<!DOCTYPE html> <!-- The above 3 meta tags *must* come first in the head; any other head content must come *after* these tags -->

<!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media queries --> <!-- WARNING: Respond.js doesn't work if you view the page via file:// --> <!--[if lt IE 9]> <![endif]-->

The Ruby on Rails Tutorial

: Korean Edition, Draft

Welcome, 임재룡(R팀) | 목차 | 원문

```
  | <a href="/signout">Log Out</a>
</div>
```

```
<ul id='chapter_navigator' class='sticker'>
  <li class=""><a href="frontmatter.html">Front matter</a> </li>
  <li class=""><a href="beginning.html">Chapter 1: From zero to deploy</a> </li>
  <li class=""><a href="toy_app.html">Chapter 2: A toy app</a> </li>
  <li class=""><a href="static_pages.html">Chapter 3: Mostly static pages</a> </li
>
  <li class="active">Chapter 4: Rails-flavored Ruby </li>
  <li class=""><a href="filling_in_the_layout.html">Chapter 5: Filling in the layo
ut</a> </li>
  <li class=""><a href="modeling_users.html">Chapter 6: Modeling users</a> </li>
  <li class=""><a href="sign_up.html">Chapter 7: Sign up</a> </li>
  <li class=""><a href="log_in_log_out.html">Chapter 8: Log in, log out</a> </li>
  <li class=""><a href="updating_and_deleting_users.html">Chapter 9: Updating, sho
wing, and deleting users</a> </li>
  <li class=""><a href="account_activation_password_reset.html">Chapter 10: Accoun
t activation and password reset</a> </li>
  <li class=""><a href="user_microposts.html">Chapter 11: User microposts</a> </li
>
  <li class=""><a href="following_users.html">Chapter 12: Following users</a> </li
>
</ul>
```

<!DOCTYPE html>

```
<title>ruby_on_rails_tutorial</title>

  <script type="text/x-mathjax-config">
          MathJax.Hub.Config({
    "HTML-CSS": {
      availableFonts: ["TeX"],
    },
    TeX: {
      extensions: ["AMSmath.js", "AMSsymbols.js", "color.js"],
      equationNumbers: {
        autoNumber: "AMS",
        formatNumber: function (n) { return "4" + '.' + n }
      },
      Macros: {
        PolyTeX:    "Poly{\\TeX}",
        PolyTeXnic: "Poly{\\TeX}nic",
        "failing": "\\coloredtext{red}{\\textsc{\\textbf{red}}}",
```

"passing": "\coloredtext{ForestGreen}{\textsc{\textbf{green}}}" } }, showProcessingMessages: false, messageStyle: "none", imageFont: null });

```
  </script>
  <script type="text/javascript" src="MathJax/MathJax.js?config=TeX-AMS-MML_SVG"><
/script>
```

```
<div id="book">
  <div id="cha-rails_flavored_ruby" data-tralics-id="cid23" class="chapter" data-n
umber="4" data-chapter="rails_flavored_ruby">
```

Chapter 4 레일스용 루비

3장의 예시에 기초하여, 이 장에서는 레일스에서 중요하게 사용하는 루비의 몇가지 요소(독자들의 이해를 돕기 위해서 “레일스용 루비” 로 의역함, 역자 주)를 자세히 살펴보겠다. 루비는 중요한 언어이지만, 다행히 레일스 개발자 입장에서 볼 때 루비언어로 작업할 부분은 비교적 적다. 루비언어 소개에서 다루는 대부분의 주제와는 다소 차이가 있다. 4장의 목적은 레일스용 루비의 기초를 튼튼히 하는데 있으며, 과거 프로그래밍 언어 경험과는 무관한다. 여기서는 여러가지를 소개하기 때문에, 한번에 모두 이해하지 못해도 괜찮다. 나중에 다른 장에서 여기를 종종 참고하겠다.

4.1 동기

앞에서 보았듯이, 루비 언어에 대한 배경지식이 없이도 레일스 애플리케이션의 기본 틀을 만들고 테스트할 수 있었다. 튜토리얼에 있는 테스트 코드를 따라서 실행하고 일련의 테스트가 통과될 때까지 에러 메시지를 처리하였다. 이렇게 계속 할 수 없어서, 이번 장을 통해서 루비에 대한 부족한 부분을 사이트에 추가하기로 했다.

앞서 새로 만든 애플리케이션에서는, 뷰에서 중복되는 부분을 제거하기 위해, 목록 4.1과 같이 레일스 레이아웃을 사용하여 정적 페이지를 업데이트했다. (목록 3.32도 같다).

