program?  I'm sure you can guess what the
  #{code '1+2'} does; our program is basically the same as:
  END_PARAGRAPH
end
prog false do <<-END_CODE
  puts 3
  END_CODE
end
para do <<-END_PARAGRAPH
  #{code 'puts'} simply writes onto the screen whatever comes after it.
  END_PARAGRAPH
end
h2 {'Integer and Float'}
para do <<-END_PARAGRAPH
  In most programming languages (and Ruby is no exception)
  numbers without decimal points are called <dfn>integers</dfn>, and
  numbers with decimal points are usually called
  <dfn>floating-point numbers</dfn>,
  or more simply, <dfn>floats</dfn>.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Here are some integers:
  END_PARAGRAPH
end
prog false do <<-END_CODE
  5
  -205
  999999999999999999999999
  0
  END_CODE
end
para do <<-END_PARAGRAPH
  And here are some floats:
  END_PARAGRAPH
end
prog false do <<-END_CODE
  54.321
  0.001
  -205.3884
  0.0
  END_CODE
end
```

```
  para do <<-END_PARAGRAPH
    In practice, most programs don't use floats; only integers.
    (After all, no one wants to look at 7.4 emails, or browse 1.8
    webpages, or listen to 5.24 of their favorite songs...)
    Floats are used more for academic purposes (physics experiments and such)
    and for 3D graphics.  Even most money programs use integers; they just
    keep track of the number of pennies!
    END_PARAGRAPH
  end
  h2 {'Simple Arithmetic'}
  para do <<-END_PARAGRAPH
    So far, we've got all the makings of a simple calculator.
    (Calculators always use floats, so if you want your computer
    to act just like a calculator, you should also use floats.)  For addition
    and subtraction, we use <kbd>+</kbd> and <kbd>-</kbd>,
    as we saw.  For multiplication, we use <kbd>*</kbd>,
    and for division we use <kbd>/</kbd>.  Most keyboards have
    these keys in the numeric keypad on the far right side.
    If you have a smaller keyboard or a laptop, though, you can just use
    <kbd>Shift 8</kbd> and <kbd>/</kbd> (same key as the
    <kbd>?</kbd> key).  Let's try to expand our calc.rb program a little.
    Type in the following and then run it.
    END_PARAGRAPH
  end
  prog [], 'This is what the program returns:' do <<-END_CODE
    puts 1.0 + 2.0
    puts 2.0 * 3.0
    puts 5.0 - 8.0
    puts 9.0 / 2.0
    END_CODE
  end
  para do <<-END_PARAGRAPH
    (The spaces in the program are not important; they just make
    the code easier to read.)  Well, that wasn't too surprising.
    Now let's try it with integers:
    END_PARAGRAPH
  end
  prog [], 'Mostly the same, right?' do <<-END_CODE
    puts 1+2
    puts 2*3
    puts 5-8
    puts 9/2
    END_CODE
  end
  para do <<-END_PARAGRAPH
```

```
      Uh... except for that last one!
      But when you do arithmetic with integers, you'll get integer answers.
      When your computer can't get the "right" answer, it always rounds down.
      (Of course, #{output '4'} <em>is</em> the right answer in integer arithmetic
      for #{code '9/2'}; just maybe not the answer you were expecting.)
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Perhaps you're wondering what integer division is good for.  Well, let's
      say you're going to the movies, but you only have $9.  Here in
      Portland, you can see a movie at the Bagdad for 2 bucks.  How many movies
      can you see there?  #{code '9/2'}... #{output '4'} movies.  4.5
      is definitely <em>not</em> the right answer in this case; they will
      not let you watch half of a movie, or let half of you in to
      see a whole movie... some things just aren't divisible.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      So now experiment with some programs of your own!  If you want
      to write more complex expressions, you can use parentheses.
      For example:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts 5 * (12-8) + -15
      puts 98 + (59872 / (13*8)) * -52
      END_CODE
    end
    h2 {'A Few Things to Try'}
    para do
      puts 'Write a program which tells you:'
    end
    ul do
      li {'how many hours are in a year?'}
      li {'how many minutes are in a decade?'}
      li {'how many seconds old are you?'}
      li {'how many chocolates do you hope to eat in your life? <br />'+
          '<em><strong>Warning:</strong>  This part of the program could take a
while to compute!</em>'}
    end
    para do
      puts "Here's a tougher question:"
    end
    ul do
      li {"If I am #{(Time.now - Time.mktime(1976,8,3)).to_i / 1000000} million
```

```
seconds old, how old am I?"}
    end
    para do <<-END_PARAGRAPH
      When you're done playing around with numbers, let's have a look
      at some #{makeLink('letters', :generateLetters)}.
      END_PARAGRAPH
    end
  end

  #
  # LETTERS
  #

  def generateLetters
    para do <<-END_PARAGRAPH
      So we've learned all about #{makeLink('numbers', :generateNumbers)},
      but what about letters?  words?  text?
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      We refer to groups of letters in a program as <dfn>strings</dfn>.  (You can
      think of printed letters being strung together on a banner.)
      To make it easier to see just what part of the code is in a string,
      I'll color strings
      <span class="L2Pcode"><span class="L2Pstring">#{@@STRING_COLOR}</span></span>.
      Here are some strings:
      END_PARAGRAPH
    end
    prog false do <<-END_CODE
      'Hello.'
      'Ruby rocks.'
      '5 is my favorite number... what is yours?'
      'Snoopy says #%^?&*@! when he stubs his toe.'
      '     '
      ''
      END_CODE
    end
    para do <<-END_PARAGRAPH
      As you can see, strings can have punctuation, digits, symbols,
      and spaces in them... more than just letters.  That last string
      doesn't have anything in it at all; we would call that an
      <dfn>empty string</dfn>.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
```

```
      We have been using #{code 'puts'} to print numbers;
      let's try it with some strings:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts 'Hello, world!'
      puts ''
      puts 'Good-bye.'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      That worked out well.  Now try some strings of your own.
      END_PARAGRAPH
    end
    h2 {'String Arithmetic'}
    para do <<-END_PARAGRAPH
      Just as you can do arithmetic on numbers, you can also do
      arithmetic on strings!  Well, sort of... you can add strings, anyway.
      Let's try to add two strings and see what
      #{code 'puts'} does with that.
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts 'I like' + 'apple pie.'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Whoops!  I forgot to put a space between #{code "'I like'"} and #{code "'apple
pie.'"}.
      Spaces don't matter usually, but they matter inside strings.
      (It's true what they say:  computers don't do what you <em>want</em>
      them to do, only what you <em>tell</em> them to do.)  Let's try that again:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts 'I like ' + 'apple pie.'
      puts 'I like' + ' apple pie.'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      (As you can see, it didn't matter which string I added the space to.)
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      So you can add strings, but you can also multiply them!
```

```
      (By a number, anyway.)  Watch this:
      END_PARAGRAPH
    end
    prog [], '(Just kidding... it really does this:)', 'batting her eyes' do <<-
END_CODE
      puts 'blink ' * 4
      END_CODE
    end
    para do <<-END_PARAGRAPH
      If you think about it, this makes perfect sense.  After all,
      #{code '7*3'} really just means #{code '7+7+7'}, so #{code "'moo'*3"} just
      means #{code "'moo'+'moo'+'moo'"}.
      END_PARAGRAPH
    end
    h2 {"#{code '12'} vs #{code "'12'"}"}
    para do <<-END_PARAGRAPH
      Before we get any further, we should make sure we understand the
      difference between <em>numbers</em> and <em>digits</em>.
      #{code '12'} is a number, but #{code "'12'"} is a string of two digits.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Let's play around with this for a while:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts  12  +  12
      puts '12' + '12'
      puts '12  +  12'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      How about this:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts  2  *  5
      puts '2' *  5
      puts '2  *  5'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      These examples were pretty straightforward.  However, if you're not too
      careful with how you mix your strings and your numbers, you might run into...
      END_PARAGRAPH
```

```
end
h2 {'Problems'}
para do <<-END_PARAGRAPH
  At this point you may have tried out a few things which
  <em>didn't</em> work.  If not, here are a few:
  END_PARAGRAPH
end
prog do <<-END_CODE
  puts '12' + 12
  puts '2' * '5'
  END_CODE
end
para do <<-END_PARAGRAPH
  Hmmm... an error message.  The problem is that you
  can't really add a number to a string, or multiply a
  string by another string.  It doesn't make any more sense than does this:
  END_PARAGRAPH
end
prog false do <<-END_CODE
  puts 'Betty' + 12
  puts 'Fred' * 'John'
  END_CODE
end
para do <<-END_PARAGRAPH
  Something else to be aware of:  you can write #{code "'pig'*5"} in a program,
  since it just means #{code '5'} sets of the string #{code "'pig'"} all added
  together.  However,
  you <em>can't</em> write #{code "5*'pig'"}, since that means #{code "'pig'"}
  sets of the number #{code '5'}, which is just silly.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Finally, what if I want a program to print out
  #{output 'You\'re swell!'}?  We can try this:
  END_PARAGRAPH
end
prog false do <<-END_CODE
  puts 'You're swell!'
  END_CODE
end
para do <<-END_PARAGRAPH
  Well, <em>that</em> won't work; I won't even try to run it.
  The computer thought we were done with the string.
  (This is why it's nice to have a text editor which does
  <dfn>syntax coloring</dfn> for you.)  So how do we let
```

```
    the computer know we want to stay in the string?  We have
    to <dfn>escape</dfn> the apostrophe, like this:
    END_PARAGRAPH
  end
  prog do <<-END_CODE
    puts 'You\\'re swell!'
    END_CODE
  end
  para do <<-END_PARAGRAPH
    The backslash is the escape character.  In other words, if you have
    a backslash and another character, they are sometimes translated
    into a new character.  The only things the backslash escapes,
    though, are the apostrophe and the backslash itself.  (If you
    think about it, escape characters must always escape themselves.)
    A few examples are in order here, I think:
    END_PARAGRAPH
  end
  prog do <<-END_CODE
    puts 'You\\'re swell!'
    puts 'backslash at the end of a string:  \\\\'
    puts 'up\\\\down'
    puts 'up\\down'
    END_CODE
  end
  para do <<-END_PARAGRAPH
    Since the backslash does <em>not</em> escape a #{code "'d'"},
    but <em>does</em> escape itself, those last two strings are
    identical.  They don't look the same in the code, but in your
    computer they really are the same.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    If you have any other questions, just
    #{makeLink('keep reading', :generateVariables)}!  I couldn't
    answer every question on <em>this</em> page, after all.
    END_PARAGRAPH
  end
end

#
# VARIABLES
#

def generateVariables
  para do <<-END_PARAGRAPH
```

```
      So far, whenever we have #{code 'puts'}ed a string or a number, the thing
      we #{code 'puts'}ed is gone.  What I mean is, if we wanted to print
      something out twice, we would have to type it in twice:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts '...you can say that again...'
      puts '...you can say that again...'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      It would be nice if we could just type it in once and then hang on to it...
      store it somewhere.
      Well, we can, of course&mdash;otherwise, I wouldn't have brought it up!
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      To store the string in your computer's memory, we need to
      give the string a name.  Programmers often refer to this process
      as <dfn>assignment</dfn>,
      and they call the names <dfn>variables</dfn>.  This variable can be just
      about any sequence of letters and numbers, but the first character
      needs to be a lowercase letter.  Let's try that last program again,
      but this time I will give the string the name #{code 'myString'} (though I
could
      just as well have named it #{code 'str'} or
      #{code 'myOwnLittleString'} or #{code 'henryTheEighth'}).
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      myString = '...you can say that again...'
      puts myString
      puts myString
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Whenever you tried to do something to #{code 'myString'}, the program did it
      to #{code "'...you can say that again...'"} instead.  You can think of the
      variable #{code 'myString'} as "pointing to" the string
      #{code "'...you can say that again...'"}.  Here's a slightly more interesting
example:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      name = 'Patricia Rosanna Jessica Mildred Oppenheimer'
```

```
  puts 'My name is ' + name + '.'
  puts 'Wow!  ' + name + ' is a really long name!'
  END_CODE
end
para do <<-END_PARAGRAPH
  Also, just as we can <em>assign</em> an object to a variable,
  we can <em>reassign</em> a different object to that variable.
  (This is why we call them variables:  because what they
  point to can vary.)
  END_PARAGRAPH
end
prog do <<-END_CODE
  composer = 'Mozart'
  puts composer + ' was "da bomb", in his day.'

  composer = 'Beethoven'
  puts 'But I prefer ' + composer + ', personally.'
  END_CODE
end
para do <<-END_PARAGRAPH
  Of course, variables can point to any kind of object, not just strings:
  END_PARAGRAPH
end
prog do <<-END_CODE
  var = 'just another ' + 'string'
  puts var

  var = 5 * (1+2)
  puts var
  END_CODE
end
para do <<-END_PARAGRAPH
  In fact, variables can point to just about anything...
  except other variables.
  So what happens if we try?
  END_PARAGRAPH
end
prog do <<-END_CODE
  var1 = 8
  var2 = var1
  puts var1
  puts var2

  puts ''
```

```
      var1 = 'eight'
      puts var1
      puts var2
      END_CODE
    end
    para do <<-END_PARAGRAPH
      So first, when we tried to point #{code 'var2'} to #{code 'var1'}, it really
      pointed to #{code '8'} instead (just like #{code 'var1'}
      was pointing to).  Then we had #{code 'var1'} point to
      #{code "'eight'"}, but since #{code 'var2'} was never really
      pointing at #{code 'var1'}, it stays pointing at #{code '8'}.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      So now that we've got variables, numbers, and strings, let's learn how to
      #{makeLink 'mix them all up', :generateConversion}!
      END_PARAGRAPH
    end
  end

  #
  # CONVERSION
  #

  def generateConversion
    para do <<-END_PARAGRAPH
      We've looked at a few different kinds of objects
      (#{makeLink 'numbers', :generateNumbers} and #{makeLink 'letters',
:generateLetters}),
      and we made #{makeLink 'variables', :generateVariables} to point to them;
      the next thing we want to do is to get them all to play nicely together.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      We've seen that if we want a program to print #{output '25'}, the following
      <em>does <strong>not</strong></em> work, because you can't add
      numbers and strings:
      END_PARAGRAPH
    end
    prog false do <<-END_CODE
      var1 = 2
      var2 = '5'

      puts var1 + var2
      END_CODE
```

```
end
para do <<-END_PARAGRAPH
  Part of the problem is that your computer doesn't know if you
  were trying to get #{output '7'} (#{code '2 + 5'}), or if you wanted
  to get #{output '25'} (#{code "'2' + '5'"}).
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Before we can add these together, we need some way of getting the
  string version of #{code 'var1'}, or to get the integer version
  of #{code 'var2'}.
  END_PARAGRAPH
end
h2 {'Conversions'}
para do <<-END_PARAGRAPH
  To get the string version of an object, we simply write
  #{code '.to_s'} after it:
  END_PARAGRAPH
end
prog do <<-END_CODE
  var1 = 2
  var2 = '5'

  puts var1.to_s + var2
  END_CODE
end
para do <<-END_PARAGRAPH
  Similarly, #{code 'to_i'} gives the integer version of an object,
  and #{code 'to_f'} gives the float version.  Let's look at what
  these three methods do (and <em>don't</em> do) a little more closely:
  END_PARAGRAPH
end
prog do <<-END_CODE
  var1 = 2
  var2 = '5'

  puts var1.to_s + var2
  puts var1 + var2.to_i
  END_CODE
end
para do <<-END_PARAGRAPH
  Notice that, even after we got the string version of
  #{code 'var1'} by calling #{code 'to_s'}, #{code 'var1'} was always pointing
  at #{code '2'}, and never at #{code "'2'"}.  Unless we explicitly reassign
  #{code 'var1'} (which requires an #{code '='} sign), it will point
```

```
      at #{code '2'} for the life of the program.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Now let's try some more interesting (and a few just weird) conversions:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts '15'.to_f
      puts '99.999'.to_f
      puts '99.999'.to_i
      puts ''
      puts '5 is my favorite number!'.to_i
      puts 'Who asked you about 5 or whatever?'.to_i
      puts 'Your momma did.'.to_f
      puts ''
      puts 'stringy'.to_s
      puts 3.to_i
      END_CODE
    end
    para do <<-END_PARAGRAPH
      So, this probably gave some surprises.  The first one is pretty
      standard, giving #{output '15.0'}.
      After that, we converted the string #{code "'99.999'"} to a float and
      to an integer.  The float did what we expected; the integer was, as always,
rounded down.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Next, we had some examples of some... <em>unusual</em> strings being converted
      into numbers.  #{code 'to_i'} ignores the first thing it doesn't understand,
      and the rest of the string from that point on.  So the first one
      was converted to #{code '5'}, but the others, since they started with
      letters, were ignored completely... so the computer just picks zero.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Finally, we saw that our last two conversions did nothing at all,
      just as we would expect.
      END_PARAGRAPH
    end
    h2 {'Another Look at '+(code 'puts')}
    para do <<-END_PARAGRAPH
      There's something strange about our favorite method...  Take a look at this:
      END_PARAGRAPH
```

```
    end
    prog do <<-END_CODE
      puts 20
      puts 20.to_s
      puts '20'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Why do these three all print the same thing?  Well, the last two
      should, since #{code '20.to_s'} <em>is</em> #{code "'20'"}.  But what
      about the first one, the integer #{code '20'}?  For that matter, what
      does it even mean to write out <em>the integer</em> 20?  When
      you write a <em>2</em> and then a <em>0</em> on a piece of paper, you
      are writing down a string, not an integer.  <em>The integer</em> 20 is the
number of
      fingers and toes I have; it isn't a <em>2</em> followed by a <em>0</em>.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Well, here's the big secret behind our friend, #{code 'puts'}:  Before
      #{code 'puts'} tries to write out an object, it uses #{code 'to_s'} to
      get the string version of that object.  In fact, the <em>s</em> in
      #{code 'puts'} stands for <em>string</em>; #{code 'puts'} really means
      <dfn>put string</dfn>.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      This may not seem too exciting now, but there are many,
      <em>many</em> kinds of objects in Ruby (you'll even learn how
      to make your own!), and it's nice to know what will happen if
      you try to #{code 'puts'} a really weird object,
      like a picture of your grandmother, or a music file or something.
      But that will come later...
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      In the meantime, we have a few more methods for you, and they
      allow us to write all sorts of fun programs...
      END_PARAGRAPH
    end
    h2 {'The Methods '+(code 'gets')+' and '+(code 'chomp')}
    para do <<-END_PARAGRAPH
      If #{code 'puts'} means <dfn>put string</dfn>, I'm sure you can guess
      what #{code 'gets'} stands for.  And just as #{code 'puts'} always
      spits out strings, #{code 'gets'} will only retrieve strings.  And
```

```
      whence does it get them?
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      From you!  Well, from your keyboard, anyway.  Since your keyboard only
      makes strings, that works out beautifully.  What actually happens
      is that #{code 'gets'} just sits there, reading what you type until
      you press <kbd>Enter</kbd>.  Let's try it out:
      END_PARAGRAPH
    end
    prog ['Is there an echo in here?'] do <<-END_CODE
      puts gets
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Of course, whatever you type in will just get repeated back
      to you.  Run it a few times and try typing in different things.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Now we can make interactive programs!  In this one, type in
      your name and it will greet you:
      END_PARAGRAPH
    end
    prog ['Chris'], '<em>Eek!</em>  I just ran it&mdash;I typed in my name, and this
is what happened:' do <<-END_CODE
      puts 'Hello there, and what\\'s your name?'
      name = gets
      puts 'Your name is ' + name + '?  What a lovely name!'
      puts 'Pleased to meet you, ' + name + '.  :)'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Hmmm... it looks like when I typed in the letters <kbd>C</kbd>,
      <kbd>h</kbd>, <kbd>r</kbd>, <kbd>i</kbd>,
      <kbd>s</kbd>, and then pressed <kbd>Enter</kbd>, #{code 'gets'}
      got all of the letters in my name <em>and</em> the
      <kbd>Enter</kbd>!  Fortunately, there's a method just for
      this sort of thing:  #{code 'chomp'}.  It takes off any <kbd>Enter</kbd>s
      hanging out at the end of your string.  Let's try that program again,
      but with #{code 'chomp'} to help us this time:
      END_PARAGRAPH
    end
    prog ['Chris'] do <<-END_CODE
      puts 'Hello there, and what\\'s your name?'
```

```
      name = gets.chomp
      puts 'Your name is ' + name + '?  What a lovely name!'
      puts 'Pleased to meet you, ' + name + '.  :)'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Much better!  Notice that since #{code 'name'} is pointing to
      #{code 'gets.chomp'}, we don't ever have to say
      #{code 'name.chomp'};  #{code 'name'} was already
      #{code 'chomp'}ed.
      END_PARAGRAPH
    end
    h2 {'A Few Things to Try'}
    ul do
      li {"Write a program which asks for a person's first name, then middle,
      then last.  Finally, it should greet the person using their full name."}
      li {"Write a program which asks for a person's favorite number.
      Have your program add one to the number, then suggest the result
      as a <em>bigger and better</em> favorite number.
      (Do be tactful about it, though.)"}
    end
    para do <<-END_PARAGRAPH
      Once you have finished those two programs (and any others you would like to
try),
      let's learn some more (and some more about) #{makeLink('methods',
:generateMethods)}.
      END_PARAGRAPH
    end
  end

  #
  # METHODS
  #

  def generateMethods
    para do <<-END_PARAGRAPH
      So far we've seen a number of different methods,
      #{code 'puts'} and #{code 'gets'}
      and so on (<em><strong>Pop Quiz:</strong>  List all
      of the methods we have seen so far!
      There are ten of them; the answer is below.</em>),
      but we haven't really talked about what methods are.
      We know what they do, but
      we don't know what they are.
      END_PARAGRAPH
```

```
end
para do <<-END_PARAGRAPH
  But really, that <em>is</em> what they are:  things
  that do stuff.  If objects (like strings,
  integers, and floats) are the nouns in the Ruby
  language, then methods are like the verbs.
  And, just like in English, you can't have a
  verb without a noun to <em>do</em> the verb.
  For example, ticking isn't something that just
  happens; a clock (or a watch or something)
  has to do it.  In English we would say, "The
  clock ticks."  In Ruby we would say
  #{code 'clock.tick'} (assuming that #{code 'clock'}
  was a Ruby object, of course).
  Programmers might say we were "calling #{code 'clock'}'s
  #{code 'tick'} method,"
  or that we "called #{code 'tick'} on #{code 'clock'}."
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  So, did you take the quiz?  Good.  Well, I'm
  sure you remembered the methods
  #{code 'puts'}, #{code 'gets'}, and #{code 'chomp'},
  since we just covered those.
  You probably also got our conversion methods,
  #{code 'to_i'}, #{code 'to_f'},
  and #{code 'to_s'}.  However, did you get
  the other four?  Why, it's none other
  than our old arithmetic buddies #{code '+'},
  #{code '-'}, #{code '*'}, and #{code '/'}!
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  So as I was saying, just as every verb needs
  a noun, so every method needs an object.
  It's usually easy to tell which object is
  performing the method:  it's what comes right
  before the dot, like in our #{code 'clock.tick'}
  example, or in #{code '101.to_s'}.
  Sometimes, however, it's not quite as
  obvious; like with the arithmetic methods.  As
  it turns out, #{code '5 + 5'} is really
  just a shortcut way of writing #{code '5.+ 5'}.
  For example:
  END_PARAGRAPH
```

```
end
prog do <<-END_CODE
  puts 'hello '.+ 'world'
  puts (10.* 9).+ 9
  END_CODE
end
para do <<-END_PARAGRAPH
  It isn't very pretty, so we won't ever write
  it like that; however, it's important to
  understand what is <em>really</em> happening.
  (On older versions of Ruby, this code might also give a <dfn>warning</dfn>:
  #{output 'warning: parenthesize argument(s) for future version'}.
  It would still run the code just fine, though.)
  This also gives us a deeper understanding
  of why we can do #{code "'pig'*5"} but we
  can't do #{code "5*'pig'"}:  #{code "'pig'*5"} is
  telling #{code "'pig'"} to do the multiplying,
  but #{code "5*'pig'"} is telling #{code '5'}
  to do the multiplying.  #{code "'pig'"} knows how
  to make #{code '5'} copies of itself and
  add them all together; however, #{code '5'}
  will have a much more difficult time of making
  #{code "'pig'"} copies of <em>itself</em>
  and adding them together.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  And, of course, we still have #{code 'puts'}
  and #{code 'gets'} to explain.  Where are their
  objects?  In English, you can sometimes leave
  out the noun; for example, if a villain
  yells "Die!", the implicit noun is whoever
  he is yelling at.  In Ruby, if I say
  #{code "puts 'to be or not to be'"}, what
  I am really saying is
  #{code "self.puts 'to be or not to be'"}.
  So what is #{code 'self'}?  It's a special variable
  which points to whatever object you are in.
  We don't even know how to be <em>in</em>
  an object yet, but until we find out, we
  are always going to be in a big object which
  is... the whole program!  And lucky for us,
  the program has a few methods of its own,
  like #{code 'puts'} and #{code 'gets'}.
  Watch this:
```

```
  END_PARAGRAPH
end
prog do <<-END_CODE
  iCantBelieveIMadeAVariableNameThisLongJustToPointToA3 = 3
  puts iCantBelieveIMadeAVariableNameThisLongJustToPointToA3
  self.puts iCantBelieveIMadeAVariableNameThisLongJustToPointToA3
  END_CODE
end
para do <<-END_PARAGRAPH
  If you didn't entirely follow all of that,
  that's OK.  The important thing to take away from
  all of this is that every method is being
  done by some object, even if it doesn't have
  a dot in front of it.  If you understand
  that, then you're all set.
  END_PARAGRAPH
end
h2 {'Fancy String Methods'}
para do <<-END_PARAGRAPH
  Let's learn a few fun string methods.  You don't
  have to memorize them all; you can
  just look up this page again if you forget
  them.  I just want to show you a <em>small</em>
  part of what strings can do.  In fact, I
  can't remember even half of the string methods myself&mdash;but
  that's fine, because there are great references
  on the internet with all of the string
  methods listed and explained.  (I will show
  you where to find them at the end of this tutorial.)
  Really, I don't even <em>want</em> to know
  all the string methods; it's kind of like knowing every
  word in the dictionary.  I can speak English
  just fine without knowing every word in
  the dictionary... and isn't that really the whole
  point of the dictionary?  So you don't <em>have</em>
  to know what's in it?
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  So, our first string method is #{code 'reverse'},
  which gives a backwards version of a string:
  END_PARAGRAPH
end
prog do <<-END_CODE
  var1 = 'stop'
```

```
      var2 = 'stressed'
      var3 = 'Can you pronounce this sentence backwards?'

      puts var1.reverse
      puts var2.reverse
      puts var3.reverse
      puts var1
      puts var2
      puts var3
      END_CODE
    end
    para do <<-END_PARAGRAPH
      As you can see, #{code 'reverse'} doesn't reverse the
      original string; it just makes
      a new backwards version of it.  That's why #{code 'var1'}
      is still #{code "'stop'"}
      even after we called #{code 'reverse'} on #{code 'var1'}.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Another string method is #{code 'length'}, which tells
      us the number of characters (including
      spaces) in the string:
      END_PARAGRAPH
    end
    prog ['Christopher David Pine'] do <<-END_CODE
      puts 'What is your full name?'
      name = gets.chomp
      puts 'Did you know there are ' + name.length +
           ' characters in your name, ' + name + '?'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Uh-oh!  Something went wrong, and it looks like it happened sometime after the
line
      #{code 'name = gets.chomp'}...  Do you see the problem?  See if you can figure
it out.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      The problem is with #{code 'length'}:  it gives us a number, but we want a
string.  Easy enough,
      we'll just throw in a #{code 'to_s'} (and cross our fingers):
      END_PARAGRAPH
    end
```

```
prog ['Christopher David Pine'] do <<-END_CODE
  puts 'What is your full name?'
  name = gets.chomp
  puts 'Did you know there are ' + name.length.to_s +
       ' characters in your name, ' + name + '?'
  END_CODE
end
para do <<-END_PARAGRAPH
  No, I did not know that.  <strong>Note:</strong>  that's the number of
  <em>characters</em> in my name, not the number of <em>letters</em>
  (count 'em).  I guess we could write a program which
  asks for your first, middle, and last names individually, and then
  adds those lengths together... hey, why don't you do that!  Go ahead,
  I'll wait.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Did you do it?  Good!  It's nice to program, isn't it?
  After a few more chapters, though, you'll be amazed at
  what you can do.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  So, there are also a number of string methods which change the case
  (uppercase and lowercase) of your string.  #{code 'upcase'} changes
  every lowercase letter to uppercase, and #{code 'downcase'} changes
  every uppercase letter to lowercase.  #{code 'swapcase'} switches
  the case of every letter in the string, and finally, #{code 'capitalize'}
  is just like #{code 'downcase'}, except that it switches the first
  character to uppercase (if it is a letter).
  END_PARAGRAPH
end
prog do <<-END_CODE
  letters = 'aAbBcCdDeE'
  puts letters.upcase
  puts letters.downcase
  puts letters.swapcase
  puts letters.capitalize
  puts ' a'.capitalize
  puts letters
  END_CODE
end
para do <<-END_PARAGRAPH
  Pretty standard stuff.  As you can see from the line
  #{code "puts ' a'.capitalize"}, the method #{code 'capitalize'}
```

```
  only capitalizes the first <em>character</em>, not the first
  <em>letter</em>.  Also, as we have seen before, throughout all of
  these method calls, #{code 'letters'} remains unchanged.  I don't mean
  to belabor the point, but it's important to understand.  There are
  some methods which <em>do</em> change the associated object, but we haven't
  seen any yet, and we won't for some time.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  The last of the fancy string methods we'll look at
  are for visual formatting.
  The first one, #{code 'center'}, adds spaces to the beginning and
  end of the string to make it centered.  However, just like you have
  to tell #{code 'puts'} what you want it to print, and #{code '+'}
  what you want it to add, you have to tell #{code 'center'} how
  wide you want your centered string to be.  So if I wanted to center
  the lines of a poem, I would do it like this:
  END_PARAGRAPH
end
prog do <<-END_CODE
  lineWidth = 50
  puts(                'Old Mother Hubbard'.center(lineWidth))
  puts(               'Sat in her cupboard'.center(lineWidth))
  puts(         'Eating her curds an whey,'.center(lineWidth))
  puts(          'When along came a spider'.center(lineWidth))
  puts(         'Which sat down beside her'.center(lineWidth))
  puts('And scared her poor shoe dog away.'.center(lineWidth))
  END_CODE
end
para do <<-END_PARAGRAPH
  Hmmm... I don't think that's how that nursery rhyme goes, but I'm
  too lazy to look it up.  (Also, I wanted to line up the
  #{code '.center lineWidth'} part, so I put in those extra spaces
  before the strings.  This is just because I think it is prettier
  that way.  Programmers often have strong feelings about what is pretty
  in a program, and they often disagree about it.  The more you
  program, the more you will come into your own style.)  Speaking of
  being lazy, laziness isn't always
  a bad thing in programming.  For example, see how I stored the
  width of the poem in the variable #{code 'lineWidth'}?  This was so that
  if I want to go back later and make the poem wider, I only have to
  change the very top line of the program, instead of every line which
  does centering.  With a very long poem, this could save me a lot of
  time.  That kind of laziness is really a virtue in programming.
  END_PARAGRAPH
```

```
    end
    para do <<-END_PARAGRAPH
      So, about that centering... you may have noticed that it isn't quite
      as beautiful as what a word processor would have done.  If you really
      want perfect centering (and maybe a nicer font), then you should just use
      a word processor!  Ruby is a wonderful tool, but no tool is the right
      tool for <em>every</em> job.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      The other two string formatting methods are #{code 'ljust'} and
      #{code 'rjust'}, which stand for <dfn>left justify</dfn> and
      <dfn>right justify</dfn>.  They are similar to #{code 'center'}, except
      that they pad the string with spaces on the right and left sides,
      respectively.  Let's take a look at all three in action:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      lineWidth = 40
      str = '--> text <--'
      puts str.ljust  lineWidth
      puts str.center lineWidth
      puts str.rjust  lineWidth
      puts str.ljust(lineWidth/2) + str.rjust(lineWidth/2)
      END_CODE
    end
    h2 {'A Few Things to Try'}
    ul do
      li {"Write an Angry Boss program.  It should rudely ask what you want.
      Whatever you answer, the Angry Boss should yell it back to you, and
      then fire you.  For example, if you type in #{input 'I want a raise.'}, it
should yell back
      #{output 'WHADDAYA MEAN "I WANT A RAISE."?!?  YOU\'RE FIRED!!'}"}

      li do
        para do <<-END_PARAGRAPH
          So here's something for you to do in order to play around more with
          #{code 'center'}, #{code 'ljust'}, and #{code 'rjust'}:  Write a program
          which will display a Table of Contents so that it looks like this:
          END_PARAGRAPH
        end
        puts '<pre class="L2PoutputBlock">' +
             '                Table of Contents                ' + $/ +
             '                                                 ' + $/ +
             'Chapter 1:  Numbers                      page 1' + $/ +
```

```
          'Chapter 2:  Letters                      page 72' + $/ +
          'Chapter 3:  Variables                   page 118' + $/ +
          '</pre>'
  end
end

h2 {'Higher Math'}

para do <<-END_PARAGRAPH
  <em>(This section is totally optional.  It assumes a fair degree
  of mathematical knowledge.  If you aren't interested, you
  can go straight to #{makeLink 'Flow Control', :generateFlowControl}
  without any problems.  However, a quick look at the section
  on <strong>Random Numbers</strong> might come in handy.)</em>
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  There aren't nearly as many number methods as there are string methods
  (though I still don't know them all off the top of my head).  Here, we'll
  look at the rest of the arithmetic methods, a random number generator,
  and the #{code 'Math'} object, with its trigonometric and transcendental
  methods.
  END_PARAGRAPH
end
h2 {'More Arithmetic'}
para do <<-END_PARAGRAPH
  The other two arithmetic methods are #{code '**'} (exponentiation)
  and #{code '%'} (modulus).  So if you want to say "five squared"
  in Ruby, you would write it as #{code '5**2'}.  You can also use
  floats for your exponent, so if you want the square root of 5, you
  could write #{code '5**0.5'}.  The modulus method gives you the remainder
  after division by a number.  So, for example, if I divide 7 by 3,
  I get 2 with a remainder of 1.  Let's see it working in a program:
  END_PARAGRAPH
end
prog do <<-END_CODE
  puts 5**2
  puts 5**0.5
  puts 7/3
  puts 7%3
  puts 365%7
  END_CODE
end
para do <<-END_PARAGRAPH
  From that last line, we learn that a (non-leap) year has some number
```

```
      of weeks, plus one day.  So if your birthday was on a Tuesday this year,
      it will be on a Wednesday next year.  You can also use floats with the modulus
      method.  Basically, it works the only sensible way it could... but I'll
      let you play around with that.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      There's one last method to mention before we check out the random number
      generator:  #{code 'abs'}.  It just takes the absolute value of the number:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts((5-2).abs)
      puts((2-5).abs)
      END_CODE
    end
    h2 {'Random Numbers'}
    para do <<-END_PARAGRAPH
      Ruby comes with a pretty nice random number generator.  The method to get
      a randomly chosen number is #{code 'rand'}.  If you call #{code 'rand'} just
like
      that, you'll get a float greater than or equal to #{code '0.0'} and less
      than #{code '1.0'}.  If you give #{code 'rand'} an integer (#{code '5'}
      for example), it will give you an integer greater than or equal to
      #{code '0'} and less than #{code '5'} (so five possible numbers,
      from #{code '0'} to #{code '4'}).
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Let's see #{code 'rand'} in action.
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts rand
      puts rand
      puts rand
      puts(rand(100))
      puts(rand(100))
      puts(rand(100))
      puts(rand(1))
      puts(rand(1))
      puts(rand(1))
      puts(rand(99999999999999999999999999999999999999999999999999999999))
      puts('The weatherman said there is a '+rand(101).to_s+'% chance of rain,')
      puts('but you can never trust a weatherman.')
```

```
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Note that I used #{code 'rand(101)'} to get back numbers from #{code '0'}
      to #{code '100'}, and that #{code 'rand(1)'} always
      gives back #{code '0'}.  Not understanding the range of possible return
      values is the biggest mistake I see people make with #{code 'rand'}; even
professional
      programmers; even in finished products you can buy at the store.  I even
      had a CD player once which, if set on "Random Play," would play every song but
      the last one...  (I wonder what would have happened if I had put in a CD with
      only one song on it?)
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Sometimes you might want #{code 'rand'} to return the <em>same</em> random
numbers
      in the same sequence on two different runs of your program.  (For example,
once I
      was using randomly generated numbers to create a randomly generated world for
a computer
      game.  If I found a world that I really liked, perhaps I would want to play on
it
      again, or send it to a friend.)  In order to do this, you need to set the
      <em>seed</em>, which you can do with #{code 'srand'}.  Like this:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      srand 1776
      puts(rand(100))
      puts(rand(100))
      puts(rand(100))
      puts(rand(100))
      puts(rand(100))
      puts ''
      srand 1776
      puts(rand(100))
      puts(rand(100))
      puts(rand(100))
      puts(rand(100))
      puts(rand(100))
      END_CODE
    end
    para do <<-END_PARAGRAPH
      It will do the same thing every time you seed it with the same number.  If you
```

```
want
      to get different numbers again (like what happens if you never use
      #{code 'srand'}), then just call #{code 'srand 0'}.  This seeds it with a
      really weird number, using (among other things) the current time on
      your computer, down to the millisecond.
      END_PARAGRAPH
    end
    h2 {"The #{code 'Math'} Object"}
    para do <<-END_PARAGRAPH
      Finally, let's look at the #{code 'Math'} object.  We might as well
      jump right in:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts(Math::PI)
      puts(Math::E)
      puts(Math.cos(Math::PI/3))
      puts(Math.tan(Math::PI/4))
      puts(Math.log(Math::E**2))
      puts((1 + Math.sqrt(5))/2)
      END_CODE
    end
    para do <<-END_PARAGRAPH
      The first thing you noticed was probably the #{code '::'}
      notation.  Explaining the <dfn>scope operator</dfn> (which is what that is)
      is really beyond the, uh... scope of this tutorial.  No pun
      intended.  I swear.  Suffice it to say, you can use
      #{code 'Math::PI'} just like you would expect to.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      As you can see, #{code 'Math'} has all of the things you would
      expect a decent scientific calculator to have.  And as always,
      the floats are <em>really close</em> to being the right answers.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      So now let's #{makeLink 'flow', :generateFlowControl}!
      END_PARAGRAPH
    end
  end

  #
  # FLOW CONTROL
  #
```

```
  def generateFlowControl
    para do <<-END_PARAGRAPH
      Ahhhh, flow control.  This is where it all comes together.  Even though
      this chapter is shorter and easier than the #{makeLink 'methods',
:generateMethods}
      chapter, it will open up a whole world of programming possibilities.
      After this chapter, we'll be able to write truly interactive
      programs; in the past we have made programs which <em>say</em> different
      things depending on your keyboard input, but after this chapter
      they will actually <em>do</em> different things, too.  But
      before we can do that, we need to be
      able to compare the objects in our programs.  We need...
      END_PARAGRAPH
    end
    h2 {'Comparison Methods'}
    para do <<-END_PARAGRAPH
      Let's rush through this part so we can get to the next
      section, <strong>Branching</strong>, where all the cool
      stuff happens.  So, to see if one object is greater than
      or less than another, we use the methods #{code '>'}
      and #{code '<'}, like this:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts 1 > 2
      puts 1 < 2
      END_CODE
    end
    para do <<-END_PARAGRAPH
      No problem.  Likewise, we can find out if an object is
      greater-than-or-equal-to another (or less-than-or-equal-to)
      with the methods #{code '>='} and #{code '<='}
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts 5 >= 5
      puts 5 <= 4
      END_CODE
    end
    para do <<-END_PARAGRAPH
      And finally, we can see if two objects are equal or not
      using #{code '=='} (which means "are these equal?")
      and #{code '!='} (which means "are these different?").
      It's important not to confuse #{code '='} with #{code '=='}.
```

```
      #{code '='} is for telling a variable to point at an object
      (assignment), and #{code '=='} is for asking the question:  "Are
      these two objects equal?"
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts 1 == 1
      puts 2 != 1
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Of course, we can compare strings, too.  When strings
      get compared, they compare their <dfn>lexicographical ordering</dfn>,
      which basically means their dictionary ordering.  #{code 'cat'}
      comes before #{code 'dog'} in the dictionary, so:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      puts 'cat' < 'dog'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      There's a catch, though:  the way computers usually do things,
      they order capital letters as coming before lowercase letters.
      (That's how they store the letters in fonts, for example:
      all the capital letters first, then the lowercase ones.)
      This means that it will think #{code "'Zoo'"} comes before #{code "'ant'"}, so
if you
      want to figure out which word would come first in a real dictionary,
      make sure to use #{code 'downcase'} (or #{code 'upcase'} or
      #{code 'capitalize'}) on both words before you try to compare them.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      One last note before <strong>Branching</strong>:  The comparison
      methods aren't giving us the strings #{code "'true'"} and
      #{code "'false'"}; they are giving us the special objects #{code 'true'} and
      #{code 'false'}.  (Of course, #{code 'true.to_s'} gives us
      #{code "'true'"}, which is why #{code 'puts'} printed #{code "'true'"}.)
      #{code 'true'} and #{code 'false'} are used all the time in...
      END_PARAGRAPH
    end
    h2 {'Branching'}
    para do <<-END_PARAGRAPH
      Branching is a simple concept, but powerful.  In fact, it's so simple
```

```
      that I bet I don't even have to explain it at all; I'll just show you:
      END_PARAGRAPH
    end
    run1 = {:input => ['Chris']}
    run2 = {:input => ['Chewbacca'], :remark => 'But if we put in a different
name...'}
    progN run1, run2 do <<-END_CODE
      puts 'Hello, what\\'s your name?'
      name = gets.chomp
      puts 'Hello, ' + name + '.'
      if name == 'Chris'
        puts 'What a lovely name!'
      end
      END_CODE
    end
    para do <<-END_PARAGRAPH
      And that is branching.  If what comes after the #{code 'if'} is
      #{code 'true'}, we run the code between the
      #{code 'if'} and the #{code 'end'}.  If what comes after the
      #{code 'if'} is #{code 'false'}, we don't.  Plain and simple.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      I indented the code between the #{code 'if'} and the #{code 'end'}
      just because I think it's easier to keep track of the
      branching that way.  Almost all
      programmers do this, regardless of what language they are
      programming in.  It may not seem much help in this simple
      example, but when things get more complex, it makes a big
      difference.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Often, we would like a program to do one thing if an expression
      is #{code 'true'}, and another if it is #{code 'false'}.  That's
      what #{code 'else'} is for:
      END_PARAGRAPH
    end
    run1 = {:input => ['Chris']}
    run2 = {:input => ['Ringo'], :remark => 'Now let\'s try a different name...'}
    progN run1, run2 do <<-END_CODE
      puts 'I am a fortune-teller.  Tell me your name:'
      name = gets.chomp
      if name == 'Chris'
        puts 'I see great things in your future.'
```

```
  else
    puts 'Your future is... Oh my!  Look at the time!'
    puts 'I really have to go, sorry!'
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  Branching is kind of like coming to a fork in the code:  Do
  we take the path for people whose #{code "name == 'Chris'"},
  or #{code 'else'} do we take the other path?
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  And just like the branches of a tree, you can have branches
  which themselves have branches:
  END_PARAGRAPH
end
run1 = {:input => ['chris', 'yes']}
run2 = {:input => ['Chris'], :remark => 'Fine, I\'ll capitalize it...'}
progN run1, run2 do <<-END_CODE
  puts 'Hello, and welcome to 7th grade English.'
  puts 'My name is Mrs. Gabbard.  And your name is...?'
  name = gets.chomp

  if name == name.capitalize
    puts 'Please take a seat, ' + name + '.'
  else
    puts name + '?  You mean ' + name.capitalize + ', right?'
    puts 'Don\\'t you even know how to spell your name??'
    reply = gets.chomp

    if reply.downcase == 'yes'
      puts 'Hmmph!  Well, sit down!'
    else
      puts 'GET OUT!!'
    end
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  Sometimes it might get confusing trying to figure out
  where all of the #{code 'if'}s, #{code 'else'}s, and
  #{code 'end'}s go.  What I do is write the #{code 'end'}
  <em>at the same time</em> I write the #{code 'if'}.  So
  as I was writing the above program, this is how it looked
```

```
  first:
  END_PARAGRAPH
end
prog false do <<-END_CODE
  puts 'Hello, and welcome to 7th grade English.'
  puts 'My name is Mrs. Gabbard.  And your name is...?'
  name = gets.chomp

  if name == name.capitalize
  else
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  Then I filled it in with <dfn>comments</dfn>, stuff
  in the code the computer will ignore:
  END_PARAGRAPH
end
prog false do <<-END_CODE
  puts 'Hello, and welcome to 7th grade English.'
  puts 'My name is Mrs. Gabbard.  And your name is...?'
  name = gets.chomp

  if name == name.capitalize
    # She's civil.
  else
    # She gets mad.
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  Anything after a #{code '#'} is
  considered a comment (unless, of course, you
  are in a string).  After that, I replaced the comments
  with working code.  Some people like to leave the comments
  in; personally, I think well-written code usually speaks
  for itself.  I used to use more comments, but the more
  "fluent" in Ruby I become, the less I use them.  I actually
  find them distracting much of the time.  It's a personal
  choice; you'll find your own (usually evolving) style.
  So my next step looked like this:
  END_PARAGRAPH
end
prog false do <<-END_CODE
  puts 'Hello, and welcome to 7th grade English.'
```

```
  puts 'My name is Mrs. Gabbard.  And your name is...?'
  name = gets.chomp

  if name == name.capitalize
    puts 'Please take a seat, ' + name + '.'
  else
    puts name + '?  You mean ' + name.capitalize + ', right?'
    puts 'Don\\'t you even know how to spell your name??'
    reply = gets.chomp

    if reply.downcase == 'yes'
    else
    end
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  Again, I wrote down the #{code 'if'}, #{code 'else'}, and
  #{code 'end'} all at the same time.  It really helps me keep
  track of "where I am" in the code.  It also makes the job
  seem easier because I can focus on one small part, like
  filling in the code between the #{code 'if'} and the
  #{code 'else'}.  The other benefit of doing it this way
  is that the computer can understand the program at any
  stage.  Every one of the unfinished versions of the
  program I showed you would run.  They weren't finished,
  but they were working programs.  That way I could test it
  as I wrote it, which helped to see how it was coming along
  and where it still needed work.  When it passed all
  of the tests, that's how I knew I was done!
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  These tips will help you write programs with branching,
  but they also help with the other main type of flow control:
  END_PARAGRAPH
end
h2 {'Looping'}
para do <<-END_PARAGRAPH
  Often, you'll want your computer to do the same thing over and
  over again&mdash;after all, that's what computers are supposed to
  be so good at.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
```

```
      When you tell your computer to keep repeating something,
      you also need to tell it when to stop.  Computers never get bored,
      so if you don't tell it to stop, it won't.  We make sure this
      doesn't happen by telling the computer to repeat certain parts
      of a program #{code 'while'} a certain condition is true.  This
      works very similarly to how #{code 'if'} works:
      END_PARAGRAPH
    end
    prog ['Hello?', 'Hi!', 'Very nice to meet you.', 'Oh... how sweet!', 'bye'] do
<<-END_CODE
      command = ''

      while command != 'bye'
        puts command
        command = gets.chomp
      end

      puts 'Come again soon!'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      And that's a loop.  (You may have noticed the blank line at the
      beginning of the output; it's from the first #{code 'puts'}, before
      the first #{code 'gets'}.  How would you change the program to get
      rid of this first line.  Test it!  Did it work <em>exactly</em>
      like the program above, other than that first blank line?)
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Loops allow you to do all kinds of interesting things, as I'm sure
      you can imagine.  However, they can also cause problems if you
      make a mistake.  What if your computer gets trapped in an infinite
      loop?  If you think this may have happened, just hold down the
      <kbd>Ctrl</kbd> key and press <kbd>C</kbd>.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Before we start playing around with loops, though,
      let's learn a few things to make our job easier.
      END_PARAGRAPH
    end
    h2 {'A Little Bit of Logic'}
    para do <<-END_PARAGRAPH
      Let's take a look at our first branching program again.  What if
      my wife came home, saw the program, tried it out, and it didn't
```

```
  tell her what a lovely name <em>she</em> had?  Well... she probably
  wouldn't care. But <em>I'd</em> care! So let's rewrite it:
  END_PARAGRAPH
end
prog ['Katy'] do <<-END_CODE
  puts 'Hello, what\\'s your name?'
  name = gets.chomp
  puts 'Hello, ' + name + '.'
  if name == 'Chris'
    puts 'What a lovely name!'
  else
    if name == 'Katy'
      puts 'What a lovely name!'
    end
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  It works... but it isn't a very pretty program.  Why not?
  Well, the best
  rule I ever learned in programming was the <dfn>DRY</dfn> rule:
  <dfn>Don't Repeat Yourself</dfn>.  I could probably write a small
  book just on why that is such a good rule.  In our case, we
  repeated the line #{code "puts 'What a lovely name!'"}.  Why is
  this such a big deal?  Well, what if I made a spelling mistake
  when I rewrote it?  What if I wanted to change it from
  #{code "'lovely'"} to #{code "'beautiful'"} on both lines?
  I'm lazy, remember?  Basically, if
  I want the program to do the same thing when it gets
  #{code "'Chris'"} or #{code "'Katy'"}, then it should really
  <em>do the same thing</em>:
  END_PARAGRAPH
end
prog ['Katy'] do <<-END_CODE
  puts 'Hello, what\\'s your name?'
  name = gets.chomp
  puts 'Hello, ' + name + '.'
  if (name == 'Chris' or name == 'Katy')
    puts 'What a lovely name!'
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  Much better.  In order to make it work, I used #{code 'or'}.
  The other <em>logical operators</em> are #{code 'and'} and
```

```
  #{code 'not'}.  It is always a good idea to use parentheses
  when working with these.  Let's see how they work:
  END_PARAGRAPH
end
prog do <<-END_CODE
  iAmChris  = true
  iAmPurple = false
  iLikeFood = true
  iEatRocks = false

  puts (iAmChris  and iLikeFood)
  puts (iLikeFood and iEatRocks)
  puts (iAmPurple and iLikeFood)
  puts (iAmPurple and iEatRocks)
  puts
  puts (iAmChris  or iLikeFood)
  puts (iLikeFood or iEatRocks)
  puts (iAmPurple or iLikeFood)
  puts (iAmPurple or iEatRocks)
  puts
  puts (not iAmPurple)
  puts (not iAmChris )
  END_CODE
end
para do <<-END_PARAGRAPH
  The only one of these which might trick you is
  #{code 'or'}.  In English, we often use "or" to mean
  "one or the other, but not both."  For example, your
  mom might say, "For dessert, you can have pie or cake."
  She did <em>not</em> mean you could have them both!
  A computer, on the other hand, uses #{code 'or'} to mean "one or the other,
  or both."  (Another way of saying it is, "at least one of
  these is true.")  This is why computers are more fun than
  moms.
  END_PARAGRAPH
end

h2 {'A Few Things to Try'}

ul do
  li {'<em>"99 bottles of beer on the wall..."</em>  Write a program
       which prints out the lyrics to that beloved classic, that
       field-trip favorite: "99 Bottles of Beer on the Wall."'}
  li {"Write a Deaf Grandma program.  Whatever you say
       to grandma (whatever you type in), she should respond with
```

```
        #{output 'HUH?!  SPEAK UP, SONNY!'}, unless you shout it (type in
        all capitals).  If you shout, she can hear you (or at least
        she thinks so) and yells back, #{output 'NO, NOT SINCE 1938!'}  To
        make your program <em>really</em> believable, have grandma
        shout a different year each time; maybe any year at random
        between 1930 and 1950.  (This part is optional, and would be
        much easier if you read the section on Ruby's random number
        generator at the end of the #{makeLink 'methods', :generateMethods}
        chapter.)  You can't stop talking to grandma
        until you shout #{input 'BYE'}.<br />
        <em><strong>Hint:</strong>  Don't forget about
        </em>#{code 'chomp'}<em>!  </em>#{code "'BYE'"}<em>with an
        Enter is not the same as </em>#{code "'BYE'"}<em> without
        one!</em><br />
        <em><strong>Hint 2:</strong>  Try to think about what parts
        of your program should happen over and over again.  All
        of those should be in your </em>#{code 'while'}<em> loop.</em>"}
  li {"Extend your Deaf Grandma program:  What if grandma
        doesn't want you to leave?  When you shout #{input 'BYE'}, she
        could pretend not to hear you.  Change your previous
        program so that you have to shout #{input 'BYE'} three times
        <em>in a row</em>.  Make sure to test your program:
        if you shout #{input 'BYE'} three times, but not in a row, you
        should still be talking to grandma."}
  li {"Leap Years.  Write a program which will ask for
        a starting year and an ending year, and then #{code 'puts'}
        all of the leap years between them (and including them,
        if they are also leap years).  Leap years are years divisible
        by four (like 1984 and 2004).  However, years divisible
        by 100 are <em>not</em> leap years (such as 1800 and
        1900) <strong><em>unless</em></strong> they are divisible
        by 400 (like 1600 and 2000, which were in fact leap years).
        <em>(Yes, it's all pretty
        confusing, but not as confusing has having July in the
        middle of the winter, which is what would eventually
        happen.)</em>"}
end

para do <<-END_PARAGRAPH
  When you finish those, take a break!  You've learned a lot
  already.  Congratulations!  Are you surprised at the number
  of things you can tell a computer to do?  A few more chapters
  and you'll be able to program just about anything.  Seriously!
  Just look at all the things you can do now that you couldn't
  do without looping and branching.
```

```
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    Now let's learn about a new kind of
    object, one which keeps track of lists of other objects:
    #{makeLink 'arrays', :generateArrays}.
    END_PARAGRAPH
  end
end

#
# ARRAYS AND ITERATORS
#

def generateArrays
  para do <<-END_PARAGRAPH
    Let's write a program which asks us to type in as many words
    as we want (one word per line, continuing until we just press
    <kbd>Enter</kbd> on an empty line), and which then repeats
    the words back to us in alphabetical order.  OK?
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    So... first we'll&mdash;uh... um... hmmm...  Well, we could&mdash;er...
    um...
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    You know, I don't think we can do it.  We need a way to store
    an unknown amount of words, and how to keep track of them all
    together, so they don't get mixed up with other variables.  We
    need to put them in some sort of a list.  We need <dfn>arrays</dfn>.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    An array is just a list in your computer.  Every slot in
    the list acts like a variable:  you can see what object
    a particular slot points to, and you can make it point to a
    different object.  Let's take a look at some arrays:
    END_PARAGRAPH
  end
  prog false do <<-END_CODE
    []
    [5]
    ['Hello', 'Goodbye']
```

```
  flavor = 'vanilla'             # This is not an array, of course...
  [89.9, flavor, [true, false]]  # ...but this is.
  END_CODE
end
para do <<-END_PARAGRAPH
  So first we have an empty array, then an array holding
  a single number, then an array holding two strings.
  Next, we have a simple assignment; then an
  array holding three objects, the last
  of which is the array #{code '[true, false]'}.  Remember,
  variables aren't objects, so our last array is really
  pointing to float, a <em>string</em>, and an array.  Even if we
  were to set #{code 'flavor'} to
  point to something else, that wouldn't change the
  array.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  To help us find a particular object in an array, each
  slot is given an index number.  Programmers (and, incidentally,
  most mathematicians) start counting from zero, though,
  so the first slot in the array is slot zero.  Here's
  how we would reference the objects in an array:
  END_PARAGRAPH
end
prog do <<-END_CODE
  names = ['Ada', 'Belle', 'Chris']

  puts names
  puts names[0]
  puts names[1]
  puts names[2]
  puts names[3]  # This is out of range.
  END_CODE
end
para do <<-END_PARAGRAPH
  So, we see that #{code 'puts names'} prints each name in
  the array #{code 'names'}.  Then we use #{code 'puts names[0]'}
  to print out the "first" name in the array, and
  #{code 'puts names[1]'} to print the "second"... I'm sure this seems
  confusing, but you <em>do</em> get used to it.  You just have to really
  start <em>thinking</em> that counting begins at zero, and
  stop using words like "first" and "second".
  If you go out to a five-course meal, don't talk about
```

```
  the "first" course; talk about course zero
  (and in your head, be thinking #{code 'course[0]'}).
  You have five fingers on your right hand, and their
  numbers are 0, 1, 2, 3, and 4.  My wife and I are
  jugglers.  When we juggle six clubs, we are juggling
  clubs 0-5.  Hopefully in the next few months, we'll
  be able to juggle club 6 (and thus be juggling seven
  clubs between us).  You'll know you've got it when you
  start using the word "zeroth".  :-) Yes, it's a real
  word; ask any programmer or mathematician.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Finally, we tried #{code 'puts names[3]'}, just to see what
  would happen.  Were you expecting an error?  Sometimes when
  you ask a question, your question doesn't make sense (at
  least to your computer); that's when you get an error.
  Sometimes, however, you can ask a question and the answer
  is <em>nothing</em>.  What's in slot three?  Nothing.
  What is #{code 'names[3]'}?  #{code 'nil'}:  Ruby's way
  of saying "nothing".  #{code 'nil'} is a special object
  which basically means "not any other object." And when you
  #{code 'puts nil'}, it prints out nothing. (Just a new line.)
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  If all this funny numbering of array slots is getting to
  you, fear not!  Often, we can avoid them completely by
  using various array methods, like this one:
  END_PARAGRAPH
end
h2 {"The Method #{code 'each'}"}
para do <<-END_PARAGRAPH
  #{code 'each'} allows us to do something (whatever we
  want) to #{code 'each'} object the array points to.  So, if we
  want to say something nice about each language in the array
  below, we'd do this:
  END_PARAGRAPH
end
prog do <<-END_CODE
  languages = ['English', 'German', 'Ruby']

  languages.each do |lang|
    puts 'I love ' + lang + '!'
    puts 'Don\\'t you?'
```

```
      end

      puts 'And let\\'s hear it for C++!'
      puts '...'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      So what just happened?  Well, we were able to go through
      every object in the array without using any numbers, so
      that's definitely nice.  Translating into English, the above
      program reads something like:  For #{code 'each'} object
      in #{code 'languages'}, point the variable #{code 'lang'}
      to the object and then #{code 'do'} everything I tell you to,
      until you come to the #{code 'end'}.  (Just so you know,
      C++ is another programming language.  It's much harder to
      learn than Ruby; usually, a C++ program will be many times
      longer than a Ruby program which does the same thing.)
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      You might be thinking to yourself, "This is a lot like
      the loops we learned about earlier."  Yep, it's similar.
      One important difference is that the method #{code 'each'}
      is just that:  a method.  #{code 'while'} and #{code 'end'}
      (much like #{code 'do'}, #{code 'if'}, #{code 'else'}, and all the other
      <span class="L2Pcode"><span class="L2Pkeyword">#{@@KEYWORD_COLOR}</span>
</span>
      words) are not methods.  They are a fundamental part of the Ruby
      language, just like #{code '='} and parentheses; kind of
      like punctuation marks in English.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      But not #{code 'each'}; #{code 'each'} is just another
      array method.  Methods like #{code 'each'} which "act like"
      loops are often called <dfn>iterators</dfn>.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      One thing to notice about iterators is that they are
      always followed by #{code 'do'}...#{code 'end'}.
      #{code 'while'} and #{code 'if'} never had a #{code 'do'}
      near them; we only use #{code 'do'} with iterators.
      END_PARAGRAPH
    end
```

```
para do <<-END_PARAGRAPH
  Here's another cute little iterator, but it's not an
  array method... it's an integer method!
  END_PARAGRAPH
end
prog do <<-END_CODE
  3.times do
    puts 'Hip-Hip-Hooray!'
  end
  END_CODE
end
h2 {'More Array Methods'}
para do <<-END_PARAGRAPH
  So we've learned #{code 'each'},
  but there are many other array methods... almost as
  many as there are string methods!  In fact, some of
  them (like #{code 'length'}, #{code 'reverse'},
  #{code '+'}, and #{code '*'})
  work just like they do for strings, except that they
  operate on the slots of the array rather than the
  letters of the string.  Others, like #{code 'last'}
  and #{code 'join'}, are specific to arrays.  Still
  others, like #{code 'push'} and #{code 'pop'},
  actually change the array.  And just as with
  the string methods, you don't have to remember
  all of these, as long as you can remember where to
  find out about them (right here).
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  First, let's look at #{code 'to_s'} and #{code 'join'}.
  #{code 'join'} works much like #{code 'to_s'} does, except
  that it adds a string in between the array's objects.
  Let's take a look:
  END_PARAGRAPH
end
prog do <<-END_CODE
  foods = ['artichoke', 'brioche', 'caramel']

  puts foods
  puts
  puts foods.to_s
  puts
  puts foods.join(', ')
  puts
```

```
  puts foods.join('  :)  ') + '  8)'

  200.times do
    puts []
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  As you can see, #{code 'puts'} treats arrays differently
  from other objects:  it just calls #{code 'puts'} on each
  of the objects in the array.  That's why #{code 'puts'}ing
  an empty array 200 times doesn't do anything; the array doesn't
  point to anything, so there's nothing to #{code 'puts'}.  (Doing
  nothing 200 times is still doing nothing.)
  Try #{code 'puts'}ing an array containing other arrays;
  does it do what you expected?
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Also, did you notice that I left out the empty strings when
  I wanted to #{code 'puts'} a blank line?  It does the same
  thing.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Now let's take a look at #{code 'push'}, #{code 'pop'},
  and #{code 'last'}.  The methods #{code 'push'} and #{code 'pop'}
  are sort of opposites,
  like #{code '+'} and #{code '-'} are.  #{code 'push'} adds
  an object to the end of your array, and #{code 'pop'}
  removes the last object from the array (and tell you
  what it was).  #{code 'last'} is similar to #{code 'pop'}
  in that it tells you what's at the end of the array,
  except that it leaves the array alone.
  Again, #{code 'push'} and #{code 'pop'} <em>actually
  change the array</em>:
  END_PARAGRAPH
end
prog do <<-END_CODE
  favorites = []
  favorites.push 'raindrops on roses'
  favorites.push 'whiskey on kittens'

  puts favorites[0]
  puts favorites.last
```

```
    puts favorites.length

    puts favorites.pop
    puts favorites
    puts favorites.length
    END_CODE
  end

  h2 {'A Few Things to Try'}
  ul do
    li {"Write the program we talked about at the very beginning
         of this chapter.<br />
         <em><strong>Hint:</strong>  There's a lovely
         array method which will give you a sorted version of an
         array:  </em>#{code 'sort'}<em>.  Use it!</em>"}
    li {"Try writing the above program <em>without</em> using
         the #{code 'sort'} method.  A large part of programming is
         solving problems, so get all the practice you can!"}
    li {"Rewrite your Table of Contents program (from the chapter
         on #{makeLink 'methods', :generateMethods}).  Start the program
         with an array holding all of the information for your Table
         of Contents (chapter names, page numbers, etc.).  Then print
         out the information from the array in a beautifully formatted
         Table of Contents."}
  end

  para do <<-END_PARAGRAPH
    So far we have learned quite a number of different methods.
    Now it's time to learn how to
    #{makeLink 'make our own', :generateDefMethod}.
    END_PARAGRAPH
  end
end

#
# WRITING METHODS
#

def generateDefMethod
  para do <<-END_PARAGRAPH
    As we've seen, loops and iterators allow us to do the
    same thing (run the same code) over and over again.
    However, sometimes we want to do the same thing a
    number of times, but from different places in the program.
    For example, let's say we were writing a questionnaire
```

```
  program for a psychology student.  From the psychology
  students I have known and the questionnaires they have
  given me, it would probably go something like this:
  END_PARAGRAPH
end
prog ['yes','yes','no way!','NO','yes','yes'] do <<-END_CODE
  puts 'Hello, and thank you for taking the time to'
  puts 'help me with this experiment.  My experiment'
  puts 'has to do with the way people feel about'
  puts 'Mexican food.  Just think about Mexican food'
  puts 'and try to answer every question honestly,'
  puts 'with either a "yes" or a "no".  My experiment'
  puts 'has nothing to do with bed-wetting.'
  puts

  # We ask these questions, but we ignore their answers.

  goodAnswer = false
  while (not goodAnswer)
    puts 'Do you like eating tacos?'
    answer = gets.chomp.downcase
    if (answer == 'yes' or answer == 'no')
      goodAnswer = true
    else
      puts 'Please answer "yes" or "no".'
    end
  end

  goodAnswer = false
  while (not goodAnswer)
    puts 'Do you like eating burritos?'
    answer = gets.chomp.downcase
    if (answer == 'yes' or answer == 'no')
      goodAnswer = true
    else
      puts 'Please answer "yes" or "no".'
    end
  end

  # We pay attention to *this* answer, though.
  goodAnswer = false
  while (not goodAnswer)
    puts 'Do you wet the bed?'
    answer = gets.chomp.downcase
    if (answer == 'yes' or answer == 'no')
```

```
    goodAnswer = true
    if answer == 'yes'
      wetsBed = true
    else
      wetsBed = false
    end
  else
    puts 'Please answer "yes" or "no".'
  end
end

goodAnswer = false
while (not goodAnswer)
  puts 'Do you like eating chimichangas?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
  else
    puts 'Please answer "yes" or "no".'
  end
end

puts 'Just a few more questions...'

goodAnswer = false
while (not goodAnswer)
  puts 'Do you like eating sopapillas?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
  else
    puts 'Please answer "yes" or "no".'
  end
end

# Ask lots of other questions about Mexican food.

puts
puts 'DEBRIEFING:'
puts 'Thank you for taking the time to help with'
puts 'this experiment.  In fact, this experiment'
puts 'has nothing to do with Mexican food.  It is'
puts 'an experiment about bed-wetting.  The Mexican'
puts 'food was just there to catch you off guard'
puts 'in the hopes that you would answer more'
```

```
  puts 'honestly.  Thanks again.'
  puts
  puts wetsBed
  END_CODE
end
para do <<-END_PARAGRAPH
  That was a pretty long program, with lots of repetition.
  (All of the sections of code around the questions about Mexican food
  were identical, and the bed-wetting question was only
  slightly different.)
  Repetition is a bad thing.  Still, we can't make it into
  a big loop or iterator, because sometimes we have things
  we want to do between questions.  In situations like these,
  it's best to write a method.  Here's how:
  END_PARAGRAPH
end
prog do <<-END_CODE
  def sayMoo
    puts 'mooooooo...'
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  Uh... our program didn't #{code 'sayMoo'}.
  Why not?  Because we didn't tell it to.
  We told it <em>how</em> to #{code 'sayMoo'},
  but we never actually said to <em>do</em> it.
  Let's give it another shot:
  END_PARAGRAPH
end
prog do <<-END_CODE
  def sayMoo
    puts 'mooooooo...'
  end

  sayMoo
  sayMoo
  puts 'coin-coin'
  sayMoo
  sayMoo
  END_CODE
end
para do <<-END_PARAGRAPH
  Ahhh, much better.  (Just in case you don't speak
  French, that was a French duck in the middle of the
```

```
    program.  In France, ducks say <em>"coin-coin"</em>.)
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    So we <span class="L2Pkeyword">#{code 'def'}</span>ined
    the method #{code 'sayMoo'}.  (Method names, like
    variable names, start with a lowercase letter.
    There are a few exceptions, though, like #{code '+'}
    or #{code '=='}.)
    But don't methods always have to be associated with
    objects?  Well, yes they do, and in this case (as with
    #{code 'puts'} and #{code 'gets'}), the method is just
    associated with the object representing
    the whole program.  In the next chapter we'll see how to
    add methods to other objects.  But first...
    END_PARAGRAPH
  end
  h2 {'Method Parameters'}
  para do <<-END_PARAGRAPH
    You may have noticed that some methods (like
    #{code 'gets'}, #{code 'to_s'}, #{code 'reverse'}...)
    you can just call on an object.  However, other methods
    (like #{code '+'}, #{code '-'}, #{code 'puts'}...)
    take <dfn>parameters</dfn> to tell the object how to
    do the method.  For example, you wouldn't just say
    #{code '5+'}, right?  You're telling #{code '5'} to
    add, but you aren't telling it <em>what</em>
    to add.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    To add a parameter to #{code 'sayMoo'} (let's say, the
    number of moos), we would do this:
    END_PARAGRAPH
  end
  prog do <<-END_CODE
    def sayMoo numberOfMoos
      puts 'moooooo...'*numberOfMoos
    end

    sayMoo 3
    puts 'oink-oink'
    sayMoo  # This should give an error because the parameter is missing.
    END_CODE
  end
```

```
  para do <<-END_PARAGRAPH
    #{code 'numberOfMoos'} is a variable which points to
    the parameter passed in.  I'll say that again, but it's
    a little confusing:  #{code 'numberOfMoos'} is a variable
    which points to the parameter passed in.  So if I type in
    #{code 'sayMoo 3'}, then the parameter is #{code '3'},
    and the variable #{code 'numberOfMoos'} points to #{code '3'}.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    As you can see, the parameter is now <em>required</em>.
    After all, what is #{code 'sayMoo'} supposed to multiply
    #{code "'mooooooo...'"} by if you don't give it a
    parameter?  Your poor computer has no idea.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    If objects in Ruby are like nouns in English, and methods
    are like verbs, then you can think of parameters as
    adverbs (like with #{code 'sayMoo'}, where the parameter
    told us <em>how</em> to #{code 'sayMoo'}) or sometimes as
    direct objects (like with #{code 'puts'}, where the
    parameter is <em>what</em> gets #{code 'puts'}ed).
    END_PARAGRAPH
  end
  h2 {'Local Variables'}
  para do <<-END_PARAGRAPH
    In the following program, there are two variables:
    END_PARAGRAPH
  end
  prog do <<-END_CODE
    def doubleThis num
      numTimes2 = num*2
      puts num.to_s+' doubled is '+numTimes2.to_s
    end

    doubleThis 44
    END_CODE
  end
  para do <<-END_PARAGRAPH
    The variables are #{code 'num'} and #{code 'numTimes2'}.
    They both sit inside the method #{code 'doubleThis'}.
    These (and all of the variables you have seen
    so far) are <dfn>local variables</dfn>.  This means
    that they live inside the method, and they cannot leave.
```

```
  If you try, you will get an error:
  END_PARAGRAPH
end
prog do <<-END_CODE
  def doubleThis num
    numTimes2 = num*2
    puts num.to_s+' doubled is '+numTimes2.to_s
  end

  doubleThis 44
  puts numTimes2.to_s
  END_CODE
end
para do <<-END_PARAGRAPH
  Undefined local variable...  In fact, we <em>did</em>
  define that local variable, but it isn't local to where
  we tried to use it; it's local to the method.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  This might seem inconvenient, but it actually quite nice.
  While it does mean that you have no access to variables
  inside methods, it also means that they have no access
  to <em>your</em> variables, and thus can't screw them up:
  END_PARAGRAPH
end
prog do <<-END_CODE
  def littlePest var
    var = nil
    puts 'HAHA!  I ruined your variable!'
  end

  var = 'You can\\'t even touch my variable!'
  littlePest var
  puts var
  END_CODE
end
para do <<-END_PARAGRAPH
  There are actually <em>two</em> variables in that little
  program named #{code 'var'}:  one inside #{code 'littlePest'},
  and one outside of it.  When we called #{code 'littlePest var'},
  we really just passed the string from one #{code 'var'} to
  the other, so that both were pointing to the same string.
  Then #{code 'littlePest'} pointed its own <em>local</em>
  #{code 'var'} to #{code 'nil'}, but that did nothing to the
```

```
    #{code 'var'} outside the method.
    END_PARAGRAPH
  end
  h2 {'Return Values'}
  para do <<-END_PARAGRAPH
    You may have noticed that some methods give you something
    back when you call them.  For example, #{code 'gets'}
    <dfn>returns</dfn> a string (the string you typed in),
    and the #{code '+'} method in #{code '5+3'}, (which is
    really #{code '5.+(3)'}) returns #{code '8'}.  The
    arithmetic methods for numbers return numbers, and the
    arithmetic methods for strings return strings.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    It's important to understand the difference between methods
    returning a value to where the method was called, and
    your program outputting information to your screen, like
    #{code 'puts'} does.  Notice that #{code '5+3'} returns
    #{code '8'}; it does <strong>not</strong> output
    #{output '8'}.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    So what <em>does</em> #{code 'puts'} return?  We never cared
    before, but let's look at it now:
    END_PARAGRAPH
  end
  prog do <<-END_CODE
    returnVal = puts 'This puts returned:'
    puts returnVal
    END_CODE
  end
  para do <<-END_PARAGRAPH
    The first #{code 'puts'} didn't seem to return anything,
    and in a way it didn't; it returned #{code 'nil'}.  Though
    we didn't test it, the second #{code 'puts'} did, too;
    #{code 'puts'} always returns #{code 'nil'}.  Every method
    has to return something, even if it's just #{code 'nil'}.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    Take a quick break and write a program
    to find out what #{code 'sayMoo'} returned.
    END_PARAGRAPH
```

```
end
para do <<-END_PARAGRAPH
  Were you surprised?  Well, here's how it works:  the value
  returned from a method is simply the last line of the method.  In
  the case of #{code 'sayMoo'}, this means it returns
  #{code "puts 'moooooo...'*numberOfMoos"}, which is just
  #{code 'nil'} since #{code 'puts'} always returns
  #{code 'nil'}.  If we wanted all of our methods to
  return the string #{code "'yellow submarine'"}, we
  would just need to put <em>that</em> at the end of
  them:
  END_PARAGRAPH
end
prog do <<-END_CODE
  def sayMoo numberOfMoos
    puts 'moooooo...'*numberOfMoos
    'yellow submarine'
  end

  x = sayMoo 2
  puts x
  END_CODE
end
para do <<-END_PARAGRAPH
  So, let's try that psychology experiment again,
  but this time we'll write a method to ask the questions for us.
  It will need to take the question as a parameter, and return
  #{code 'true'} if they answered #{input 'yes'} and
  #{code 'false'} if they answered #{input 'no'}.  (Even though
  most of the time we just ignore the answer, it's still a
  good idea for our method to return the answer.  This way we
  can use it for the bed-wetting question, too.)
  I'm also going to shorten the greeting and the debriefing,
  just so this is easier to read:
  END_PARAGRAPH
end
prog ['yes','yes','no way!','NO','yes','yes','yes','yes','yes'] do <<-END_CODE
  def ask question
    goodAnswer = false
    while (not goodAnswer)
      puts question
      reply = gets.chomp.downcase

      if (reply == 'yes' or reply == 'no')
        goodAnswer = true
```

```
        if reply == 'yes'
          answer = true
        else
          answer = false
        end
      else
        puts 'Please answer "yes" or "no".'
      end
    end

    answer  # This is what we return (true or false).
  end

  puts 'Hello, and thank you for...'
  puts

  ask 'Do you like eating tacos?'      # We ignore this return value.
  ask 'Do you like eating burritos?'
  wetsBed = ask 'Do you wet the bed?'  # We save this return value.
  ask 'Do you like eating chimichangas?'
  ask 'Do you like eating sopapillas?'
  ask 'Do you like eating tamales?'
  puts 'Just a few more questions...'
  ask 'Do you like drinking horchata?'
  ask 'Do you like eating flautas?'

  puts
  puts 'DEBRIEFING:'
  puts 'Thank you for...'
  puts
  puts wetsBed
  END_CODE
end
para do <<-END_PARAGRAPH
  Not bad, huh?  We were able to add more questions (and
  adding questions is <em>easy</em> now), but our program
  is still quite a bit shorter!  It's a big improvement
  &mdash; a lazy programmer's dream.
  END_PARAGRAPH
end
h2 {'One More Big Example'}
para do <<-END_PARAGRAPH
  I think another example method would be helpful here.
  We'll call this one #{code 'englishNumber'}.
  It will take a number, like #{code '22'},
```

```
  and return the english version of it (in this case,
  the string #{code "'twenty-two'"}).  For now, let's have it
  only work on integers from #{code '0'} to #{code '100'}.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  <em>(<strong>NOTE:</strong>  This method uses a new trick
  to return from a method early using the </em>#{code 'return'}<em>
  keyword, and introduces a new twist on branching:
  </em>#{code 'elsif'}<em>.  It should be clear in context
  how these work.)</em>
  END_PARAGRAPH
end
prog do <<-END_CODE
  def englishNumber number
    # We only want numbers from 0-100.
    if number < 0
      return 'Please enter a number zero or greater.'
    end
    if number > 100
      return 'Please enter a number 100 or lesser.'
    end

    numString = ''  # This is the string we will return.

    # "left" is how much of the number we still have left to write out.
    # "write" is the part we are writing out right now.
    # write and left... get it?  :)
    left  = number
    write = left/100          # How many hundreds left to write out?
    left  = left - write*100  # Subtract off those hundreds.

    if write > 0
      return 'one hundred'
    end

    write = left/10          # How many tens left to write out?
    left  = left - write*10  # Subtract off those tens.

    if write > 0
      if write == 1  # Uh-oh...
        # Since we can't write "tenty-two" instead of "twelve",
        # we have to make a special exception for these.
        if    left == 0
          numString = numString + 'ten'
```

```
      elsif left == 1
        numString = numString + 'eleven'
      elsif left == 2
        numString = numString + 'twelve'
      elsif left == 3
        numString = numString + 'thirteen'
      elsif left == 4
        numString = numString + 'fourteen'
      elsif left == 5
        numString = numString + 'fifteen'
      elsif left == 6
        numString = numString + 'sixteen'
      elsif left == 7
        numString = numString + 'seventeen'
      elsif left == 8
        numString = numString + 'eighteen'
      elsif left == 9
        numString = numString + 'nineteen'
      end
      # Since we took care of the digit in the ones place already,
      # we have nothing left to write.
      left = 0
    elsif write == 2
      numString = numString + 'twenty'
    elsif write == 3
      numString = numString + 'thirty'
    elsif write == 4
      numString = numString + 'forty'
    elsif write == 5
      numString = numString + 'fifty'
    elsif write == 6
      numString = numString + 'sixty'
    elsif write == 7
      numString = numString + 'seventy'
    elsif write == 8
      numString = numString + 'eighty'
    elsif write == 9
      numString = numString + 'ninety'
    end

    if left > 0
      numString = numString + '-'
    end
  end
```

```
    write = left  # How many ones left to write out?
    left  = 0     # Subtract off those ones.

    if write > 0
      if    write == 1
        numString = numString + 'one'
      elsif write == 2
        numString = numString + 'two'
      elsif write == 3
        numString = numString + 'three'
      elsif write == 4
        numString = numString + 'four'
      elsif write == 5
        numString = numString + 'five'
      elsif write == 6
        numString = numString + 'six'
      elsif write == 7
        numString = numString + 'seven'
      elsif write == 8
        numString = numString + 'eight'
      elsif write == 9
        numString = numString + 'nine'
      end
    end

    if numString == ''
      # The only way "numString" could be empty is if
      # "number" is 0.
      return 'zero'
    end

    # If we got this far, then we had a number somewhere
    # in between 0 and 100, so we need to return "numString".
    numString
  end

  puts englishNumber(  0)
  puts englishNumber(  9)
  puts englishNumber( 10)
  puts englishNumber( 11)
  puts englishNumber( 17)
  puts englishNumber( 32)
  puts englishNumber( 88)
  puts englishNumber( 99)
  puts englishNumber(100)
```

```
  END_CODE
end
para do <<-END_PARAGRAPH
  Well, there are certainly a few things about this program
  I don't like.  First, it has too much repetition.  Second,
  it doesn't handle numbers greater than 100.  Third, there
  are too many special cases, too many #{code 'return'}s.
  Let's use some arrays and try to clean it up a bit:
  END_PARAGRAPH
end
prog do <<-END_CODE
  def englishNumber number
    if number < 0  # No negative numbers.
      return 'Please enter a number that isn\\'t negative.'
    end
    if number == 0
      return 'zero'
    end

    # No more special cases! No more returns!

    numString = ''  # This is the string we will return.

    onesPlace = ['one',     'two',       'three',    'four',     'five',
                 'six',     'seven',     'eight',    'nine']
    tensPlace = ['ten',     'twenty',    'thirty',   'forty',    'fifty',
                 'sixty',   'seventy',   'eighty',   'ninety']
    teenagers = ['eleven',  'twelve',    'thirteen', 'fourteen', 'fifteen',
                 'sixteen', 'seventeen', 'eighteen', 'nineteen']

    # "left" is how much of the number we still have left to write out.
    # "write" is the part we are writing out right now.
    # write and left... get it?  :)
    left  = number
    write = left/100          # How many hundreds left to write out?
    left  = left - write*100  # Subtract off those hundreds.

    if write > 0
      # Now here's a really sly trick:
      hundreds  = englishNumber write
      numString = numString + hundreds + ' hundred'
      # That's called "recursion". So what did I just do?
      # I told this method to call itself, but with "write" instead of
      # "number". Remember that "write" is (at the moment) the number of
      # hundreds we have to write out. After we add "hundreds" to
```

```
    # "numString", we add the string ' hundred' after it.
    # So, for example, if we originally called englishNumber with
    # 1999 (so "number" = 1999), then at this point "write" would
    # be 19, and "left" would be 99. The laziest thing to do at this
    # point is to have englishNumber write out the 'nineteen' for us,
    # then we write out ' hundred', and then the rest of
    # englishNumber writes out 'ninety-nine'.

    if left > 0
      # So we don't write 'two hundredfifty-one'...
      numString = numString + ' '
    end
  end

  write = left/10          # How many tens left to write out?
  left  = left - write*10  # Subtract off those tens.

  if write > 0
    if ((write == 1) and (left > 0))
      # Since we can't write "tenty-two" instead of "twelve",
      # we have to make a special exception for these.
      numString = numString + teenagers[left-1]
      # The "-1" is because teenagers[3] is 'fourteen', not 'thirteen'.

      # Since we took care of the digit in the ones place already,
      # we have nothing left to write.
      left = 0
    else
      numString = numString + tensPlace[write-1]
      # The "-1" is because tensPlace[3] is 'forty', not 'thirty'.
    end

    if left > 0
      # So we don't write 'sixtyfour'...
      numString = numString + '-'
    end
  end

  write = left  # How many ones left to write out?
  left  = 0     # Subtract off those ones.

  if write > 0
    numString = numString + onesPlace[write-1]
    # The "-1" is because onesPlace[3] is 'four', not 'three'.
  end
```

```
    # Now we just return "numString"...
    numString
  end

  puts englishNumber(  0)
  puts englishNumber(  9)
  puts englishNumber( 10)
  puts englishNumber( 11)
  puts englishNumber( 17)
  puts englishNumber( 32)
  puts englishNumber( 88)
  puts englishNumber( 99)
  puts englishNumber(100)
  puts englishNumber(101)
  puts englishNumber(234)
  puts englishNumber(3211)
  puts englishNumber(999999)
  puts englishNumber(1000000000000)
  END_CODE
end
para do <<-END_PARAGRAPH
  <em>Ahhhh....</em> That's much, much better.  The program is
  fairly dense, which is why I put in so many comments.  It
  even works for large numbers... though not quite as nicely
  as one would hope.  For example, I think #{code "'one trillion'"}
  would be a nicer return value for that last number, or even
  #{code "'one million million'"} (though all three are correct).
  In fact, you can do that right now...
  END_PARAGRAPH
end

h2 {'A Few Things to Try'}
ul do
  li {"Expand upon #{code 'englishNumber'}.  First, put in
       thousands.  So it should return #{code "'one thousand'"}
       instead of #{code "'ten hundred'"} and #{code "'ten thousand'"}
       instead of #{code "'one hundred hundred'"}."}
  li {"Expand upon #{code 'englishNumber'} some more.
       Now put in millions, so you get #{code "'one million'"}
       instead of #{code "'one thousand thousand'"}.  Then try adding
       billions and trillions.  How high can you go?"}
  li {"How about #{code 'weddingNumber'}?  It should
       work almost the same as #{code 'englishNumber'}, except
       that it should insert the word \"and\" all over the place,
```

```
          returning things like #{code "'nineteen hundred and seventy and two'"},
          or however wedding invitations are supposed to look.  I'd give you more
          examples, but I don't fully understand it myself.  You might
          need to contact a wedding coordinator to help you."}
      li {"<em>\"Ninety-nine bottles of beer...\"</em>
          Using #{code 'englishNumber'} and your old program, write out the
          lyrics to this song the <em>right</em> way this time.
          Punish your computer:  have it start at 9999.  (Don't pick
          a number too large, though, because writing all of that to
          the screen takes your computer quite a while.  A hundred
          thousand bottles of beer takes some time; and if you pick
          a million, you'll be punishing yourself as well!"}
    end

    para do <<-END_PARAGRAPH
      Congratulations!  At this point, you are a true
      programmer!  You have learned
      everything you need to build huge programs from scratch.
      If you have ideas for programs you would like to write
      for yourself, give them a shot!
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Of course, building everything from scratch can be a
      pretty slow process.  Why spend time writing code that
      someone else already wrote?
      Would you like your program to send some email?
      Would you like to save and load files on your computer?
      How about generating web pages for a tutorial where
      the code samples are all automatically tested?  ;) Ruby has many different
      #{makeLink 'kinds of objects', :generateClasses}
      we can use to help us write better programs faster.
      END_PARAGRAPH
    end
  end

  #
  # CLASSES
  #

  def generateClasses
    para do <<-END_PARAGRAPH
      So far we've seen several different kinds, or
      <dfn>classes</dfn>, of objects:
      strings, integers, floats, arrays, and a few special objects
```

```
  (#{code 'true'}, #{code 'false'}, and #{code 'nil'}) which
  we'll talk about later.
  In Ruby, these classes are always capitalized:  #{code 'String'},
  #{code 'Integer'}, #{code 'Float'}, #{code 'Array'}... etc.
  In general, if we want to create a new object of a
  certain class, we use #{code 'new'}:
  END_PARAGRAPH
end
prog do <<-END_CODE
  a = Array.new  + [12345]  # Array  addition.
  b = String.new + 'hello'  # String addition.
  c = Time.new

  puts 'a = '+a.to_s
  puts 'b = '+b.to_s
  puts 'c = '+c.to_s
  END_CODE
end
para do <<-END_PARAGRAPH
  Because we can create arrays and strings using
  #{code '[...]'} and #{code "'...'"} respectively, we rarely create
  them using #{code 'new'}.  (Though it's not really obvious
  from the above example, #{code 'String.new'} creates
  an empty string, and #{code 'Array.new'} creates an empty
  array.)  Also, numbers are special exceptions:  you can't
  create an integer with #{code 'Integer.new'}.  You just have
  to write the integer.
  END_PARAGRAPH
end
h2 {"The #{code 'Time'} Class"}
para do <<-END_PARAGRAPH
  So what's the story with this #{code 'Time'} class?
  #{code 'Time'} objects represent moments in time.  You can
  add (or subtract) numbers to (or from) times to get new times:
  adding #{code '1.5'} to a time makes a new time one-and-a-half
  seconds later:
  END_PARAGRAPH
end
prog do <<-END_CODE
  time  = Time.new   # The moment I generated this web page.
  time2 = time + 60  # One minute later.

  puts time
  puts time2
  END_CODE
```

```
end
para do <<-END_PARAGRAPH
  You can also make a time for a specific moment using
  #{code 'Time.mktime'}:
  END_PARAGRAPH
end
prog do <<-END_CODE
  puts Time.mktime(2000, 1, 1)          # Y2K.
  puts Time.mktime(1976, 8, 3, 10, 11)  # When I was born.
  END_CODE
end
para do <<-END_PARAGRAPH
  Notice:  that's when I was born in Pacific Daylight Savings
  Time (PDT).  When Y2K struck, though, it was Pacific
  Standard Time (PST), at least to us West Coasters.  The
  parentheses are to group the parameters to #{code 'mktime'}
  together.  The more parameters you add, the more accurate your
  time becomes.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  You can compare times using the comparison methods
  (an earlier time is <em>less than</em> a later time),
  and if you subtract one time from another, you'll get the
  number of seconds between them.  Play around with it!
  END_PARAGRAPH
end

h2 {'A Few Things to Try'}
ul do
  li {"One billion seconds...  Find out the exact second you
       were born (if you can).  Figure out when you will turn (or
       perhaps when you did turn?) one billion seconds old.  Then
       go mark your calendar."}
  li {"Happy Birthday!  Ask what year a person was born in,
       then the month, then the day.  Figure out how old they are
       and give them a big #{output 'SPANK!'} for each birthday
       they have had."}

end

h2 {"The #{code 'Hash'} Class"}
para do <<-END_PARAGRAPH
  Another useful class is the #{code 'Hash'} class.  Hashes
  are a lot like arrays:  they have a bunch of slots which
```

```
  can point to various objects.  However, in an array, the
  slots are lined up in a row, and each one is numbered
  (starting from zero).  In a hash, the slots aren't in
  a row (they are just sort of jumbled together), and you
  can use <em>any</em> object to refer to a slot, not just
  a number.  It's good to use hashes when you have a bunch
  of things you want to keep track of, but they don't really
  fit into an ordered list.  For example, the colors I use for different
  parts of the code which created this tutorial:
  END_PARAGRAPH
end
prog do <<-END_CODE
  colorArray = []  # same as Array.new
  colorHash  = {}  # same as Hash.new

  colorArray[0]         = '#{@@STRING_COLOR}'
  colorArray[1]         = '#{@@NUMBER_COLOR}'
  colorArray[2]         = '#{@@KEYWORD_COLOR}'
  colorHash['strings']  = '#{@@STRING_COLOR}'
  colorHash['numbers']  = '#{@@NUMBER_COLOR}'
  colorHash['keywords'] = '#{@@KEYWORD_COLOR}'

  colorArray.each do |color|
    puts color
  end
  colorHash.each do |codeType, color|
    puts codeType + ':  ' + color
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  If I use an array, I have to remember that slot #{code '0'} is for
  strings, slot #{code '1'} is for numbers, etc.  But if I use a hash, it's
  easy!  Slot #{code "'strings'"} holds the color of the strings, of course.
  Nothing to remember.  You might have noticed that when we used
  #{code 'each'}, the objects in the hash didn't come out in the same
  order we put them in.  Arrays
  are for keeping things in order, not hashes.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Though people usually use strings to name the slots in a hash, you
  could use any kind of object, even arrays and other hashes (though I can't
  think of why you would want to do this...):
  END_PARAGRAPH
```

```
end
prog false do <<-END_CODE
  weirdHash = Hash.new

  weirdHash[12] = 'monkeys'
  weirdHash[[]] = 'emptiness'
  weirdHash[Time.new] = 'no time like the present'
  END_CODE
end
para do <<-END_PARAGRAPH
  Hashes and arrays are good for different things; it's up
  to you to decide which one is best for a particular problem.
  END_PARAGRAPH
end
h2 {'Extending Classes'}
para do <<-END_PARAGRAPH
  At the end of the last chapter, you wrote a method to give
  the English phrase for a given integer.  It wasn't an integer
  method, though; it was just a generic "program" method.  Wouldn't
  it be nice if you could write something like #{code '22.to_eng'}
  instead of #{code 'englishNumber 22'}?  Here's how you would do
  that:
  END_PARAGRAPH
end
# HACK ALERT!!!  (I can't get to the global namespace transparently
#                 from inside the StringIO object in a mod_ruby script.)
integerClassHack =
    "def to_eng
      if self == 5
        english = 'five'
      else
        english = 'fifty-eight'
      end

      english
    end"

Integer.module_eval integerClassHack  # This is the real method definition.
# The following defines a method in "another" integer class:
# END HACK ALERT!!!
prog do <<-END_CODE
  class Integer
    #{integerClassHack}
  end
```

```
  # I'd better test on a couple of numbers...
  puts 5.to_eng
  puts 58.to_eng
  END_CODE
end
para do <<-END_PARAGRAPH
  Well, I tested it; it seems to work.  ;)
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  So we defined an integer method by jumping into the
  #{code 'Integer'} class, defining the method there,
  and jumping back out.  Now all integers have this
  (somewhat incomplete) method.  In fact, if you didn't
  like the way a built-in method like
  #{code 'to_s'} worked, you could just
  redefine it in much the same way... but I don't recommend
  it!  It's best to leave the old methods alone and to
  make new ones when you want to do something new.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  So... confused yet?  Let me go over that last program
  some more.  So far, whenever we executed any code or
  defined any methods, we did it in the default
  "program" object.  In our last program, we left that
  object for the first time and went into the class
  #{code 'Integer'}.  We defined a method there (which
  makes it an integer method) and all integers can
  use it.  Inside that method we use #{code 'self'}
  to refer to the object (the integer) using the method.
  END_PARAGRAPH
end
h2 {'Creating Classes'}
para do <<-END_PARAGRAPH
  We've seen a number of different classes of objects.
  However, it's easy to come up with kinds of objects
  that Ruby doesn't have.  Luckily, creating a new
  class is as easy as extending an old one.  Let's say
  we wanted to make some dice in Ruby.  Here's how we
  could make the Die class:
  END_PARAGRAPH
end
prog do <<-END_CODE
  class Die
```

```
    def roll
      1 + rand(6)
    end

  end

  # Let's make a couple of dice...
  dice = [Die.new, Die.new]

  # ...and roll them.
  dice.each do |die|
    puts die.roll
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  (If you skipped the section on random numbers,
  #{code 'rand(6)'} just gives a random number between
  #{code '0'} and #{code '5'}.)
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  And that's it!  Objects of our very own.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  We can define
  all sorts of methods for our objects... but there's
  something missing.  Working with these objects feels
  a lot like programming before we learned about
  variables.  Look at our dice, for example.  We can
  roll them, and each time we do they give us a different
  number.  But if we wanted to hang on to that number, we
  would have to create a variable to point to the number.
  It seems like any decent die should be able to <em>have</em>
  a number, and that rolling the die should change the number.
  If we keep track of the die, we shouldn't also have to keep track
  of the number it is showing.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  However, if we try to store the number we rolled in a (local)
  variable in #{code 'roll'}, it will be gone as soon as
  #{code 'roll'} is finished.  We need to store the number in
```

```
  a different kind of variable:
  END_PARAGRAPH
end
h2 {'Instance Variables'}
para do <<-END_PARAGRAPH
  Normally when we want to talk about a string, we will just
  call it a <dfn>string</dfn>.  However, we could also call
  it a <dfn>string object</dfn>.  Sometimes programmers might
  call it <dfn>an instance of the class #{code 'String'}</dfn>, but this
  is just a fancy (and rather long-winded) way of saying
  <dfn>string</dfn>.  An <dfn>instance</dfn> of a class is just an
  object of that class.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  So instance variables are just an object's variables.  A
  method's local variables last until the method is finished.
  An object's instance variables, on the other hand, will
  last as long as the object does.  To tell instance variables
  from local variables, they have #{code '@'} in front of
  their names:
  END_PARAGRAPH
end
prog do <<-END_CODE
  class Die

    def roll
      @numberShowing = 1 + rand(6)
    end

    def showing
      @numberShowing
    end

  end

  die = Die.new
  die.roll
  puts die.showing
  puts die.showing
  die.roll
  puts die.showing
  puts die.showing
  END_CODE
end
```

```
para do <<-END_PARAGRAPH
  Very nice!  So #{code 'roll'} rolls the die and
  #{code 'showing'} tells us which number is showing.
  However, what if we try to look at what's showing before
  we've rolled the die (before we've set #{code '@numberShowing'})?
  END_PARAGRAPH
end
prog do <<-END_CODE
  class Die

    def roll
      @numberShowing = 1 + rand(6)
    end

    def showing
      @numberShowing
    end

  end

  # Since I'm not going to use this die again,
  # I don't need to save it in a variable.
  puts Die.new.showing
  END_CODE
end
para do <<-END_PARAGRAPH
  Hmmm... well, at least it didn't give us an error.  Still,
  it doesn't really make sense for a die to be "unrolled", or
  whatever #{output 'nil'} is supposed to mean here.  It would
  be nice if we could set up our new die object right when it's
  created.  That's what #{code 'initialize'} is for:
  END_PARAGRAPH
end
prog do <<-END_CODE
  class Die

    def initialize
      # I'll just roll the die, though we
      # could do something else if we wanted
      # to, like setting the die with 6 showing.
      roll
    end

    def roll
      @numberShowing = 1 + rand(6)
```

```
    end

    def showing
      @numberShowing
    end

  end

  puts Die.new.showing
  END_CODE
end
para do <<-END_PARAGRAPH
  When an object is created, its #{code 'initialize'}
  method (if it has one defined) is always called.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Our dice are just about perfect.  The only thing that
  might be missing is a way to set which side of a die
  is showing... why don't you write a #{code 'cheat'}
  method which does just that!  Come back when you're
  done (and when you tested that it worked, of course).
  Make sure that someone can't set the die to have a
  #{code '7'} showing!
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  So that's some pretty cool stuff we just covered.  It's tricky,
  though, so let me give another, more interesting example.
  Let's say we want to make a simple virtual pet, a baby
  dragon.  Like most babies, it should be able to eat, sleep,
  and poop, which means we will need to be able to feed it,
  put it to bed, and take it on walks.  Internally, our dragon
  will need to keep track of if it is hungry, tired, or needs
  to go, but we won't be able to see that when we interact
  with our dragon, just like you can't ask a human baby,
  "Are you hungry?".  We'll also add a few other fun ways
  we can interact with our baby dragon, and when he is born
  we'll give him a name.  (Whatever you pass into the
  #{code 'new'} method is passed into the #{code 'initialize'}
  method for you.)  Alright, let's give it a shot:
  END_PARAGRAPH
end
prog do <<-END_CODE
  class Dragon
```

```
  def initialize name
    @name = name
    @asleep = false
    @stuffInBelly     = 10  # He's full.
    @stuffInIntestine =  0  # He doesn't need to go.

    puts @name + ' is born.'
  end

  def feed
    puts 'You feed ' + @name + '.'
    @stuffInBelly = 10
    passageOfTime
  end

  def walk
    puts 'You walk ' + @name + '.'
    @stuffInIntestine = 0
    passageOfTime
  end

  def putToBed
    puts 'You put ' + @name + ' to bed.'
    @asleep = true
    3.times do
      if @asleep
        passageOfTime
      end
      if @asleep
        puts @name + ' snores, filling the room with smoke.'
      end
    end
    if @asleep
      @asleep = false
      puts @name + ' wakes up slowly.'
    end
  end

  def toss
    puts 'You toss ' + @name + ' up into the air.'
    puts 'He giggles, which singes your eyebrows.'
    passageOfTime
  end
```

```
  def rock
    puts 'You rock ' + @name + ' gently.'
    @asleep = true
    puts 'He briefly dozes off...'
    passageOfTime
    if @asleep
      @asleep = false
      puts '...but wakes when you stop.'
    end
  end

  private

  # "private" means that the methods defined here are
  # methods internal to the object.  (You can feed
  # your dragon, but you can't ask him if he's hungry.)

  def hungry?
    # Method names can end with "?".
    # Usually, we only do this if the method
    # returns true or false, like this:
    @stuffInBelly <= 2
  end

  def poopy?
    @stuffInIntestine >= 8
  end

  def passageOfTime
    if @stuffInBelly > 0
      # Move food from belly to intestine.
      @stuffInBelly     = @stuffInBelly     - 1
      @stuffInIntestine = @stuffInIntestine + 1
    else  # Our dragon is starving!
      if @asleep
        @asleep = false
        puts 'He wakes up suddenly!'
      end
      puts @name + ' is starving!  In desperation, he ate YOU!'
      exit  # This quits the program.
    end

    if @stuffInIntestine >= 10
      @stuffInIntestine = 0
      puts 'Whoops!  ' + @name + ' had an accident...'
```

```
        end

        if hungry?
          if @asleep
            @asleep = false
            puts 'He wakes up suddenly!'
          end
          puts @name + '\\'s stomach grumbles...'
        end

        if poopy?
          if @asleep
            @asleep = false
            puts 'He wakes up suddenly!'
          end
          puts @name + ' does the potty dance...'
        end
      end

    end

    pet = Dragon.new 'Norbert'
    pet.feed
    pet.toss
    pet.walk
    pet.putToBed
    pet.rock
    pet.putToBed
    pet.putToBed
    pet.putToBed
    pet.putToBed
    END_CODE
  end
  para do <<-END_PARAGRAPH
    <em>Whew!</em>  Of course, it would be nicer if this was
    an interactive program, but you can do that part later.
    I was just trying to show the parts directly relating to
    creating a new dragon class.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    We saw a few new things in that example.  The first is
    simple:  #{code 'exit'} terminates the program right
    then and there.  The second is the word #{code 'private'}
    which we stuck right in the middle of our class definition.
```

```
    I could have left it out, but I wanted to enforce the idea
    of certain methods being things you can do to a dragon, and
    others which simply happen within the dragon.  You can think
    of these as being "under the hood":  unless you are an
    automobile mechanic, all you really need to know is the gas
    pedal, the brake pedal, and the steering wheel.  A programmer
    might call those the <dfn>public interface</dfn> to your car.
    How your airbag knows when to deploy, however, is internal to
    the car; the typical user (driver) doesn't need to know about
    this.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    Actually, for a bit more concrete example along those lines,
    let's talk about how you might represent a car in a video
    game (which happens to be my line of work).  First, you would
    want to decide what you want your public interface to look like;
    in other words, which methods should people be able to call on
    one of your car objects?  Well, they need to be able to push
    the gas pedal and the brake pedal, but they would also need to
    be able to specify how hard they are pushing the pedal.  (There's
    a big difference between flooring it and tapping it.)  They would
    also need to be able to steer, and again, they would need to be
    able to say how hard they are turning the wheel.  I suppose you
    could go further and add a clutch, turn signals, rocket launcher,
    afterburner, flux capacitor, etc... it depends
    on what type of game you are making.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    Internal to a car object, though, there would need to be much
    more going on; other things a car would need are a speed,
    a direction, and a position (at the most basic).  These attributes
    would be modified by pressing on the gas or brake pedals and
    turning the wheel, of course, but the user would not be able
    to set the position directly (which would be like warping).
    You might also want to keep track of skidding or damage, if
    you have caught any air, and so on.  These would all be internal
    to your car object.
    END_PARAGRAPH
  end

  h2 {'A Few Things to Try'}
  ul do
    li {"Make an #{code 'OrangeTree'} class.  It should have a
```

```
          #{code 'height'} method which returns its height, and a
          #{code 'oneYearPasses'} method, which, when called, ages the tree
          one year.  Each year the tree grows taller (however much you think
          an orange tree should grow in a year), and after some number of
          years (again, your call) the tree should die.  For the first few
          years, it should not produce fruit, but after a while it should,
          and I guess that older trees produce more each year than younger
          trees... whatever you think makes most sense.  And, of course,
          you should be able to #{code 'countTheOranges'} (which returns
          the number of oranges on the tree), and #{code 'pickAnOrange'}
          (which reduces the #{code '@orangeCount'} by one and returns a
          string telling you how delicious the orange was, or else it just
          tells you that there are no more oranges to pick this year).
          Make sure that any oranges you don't pick one year fall off
          before the next year."}
      li {"Write a program so that you can interact with your
          baby dragon.  You should be able to enter commands like
          #{input 'feed'} and #{input 'walk'}, and have those methods
          be called on your dragon.  Of course, since what you are
          inputting are just strings, you will have to have some sort
          of <dfn>method dispatch</dfn>, where your program checks
          which string was entered, and then calls the appropriate method."}
    end

    para do <<-END_PARAGRAPH
      And that's just about all there is to it!  But wait a second...
      I haven't told you about any of those classes for doing things
      like sending an email, or saving and loading files on your
      computer, or how to create windows and buttons, or 3D worlds,
      or anything!  Well, there are just <em>so many</em> classes
      you can use that I can't possibly show you them all; I don't
      even know what most of them are!  What I <em>can</em> tell
      you is where to find out more about them, so you can learn
      about the ones you want to program with.  Before I send you
      off, though, there is just one more feature of Ruby you should
      know about, something most languages don't have, but which I
      simply could not live without:
      #{makeLink 'blocks and procs', :generateBlocksProcs}.
      END_PARAGRAPH
    end

  end

  #
  # BLOCKS AND PROCS
```

```
#

def generateBlocksProcs
  para do <<-END_PARAGRAPH
    This is definitely one of the coolest features of Ruby.  Some
    other languages have this feature, though they may call it
    something else (like <dfn>closures</dfn>), but most of the
    more popular ones don't, and it's a shame.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    So what is this cool new thing?  It's the ability to take
    a <dfn>block</dfn> of code (code in between #{code 'do'}
    and #{code 'end'}), wrap it up in an object (called a
    <dfn>proc</dfn>), store it in a variable or pass it to a
    method, and run the code in the block whenever you feel
    like (more than once, if you want).  So it's kind of like
    a method itself, except that it isn't bound to an object
    (it <em>is</em> an object), and you can store it or pass
    it around like you can with any object.  I think it's example
    time:
    END_PARAGRAPH
  end
  prog do <<-END_CODE
    toast = Proc.new do
      puts 'Cheers!'
    end

    toast.call
    toast.call
    toast.call
    END_CODE
  end
  para do <<-END_PARAGRAPH
    So I created a proc (which I think is supposed to be short for
    "procedure", but far more importantly, it rhymes with "block")
    which held the block of code, then I #{code 'call'}ed the proc
    three times.  As you can see, it's a lot like a method.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    Actually, it's even more like a method than I have shown you, because
    blocks can take parameters:
    END_PARAGRAPH
  end
```

```
    prog do <<-END_CODE
      doYouLike = Proc.new do |aGoodThing|
        puts 'I *really* like '+aGoodThing+'!'
      end

      doYouLike.call 'chocolate'
      doYouLike.call 'ruby'
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Ok, so we see what blocks and procs are, and how to use them, but what's
      the point?  Why not just use methods?  Well, it's because there are some
      things you just can't do with methods.  In particular, you can't pass
      methods into other methods (but you can pass procs into methods), and methods
      can't return other methods (but they can return procs).  This is simply
because
      procs are objects; methods aren't.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      (By the way, is any of this looking familiar?  Yep, you've seen blocks
before... when
      you learned about iterators.  But let's talk more about that in a bit.)
      END_PARAGRAPH
    end
    h2 { 'Methods Which Take Procs' }
    para do <<-END_PARAGRAPH
      When we pass a proc into a method, we can control how, if, or how many times
we call
      the proc.  For example, let's say there's something we want to do before and
after
      some code is run:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      def doSelfImportantly someProc
        puts 'Everybody just HOLD ON!  I have something to do...'
        someProc.call
        puts 'Ok everyone, I\\'m done.  Go on with what you were doing.'
      end

      sayHello = Proc.new do
        puts 'hello'
      end
```

```
      sayGoodbye = Proc.new do
        puts 'goodbye'
      end

      doSelfImportantly sayHello
      doSelfImportantly sayGoodbye
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Maybe that doesn't appear particulary fabulous... but it is.  :-)
      It's all too common in programming to have strict requirements about what
      must be done when.  If you want to save a file, for example, you have to
      open the file, write out the information you want it to have, and then close
      the file.  If you forget to close the file, Bad Things(tm) can happen.  But
      each time you want to save or load a file, you have to do the same thing:
      open the file, do what you <em>really</em> want to do, then close the file.
      It's tedious and easy to forget.  In Ruby, saving (or loading) files works
      similarly to the code above, so you don't have to worry about anything but
      what you actually want to save (or load).  (In the next chapter I'll show you
      where to find out how to do things like save and load files.)
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      You can also write methods which will determine how many times, or even
      <em>if</em> to call a proc.  Here's a method which will call the proc passed
in
      about half of the time, and another which will call it twice:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      def maybeDo someProc
        if rand(2) == 0
          someProc.call
        end
      end

      def twiceDo someProc
        someProc.call
        someProc.call
      end

      wink = Proc.new do
        puts '<wink>'
      end
```

```
      glance = Proc.new do
        puts '<glance>'
      end

      maybeDo wink
      maybeDo glance
      twiceDo wink
      twiceDo glance
      END_CODE
    end
    para do <<-END_PARAGRAPH
      These are some of
      the more common uses of procs which enable us to do things we simply could not
have done
      using methods alone.  Sure, you could write a method to wink twice, but you
couldn't write
      one to just do <em>something</em> twice!
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      Before we move on, let's look at one last example.  So far the procs
      we have passed in have been fairly similar to each other.  This time
      they will be quite different, so you can see how much such a method
      depends on the procs passed into it.  Our
      method will take some object and a proc, and will call the proc
      on that object.  If the proc returns false, we quit; otherwise
      we call the proc with the returned object.  We keep doing this
      until the proc returns false (which it had better do eventually,
      or the program will crash).  The method will return the last
      non-false value returned by the proc.
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      def doUntilFalse firstInput, someProc
        input  = firstInput
        output = firstInput

        while output
          input  = output
          output = someProc.call input
        end

        input
      end
```

```
      buildArrayOfSquares = Proc.new do |array|
        lastNumber = array.last
        if lastNumber <= 0
          false
        else
          array.pop                          # Take off the last number...
          array.push lastNumber*lastNumber   # ...and replace it with its square...
          array.push lastNumber-1            # ...followed by the next smaller
number.
        end
      end

      alwaysFalse = Proc.new do |justIgnoreMe|
        false
      end

      puts doUntilFalse([5], buildArrayOfSquares).inspect
      puts doUntilFalse('I\\'m writing this at 3:00 am; someone knock me out!',
alwaysFalse)
      END_CODE
    end
    para do <<-END_PARAGRAPH
      Ok, so that was a pretty weird example, I'll admit.  But it shows how
differently
      our method acts when given very different procs.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      The #{code 'inspect'} method is a lot like #{code 'to_s'}, except
      that the string it returns tries to show you the ruby code for
      building the object you passed it.  Here it shows us the whole
      array returned by our first call to #{code 'doUntilFalse'}.  Also, you might
      notice that we never actually squared that #{code '0'} on the end of that
      array, but since #{code '0'} squared is still just #{code '0'}, we didn't have
to.
      And since #{code 'alwaysFalse'} was, you know, always #{code 'false'},
      #{code 'doUntilFalse'} didn't do anything at all the second time we
      called it; it just returned what was passed in.
      END_PARAGRAPH
    end
    h2 { 'Methods Which Return Procs' }
    para do <<-END_PARAGRAPH
      One of the other cool things you can do with procs is to create
      them in methods and return them.  This allows all sorts of crazy
      programming power (things with impressive names, like
```

```
  <dfn>lazy evaluation</dfn>, <dfn>infinite data structures</dfn>,
  and <dfn>currying</dfn>),
  but the fact is that I almost never do this in practice, nor
  can I remember seeing anyone else do this in their code.  I think
  it's the kind of thing you don't usually end up having to do in Ruby,
  or maybe Ruby just encourages you to find other solutions; I don't
  know.  In any case, I will only touch on this briefly.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  In this example, #{code 'compose'} takes two procs and returns a new
  proc which, when called, calls the first proc and passes its result
  into the second proc.
  END_PARAGRAPH
end
prog do <<-END_CODE
  def compose proc1, proc2
    Proc.new do |x|
      proc2.call(proc1.call(x))
    end
  end

  squareIt = Proc.new do |x|
    x * x
  end

  doubleIt = Proc.new do |x|
    x + x
  end

  doubleThenSquare = compose doubleIt, squareIt
  squareThenDouble = compose squareIt, doubleIt

  puts doubleThenSquare.call(5)
  puts squareThenDouble.call(5)
  END_CODE
end
para do <<-END_PARAGRAPH
  Notice that the call to #{code 'proc1'} had to be inside the
  parentheses for #{code 'proc2'} in order for it to be done first.
  END_PARAGRAPH
end
h2 { 'Passing Blocks (Not Procs) into Methods' }
para do <<-END_PARAGRAPH
  Ok, so this has been sort of academically interesting, but also
```

```
  sort of a hassle to use.  A lot of the problem is that there are
  three steps you have to go through (defining the method, making
  the proc, and calling the method with the proc), when it sort of
  feels like there should only be two (defining the method, and
  passing the <em>block</em> right into the method, without using
  a proc at all), since most of the time you don't want to use the
  proc/block after you pass it into the method.  Well, wouldn't you
  know, Ruby has it all figured out for us!  In fact, you've already
  been doing it every time you use iterators.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  I'll show you a quick example first, then we'll talk about it.
  END_PARAGRAPH
end
# HACK ALERT!!!  (I can't get to the global namespace transparently
#                 from inside the StringIO object in a mod_ruby script.)
arrayClassHack =
    "def eachEven(&wasABlock_nowAProc)
      # We start with \"true\" because arrays start with 0, which is even.
      isEven = true

      self.each do |object|
        if isEven
          wasABlock_nowAProc.call object
        end

        isEven = (not isEven)  # Toggle from even to odd, or odd to even.
      end
    end"

Array.module_eval arrayClassHack  # This is the real method definition.
# The following defines a method in "another" array class:
# END HACK ALERT!!!
prog do <<-END_CODE
  class Array
    #{arrayClassHack}
  end

  ['apple', 'bad apple', 'cherry', 'durian'].eachEven do |fruit|
    puts 'Yum!  I just love '+fruit+' pies, don\\'t you?'
  end

  # Remember, we are getting the even-numbered elements
  # of the array, all of which happen to be odd numbers,
```

```
      # just because I like to cause problems like that.
      [1, 2, 3, 4, 5].eachEven do |oddBall|
        puts oddBall.to_s+' is NOT an even number!'
      end
      END_CODE
    end
    para do <<-END_PARAGRAPH
      So to pass in a block to #{code 'eachEven'}, all we had to do was stick
      the block after the method.  You can pass a block into any method this
      way, though many methods will just ignore the block.  In order to make
      your method <em>not</em> ignore the block, but grab it and turn it into
      a proc, put the name of the proc at the end of your method's parameter
      list, preceded by an ampersand (#{code '&'}).  So that part is a little
      tricky, but not too bad, and you only have to do that once (when you
      define the method).  Then you can use the method over and over again,
      just like the built-in methods which take blocks, like #{code 'each'}
      and #{code 'times'}.  (Remember #{code '5.times do'}...?)
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      If you get confused, just remember what #{code 'eachEven'} is supposed to do:
call
      the block passed in with every other element in the array.  Once
      you've written it and it works, you don't need to think about what it's
      actually doing under the hood ("which block is called when??"); in
      fact, that's exactly <em>why</em> we write methods like this:  so we
      never have to think about how they work again.  We just use them.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      I remember one time I wanted to be able to time how long different
      sections of a program were taking.  (This is also known as
      <dfn>profiling</dfn> the code.)  So I wrote a method which takes
      the time before running the code, then it runs it, then it takes
      the time again at the end and figures out the difference.  I can't
      find the code right now, but I don't need it; it probably
      went something like this:
      END_PARAGRAPH
    end
    prog do <<-END_CODE
      def profile descriptionOfBlock, &block
        startTime = Time.now

        block.call
```

```
    duration = Time.now - startTime

    puts descriptionOfBlock+':  '+duration.to_s+' seconds'
  end

  profile '25000 doublings' do
    number = 1

    25000.times do
      number = number + number
    end

    # Show the number of digits in this HUGE number.
    puts number.to_s.length.to_s+' digits'
  end

  profile 'count to a million' do
    number = 0

    1000000.times do
      number = number + 1
    end
  end
  END_CODE
end
para do <<-END_PARAGRAPH
  How simple!  How elegant!  With that tiny method,
  I can now easily time any section of any program that I want to; I
  just throw the code in a block and send it to #{code 'profile'}.
  What could be simpler?  In most languages, I would have to explicitly
  add that timing code (the stuff in #{code 'profile'}) around every
  section which I wanted to time.  In Ruby, however, I get to keep it
  all in one place, and (more importantly) out of my way!
  END_PARAGRAPH
end

h2 {'A Few Things to Try'}
ul do
  li do
    "<em>Grandfather Clock</em>.  Write a method which takes a block
    and calls it once for each hour that has passed today.  That way, if I
    were to pass in the block #{code "do puts 'DONG!' end"}, it would chime
    (sort of) like a grandfather clock.  Test your method
    out with a few different blocks (including the one I just gave you).
    <em><strong>Hint:</strong>  You can use
```

```
        </em>#{code 'Time.now.hour'}<em> to get the current hour.
        However, this returns a number between </em>#{code '0'}<em> and </em>#{code
'23'}<em>,
        so you will have to alter those numbers in order to get ordinary clock-face
        numbers (</em>#{code '1'}<em> to </em>#{code '12'}<em>).</em>"
      end
      li do
        para {"<em>Program Logger</em>.  Write a method called #{code 'log'}, which
              takes a string description of a block and, of course, a block.
Similar to
              #{code 'doSelfImportantly'}, it should #{code 'puts'} a string telling
              that it has started the block, and another string at the end telling
you
              that it has finished the block, and also telling you what the block
returned.
              Test your method by sending it a code block.  Inside the block, put
<em>another</em>
              call to #{code 'log'}, passing another block to it.  (This is called
              <dfn>nesting</dfn>.)  In other words, your output should look
something like this:"}
        puts '<pre class="L2PoutputBlock">' +
              'Beginning "outer block"...' + $/ +
              'Beginning "some little block"...' + $/ +
              '..."some little block" finished, returning:  5' + $/ +
              'Beginning "yet another block"...' + $/ +
              '..."yet another block" finished, returning:  I like Thai food!' + $/
+
              '..."outer block" finished, returning:  false' + $/ +
              '</pre>'
      end
      li do
        puts "<em>Better Logger</em>.  The output from that last logger was kind
              of hard to read, and it would just get worse the more you used it.  It
would
              be so much easier to read if it indented the lines in the inner
blocks.  To
              do this, you'll need to keep track of how deeply nested you are every
time
              the logger wants to write something.  To do this, use a <dfn>global
variable</dfn>,
              a variable you can see from anywhere in your code.  To make a global
variable,
              just precede your variable name with #{code '$'}, like these:
              #{code '$global'}, #{code '$nestingDepth'}, and #{code
'$bigTopPeeWee'}.
```

```
              In the end, your logger should output code like this:"
        puts '<pre class="L2PoutputBlock">' +
              'Beginning "outer block"...' + $/ +
              '  Beginning "some little block"...' + $/ +
              '    Beginning "teeny-tiny block"...' + $/ +
              '    ..."teeny-tiny block" finished, returning:  lots of love' + $/ +
              '  ..."some little block" finished, returning:  42' + $/ +
              '  Beginning "yet another block"...' + $/ +
              '  ..."yet another block" finished, returning:  I love Indian food!' +
$/ +
              '..."outer block" finished, returning:  true' + $/ +
              '</pre>'
      end
    end

    para do <<-END_PARAGRAPH
      Well, that's about all you're going to learn from this tutorial.
      Congratulations!  You've learned a <em>lot</em>!  Maybe you don't feel
      like you remember everything, or you skipped over some parts... really,
      that's just fine.  Programming isn't about what you know; it's about
      what you can figure out.  As long as you know where to find out the
      things you forgot, you're doing just fine.  I hope you don't think
      that I wrote all of this without looking things up every other minute!
      Because I did.  I also got a lot of help with the code which runs all
      of the examples in this tutorial.  But where was <em>I</em> looking
      stuff up, and who was <em>I</em> asking for help?
      #{makeLink 'Let me show you...', :generateBeyond}
      END_PARAGRAPH
    end
  end

  #
  # BEYOND THIS TUTORIAL
  #

  def generateBeyond
    para do <<-END_PARAGRAPH
      So where do we go now?  If you have a question, who can you
      ask?  What if you want your program to open a webpage, send
      an email, or resize a digital picture?  Well, there are many,
      many places to find Ruby help.  Unfortunately,
      that's sort of unhelpful, isn't it?  :-)
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
```

```
      For me, there are really only three places I look for Ruby help.
      If it's a small question, and I think I can experiment on my own
      to find the answer, I use <dfn>irb</dfn>.  If it's a bigger question,
      I look it up in my <dfn>pickaxe</dfn>.  And if I just can't figure
      it out on my own, then I ask for help on <dfn>ruby-talk</dfn>.
      END_PARAGRAPH
    end
    h2 {'IRB:  Interactive Ruby'}
    para do <<-END_PARAGRAPH
      If you installed Ruby, then you installed irb.  To use it, just
      go to your command prompt and type #{input 'irb'}.  When you are
      in irb, you can type in any ruby expression you want, and it will tell you
      the value of it.  Type in #{input '1 + 2'}, and it will tell you
      #{output '3'}.  (Note that you don't have to use #{code 'puts'}.)
      It's kind of like a giant Ruby calculator.  When you are done,
      just type in #{input 'exit'}.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      There's a lot more to irb than this, but you can learn all about
      it in the pickaxe.
      END_PARAGRAPH
    end
    h2 {'The Pickaxe:  "Programming Ruby"'}
    para do <<-END_PARAGRAPH
      Absolutely <em>the</em> Ruby book to get is "Programming Ruby,
      The Pragmatic Programmer's Guide", by David Thomas and Andrew
      Hunt (the Pragmatic Programmers).  While I highly recommend
      picking up the
      <a href="https://www.pragprog.com/book/ruby4/programming-ruby-1-9-2-0">4th
edition</a>
      of this excellent book, with all of
      the latest Ruby covered, you can also get a slightly older
      (but still mostly relevant) version for free online.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      You can find just about everything about Ruby, from the basic
      to the advanced, in this book.  It's easy to read; it's comprehensive;
      it's just about perfect.  I wish every language had a book of
      this quality.  At the back of the book, you'll find a huge section
      detailing every method in every class, explaining it and giving
      examples.  I just love this book!
      END_PARAGRAPH
    end
```

```
    para do <<-END_PARAGRAPH
      There are a number of places you can get it (including
      the Pragmatic Programmers' own site), but my favorite place
      is at <a href="http://www.ruby-doc.org/docs/ProgrammingRuby/">ruby-
doc.org</a>.
      That version has a nice table of contents on the side,
      as well as an index.  (ruby-doc.org has lots of other
      great documentation as well, such as for the Core API and
      Standard Library... basically, it documents everything Ruby
      comes with right out of the box.
      <a href="http://www.ruby-doc.org/">Check it out.</a>)
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      And why is it called "the pickaxe"?  Well, there's a picture
      of a pickaxe on the cover of the book.  It's a silly name, I
      guess, but it stuck.
      END_PARAGRAPH
    end
    h2 {'Ruby-Talk:  the Ruby Mailing List'}
    para do <<-END_PARAGRAPH
      Even with irb and the pickaxe, sometimes you still can't figure
      it out.  Or perhaps you want to know if someone already did
      whatever it is you are working on, to see if you could use it
      instead.  In these cases, the place to go is ruby-talk, the Ruby
      Mailing List.  It's full of friendly, smart, helpful people.
      To learn more about it, or to subscribe, look
      <a href="http://www.ruby-lang.org/en/community/mailing-lists/">here</a>.
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      <strong>WARNING:</strong>  There's a <em>lot</em> of mail on the
      mailing list every day.  I have mine automatically sent to a
      different mail folder so that it doesn't get in my way.  If you
      just don't want to deal with all that mail, though, you don't
      have to!  The ruby-talk mailing list is mirrored to the newsgroup
      comp.lang.ruby, and vice versa, so you can see the same messages
      there.  Either way, you see the same messages, just in a slightly
      different format.
      END_PARAGRAPH
    end
    h2 {'Tim Toady'}
    para do <<-END_PARAGRAPH
      Something I have tried to shield you from, but which you will
      surely run in to soon, is the concept of TMTOWTDI (pronounced
```

```
    "Tim Toady"):  There's More Than One Way To Do It.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    Now some will tell you what a wonderful thing TMTOWTDI is, while
    others feel quite differently.  I don't really have strong feelings
    about it in general, but I think it's a <em>terrible</em> way to
    teach someone how to program.  (As if learning one way to do something
    wasn't challenging and confusing enough!)
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    However, now that you are moving beyond this tutorial, you'll
    be seeing much more diverse code.  For example, I can think of
    at least five other ways to make a string (aside from surrounding
    some text in single quotes), and each one works slightly differently.
    I only showed you the simplest of the six.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    And when we talked about branching, I showed you #{code 'if'},
    but I didn't show you #{code 'unless'}.  I'll let you figure
    that one out in irb.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    Another nice little shortcut you can use with #{code 'if'},
    #{code 'unless'}, and #{code 'while'}, is the cute one-line version:
    END_PARAGRAPH
  end
  prog do <<-END_CODE
    # These words are from a program I wrote to generate
    # English-like babble.  Cool, huh?
    puts 'grobably combergearl thememberate' if 5 == 2**2 + 1**1
    puts 'enlestrationshifter supposine' unless 'Chris'.length == 5
    END_CODE
  end
  para do <<-END_PARAGRAPH
    And finally, there is another way of writing methods which take blocks
    (not procs).  We saw the thing where we grabbed the block and turned
    it into a proc using the #{code '&block'} trick in your parameter list
    when you define the function.  Then, to call the block, you just use
    #{code 'block.call'}.  Well, there's a shorter way (though I personally
    find it more confusing).  Instead of this:
    END_PARAGRAPH
```

```
  end
  prog do <<-END_CODE
    def doItTwice(&block)
      block.call
      block.call
    end

    doItTwice do
      puts 'murditivent flavitemphan siresent litics'
    end
    END_CODE
  end
  para do <<-END_PARAGRAPH
    ...you do this:
    END_PARAGRAPH
  end
  prog do <<-END_CODE
    def doItTwice
      yield
      yield
    end

    doItTwice do
      puts 'buritiate mustripe lablic acticise'
    end
    END_CODE
  end
  para do <<-END_PARAGRAPH
    I don't know... what do you think?  Maybe it's just me, but...
    #{code 'yield'}?!  If it was something like #{code 'call_the_hidden_block'}
    or something, that would make a <em>lot</em> more sense to me.
    A lot of people say #{code 'yield'} makes sense to them.  But
    I guess that's what TMTOWTDI is all about:  they do it their way,
    and I'll do it my way.
    END_PARAGRAPH
  end
  h2 {'THE END'}
  para do <<-END_PARAGRAPH
    Use it for good and not evil.  :-)  And if you found this tutorial
    useful (or confusing, or if you found an error),
    <a href="mailto:chris@pine.fm">let me know</a>!
    END_PARAGRAPH
  end

end
```

```
#
# MAIN
#

def generateMain
  h2 { 'A Place to Start for the Future Programmer' }
  para do <<-END_PARAGRAPH
    I guess this all began back in 2002.  I was thinking
    about teaching programming, and what a great language
    Ruby would be for learning how to program.  I mean, we were
    all excited about Ruby because it was powerful, elegant, and
    really just fun, but it seemed to me that it would also
    be a great way to get into programming in the first place.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    Unfortunately, there wasn't much Ruby documentation
    geared for newbies at the time.  Some of us
    in the community were talking about what such a
    "Ruby for the Nuby" tutorial would
    need, and more generally, how to teach programming at all.
    The more I thought about this, the more I had to say (which
    surprised me a bit).  Finally, someone said, "Chris,
    why don't you just write a tutorial instead of talking about
    it?"  So I did.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    And it wasn't very good.  I had all these ideas that were good
    <em>in theory</em>, but the actual task of making a great
    tutorial for non-programmers was vastly more challenging than
    I had realized.  (I mean, it seemed good to me, but I already
    knew how to program.)
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    What saved me was that I made it really easy for people to
    contact me, and I always tried to help people when they got
    stuck.  When I saw a lot of people getting stuck in one place,
    I'd rewrite it.  It was a lot of work, but it slowly got better
    and better.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
```

```
  A couple of years later, it was getting pretty good.  :-)  So
  good, in fact, that I was ready to pronounce it finished, and
  move on to something else.  And right about then came an
  opportunity to turn the tutorial into a book.  Since it was
  already basically done, I figured this would be no problem.
  I'd just clean up a few spots, add some more exercises, maybe
  some more examples, a few more chapters, run it by 50 more
  reviewers...
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  It took me another year, but now I think it's really
  <em>really</em> good, mostly because of the hundreds of
  brave souls who have helped me write it.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  What's here on this site is the original tutorial, more or less
  unchanged since 2004.  For the latest and greatest, you'll
  want to check out <a href="#{FRLTP_ADDR}">the book</a>.
  END_PARAGRAPH
end
puts @@HLINE
h2 { 'Thoughts For Teachers' }
para do <<-END_PARAGRAPH
  There were a few guiding principles that I tried to stick to.
  I think they make the learning process much smoother;
  learning to program is hard enough as it is.  If you're
  teaching or guiding someone on the road to hackerdom, these
  ideas might help you, too.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  First, I tried to separate concepts as much as possible,
  so that the student would only have to learn
  one concept at a time.  This was difficult at first, but a little
  <em>too</em> easy after I had some practice.  Some things must be
  taught before others, but I was amazed at how little of
  a precedence hierarchy there really is.  Eventually, I just had to
  pick an order, and I tried to arrange things so that each
  new section was motivated by the previous ones.
  END_PARAGRAPH
end
para do <<-END_PARAGRAPH
  Another principle I've kept in mind is to teach only one way
```

```
    to do something.  It's an obvious benefit in a tutorial for
    people who have never programmed before.  For one thing,
    one way to do something is easier to learn than two.  Perhaps
    the more important benefit, though, is that the fewer things
    you teach a new programmer, the more creative and clever
    they have to be in their programming.  Since so much of programming
    is problem solving, it's crucial to encourage that as much
    as possible at every stage.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    I have tried to piggy-back programming concepts onto concepts
    the new programmer already has; to present ideas in such a way
    that their intuition will carry the load, rather than the
    tutorial.  Object-Oriented programming lends itself to this
    quite well.  I was able to begin referring to "objects" and
    different "kinds of objects" pretty early in the tutorial,
    slipping those phrases in at the most innocent of moments.
    I wasn't saying anything like "everything in Ruby is an
    object," or "numbers and strings are kinds of objects,"
    because these statements really don't mean anything to
    a new programmer.  Instead, I would talk about strings
    (not "string objects"), and sometimes I would refer to
    "objects", simply meaning "the things in these programs."
    The fact that all these <em>things</em> in Ruby <em>are</em> objects
    made this sort of sneakiness on my part work so well.
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    Although I wanted to avoid needless OO jargon, I wanted
    to make sure that, if they did need to learn a word, they
    learned the right one.  (I don't want them to have to learn
    it twice, right?)  So I called them "strings," not "text."  Methods
    needed to be called something, so I called them "methods."
    END_PARAGRAPH
  end
  para do <<-END_PARAGRAPH
    As far as the exercises are concerned, I think I came up
    with some good ones, but you can never have too many.
    Honestly, I bet I spent half of my time just trying to
    come up with fun, interesting exercises.
    Boring exercises absolutely kill any desire
    to program, while the perfect exercise creates an itch
    the new programmer can't help but scratch.  In short,
    you just can't spend too much time coming up with good
```

```
      exercises.
      END_PARAGRAPH
    end
    puts @@HLINE
    h2 { 'About the Original Tutorial' }
    para do <<-END_PARAGRAPH
      The pages of the tutorial (and even this page) are generated by a
      <a href="#{LINK_ADDR}?ShowTutorialCode=true">big Ruby program</a>,
      of course.  :-)
      All of the
      code samples were automatically run,
      and the output shown is the output they generated.
      I think this is the best, easiest, and
      certainly the coolest way to make sure that all of the
      code I present works <em>exactly</em> as I say it does.
      You don't have to worry that I might have copied the output
      of one of the examples
      wrong, or forgotten to test some of the code; it's all been tested.
      END_PARAGRAPH
    end
    para do
      '<a href="http://ruby-lang.org">'+
        '<img src="/images/PoweredByRuby.png" alt="powered by Ruby" width="234"
height="60" />'+
      '</a>'
    end
    puts @@HLINE
    h2 { 'Acknowledgements' }
    para do <<-END_PARAGRAPH
      Finally, I'd like to thank everyone on the ruby-talk mailing list
      for their thoughts and encouragement, all of my wonderful
      reviewers for their help in making the book far better than
      I could have alone, my dear wife especially
      for being my main reviewer/tester/guinea-pig/muse,
      Matz for creating this fabulous language, and the Pragmatic Programmers
      for telling me about it&mdash;and, of course, for publishing
      my book!
      END_PARAGRAPH
    end
    para do <<-END_PARAGRAPH
      If you notice any errors or typos, or have any comments or
      suggestions or good exercises I could include, please
      <a href="mailto:chris@pine.fm">let me know</a>.
      END_PARAGRAPH
    end
```

```
  end


  # menu helpers

  def menuBookLink
    para(:class=>'fancyMenuText') { 'Buy the <em>improved</em>' }
    para(:class=>' wideMenuText') { 'expanded' }
    para(:class=>'fancyMenuText') { 'version:' }
    puts "<a href=\"#{FRLTP_ADDR}\">"
      img(class: 'shadowed', width: '200', height: '240', src:
'/images/LTP2_cover.jpg', alt: 'Learn to Program, Second Edition')
    puts '</a>'
    para(:class=>'fancyMenuText') { '<em>answers now included!</em>' }
  end

  def menuTOC
    para { '&laquo; the original tutorial &raquo;' }

    ol(:start=>'0') do
      CHAPTERS.sort_by{|x| x[0]}.each do |aChapNum, aTitle, aMethod|
        if aChapNum < 'A'
          li { makeLink(aTitle, aMethod) }
        end
      end
    end
  end

  def menuTranslations
    para { '&laquo; translations &raquo;' }

    table do
      TRANSLATIONS.each do |trans|
        by = ('by '+trans[1]).gsub(' ',' ').gsub('-','&#8209;')
        tr do
          td { para(class: 'translink') {"<a href=#{trans[2]}>#{trans[0]}</a>"} }
          td { para                     {               by                    } }
        end
      end
    end
  end


  #
```

```
  # MAIN PAGE GENERATION
  #

  def generate(chap, title, generatingMethod)
    srand(12345 + 54321 * chap.to_i)
    chap = chap.sub(/^0/, '')

    pageTitle = if title.nil?
      'Learn to Program, by Chris Pine'
    else
      title + ' - Learn to Program'
    end

    puts '<!DOCTYPE html>'
    html do
      head do
        meta(charset: 'UTF-8')
        link(:href=>LINK_ADDR+'tutorial.css', :type=>'text/css', :rel=>'Stylesheet',
:media=>'screen')
        link(href: 'https://fonts.googleapis.com/css?family=Libre+Baskerville', rel:
'stylesheet', type: 'text/css')
        link(href: 'https://fonts.googleapis.com/css?
family=Source+Code+Pro&amp;subset=latin-ext,latin', rel: 'stylesheet', type:
'text/css')
        title { pageTitle }
        script(type: 'text/javascript') do
          puts ""
          puts "var _gaq = _gaq || [];"
          puts "_gaq.push(['_setAccount', 'UA-28406155-1']);"
          puts "_gaq.push(['_trackPageview']);"
          puts ""
          puts "(function() {"
          puts "  var ga = document.createElement('script'); ga.type =
'text/javascript'; ga.async = true;"
          puts "  ga.src = ('https:' == document.location.protocol ? 'https://ssl' :
'http://www') + '.google-analytics.com/ga.js';"
          puts "  var s = document.getElementsByTagName('script')[0];
s.parentNode.insertBefore(ga, s);"
          puts "})();"
          puts ""
        end
      end # head
      body do
        div do
          header do
```

```
            puts '<a href="'+LINK_ADDR+'">'
            puts 'Learn to Program'
            puts '</a>'
          end
          main do
            if chap != 'main'
              h1 {title}
              h3 {'Chapter '+chap}
              puts @@HLINE
            end
            method(generatingMethod).call
          end
          footer do
            puts @@HLINE
            para do
              '<a href="https://twitter.com/OtherChrisPine" class="twitter-follow-
button" data-show-count="false" data-size="large">Follow @OtherChrisPine</a>'
            end
            para {"&copy; 2003-#{Time.now.year} Chris Pine"}
            script(type: 'text/javascript') do
              "!function(d,s,id){var js,fjs=d.getElementsByTagName(s)
[0],p=/^http:/.test(d.location)?'http':'https';if(!d.getElementById(id))
{js=d.createElement(s);js.id=id;js.src=p+'://platform.twitter.com/widgets.js';fjs.pa
rentNode.insertBefore(js,fjs);}}(document, 'script', 'twitter-wjs');"
            end
          end
        end
        nav do
          menuBookLink

          img(class: 'divider', width: '150', src: '/images/divider.svg', alt:
'divider')

          menuTOC

          img(class: 'divider', width: '150', src: '/images/divider.svg', alt:
'divider')

          menuTranslations
        end
      end # body
    end # html

    @page.join("\n")+"\n"
  end
```

```
end




# 'format' is a hidden page for testing formatting.
CHAPTERS = [
  ['main'  , nil                      , :generateMain          ],
  ['00'    , 'Getting Started'        , :generateSetup         ],
  ['01'    , 'Numbers'                , :generateNumbers       ],
  ['02'    , 'Letters'                , :generateLetters       ],
  ['03'    , 'Variables and Assignment', :generateVariables    ],
  ['04'    , 'Mixing It Up'           , :generateConversion    ],
  ['05'    , 'More About Methods'     , :generateMethods       ],
  ['06'    , 'Flow Control'           , :generateFlowControl   ],
  ['07'    , 'Arrays and Iterators'   , :generateArrays        ],
  ['08'    , 'Writing Your Own Methods', :generateDefMethod    ],
  ['09'    , 'Classes'                , :generateClasses       ],
  ['10'    , 'Blocks and Procs'       , :generateBlocksProcs   ],
  ['11'    , 'Beyond This Tutorial'   , :generateBeyond        ],
  ['format', 'Formatting Page'        , :generateFormattingPage],
]


CHAPTERS.each do |chap, title, meth|
  page = LearnToProgramTutorial.new
  out  = page.generate(chap, title, meth)

  filename = "#{chap}.html"
  filename = 'chap_'+filename if chap < 'A'
  filename = File.expand_path(File.join(File.dirname(__FILE__), filename))

  puts "Writing #{filename}..."
  File.write(filename, out)
end

puts "Done!"
```