목록 4.1:

Sample 애플리케이션 사이트 레이아웃.app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
    <%= stylesheet_link_tag    'application', media: 'all',
                                              'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

목록 4.1에 있는 아래의 코드라인에 집중하자.

```
<%= stylesheet_link_tag 'application', media: 'all',
                                       'data-turbolinks-track' => true %>
```

레일스의 내장 함수인 `stylesheet_link_tag` (레일스 API에서 자세히 읽어 볼 수 있다) 메소드를 사용해서 모든 미디어 타입의 (컴퓨터 화면과 프린터 포함) `application.css` 파일을 포함한다. 레일스를 사용해 본 개발자에게, 이 코드라인은 특별해 보일 것이 없지만, 언듯 이해하기 어려운 루비 문법 네가지가 숨어 있다. 레일스 내장 메소드, 괄호를 생략한 메소드 호출방식, 심볼, 해시. 여기서 이러한 문법을 모두 다룰 것이다.

뷰에서 많은 내장 함수를 사용할 수 있을 뿐만 아니라 사용자 정의 함수를 만들 수도 있다. 그런 함수를 *헬퍼*라고 한다. 커스텀 헬퍼를 만드는 방법을 알아보기 위해, 목록 4.1의 *title* 코드라인부터 보도록 하겠다.

```
<%= yield(:title) %> | Ruby on Rails Tutorial Sample App
```

아래에서 보는 바와 같이, 이 *title* 값은 각각의 뷰마다 (`provide` 메소드를 사용하여) 정의하는 페이지 타이틀에 따라 달라진다.

```
<% provide(:title, "Home") %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

그런데 provide 메소드에 타이틀이 없으면 어떻게 될까? 이럴 경우에는, 모든 페이지에서 사용할 *기본 타이틀*을 두고 페이지 제목을 다른 것으로 지정하고 싶을 때 추가하는 방식이 좋을 것이다. 현재 레이아웃 설정에 따라 *거의* 다 구성하였다. 보다시피 특정 뷰에서 `provide` 를 호출하지 않을 경우, 그 페이지에 해당하는 특정 제목이 없어서 전체 제목이 다음과 같이 보이게 될 것이다.

```
  | Ruby on Rails Tutorial Sample App
```

다시 말해, *기본 타이틀*이 있지만, 맨 앞에 수직 막대 문자 `|` 가 붙게 된다.

페이지 타이틀이 누락되어 생기는 문제를 고치기 위해, `full_title` 이라는 커스텀 헬퍼 메소드를 정의하겠다. `full_title` 헬퍼 메소드는 페이지 타이틀을 따로 정의 하지 않았을 경우 “Ruby on Rails Tutorial Sample App” 이라는 기본 타이틀을 반환하고, 페이지 타이틀을 하나라도 정의해 두었으면 페이지 타이틀 뒤에 수직 막대를 붙이도록 한다 (목록 4.2).[1]

목록 4.2: `full_title` 헬퍼를 정의한다.app/helpers/application_helper.rb

```
module ApplicationHelper

  # Returns the full title on a per-page basis.
  def full_title(page_title = '')
    base_title = "Ruby on Rails Tutorial Sample App"
    if page_title.empty?
      base_title
    else
      "#{page_title} | #{base_title}"
    end
  end
end
```

헬퍼 메소드를 만든 후, 목록 4.3와 같이, 아래의 레이아웃을

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

다음과 같이

```
<title><%= full_title(yield(:title)) %></title>
```

변경하면 코드가 한결 간결해진다.

목록 4.3: `full_title` 헬퍼와 사이트 레이아웃.**green**app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <%= stylesheet_link_tag    'application', media: 'all',
                                              'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

헬퍼 메소드가 제대로 동작하기 위해서, 홈 페이지에서는 “Home” 이라는 단어가 필요없기 때문에 생략할 경우 기본 타이틀을 반환할 수 있도록 한다. 우선 목록 4.4의 테스트 코드를 수정하여, 앞서 페이지 타이틀에 대한 테스트한 코드를 수정하여 `"Home"` 이라는 문자열이 없는지 테스트하는 코드를 추가한다.

목록 4.4: 새로 갱신한 홈 페이지 타이틀에 대한 테스트 코드.**red**test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase
  test "should get home" do
    get :home
    assert_response :success
    assert_select "title", "Ruby on Rails Tutorial Sample App"
  end

  test "should get help" do
    get :help
    assert_response :success
    assert_select "title", "Help | Ruby on Rails Tutorial Sample App"
  end

  test "should get about" do
    get :about
    assert_response :success
    assert_select "title", "About | Ruby on Rails Tutorial Sample App"
  end
end
```

테스트를 실행한 후 테스트 중 하나가 실패했음을 확인한다.

목록 4.5: **red**

```
$ bundle exec rake test
3 tests, 6 assertions, 1 failures, 0 errors, 0 skips
```

일련의 테스트가 통과하려면, 목록4.6 같이 홈 페이지 뷰에서 `provide` 메소드가 있는 줄을 삭제해야 한다.

목록 4.6: 홈 페이지에 타이틀을 추가하지 않음.**green**app/views/static_pages/home.html.erb

```
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

이제 테스트를 통과해야 한다.

목록 4.7: **green**

```
$ bundle exec rake test
```

(*노트*: 앞서 `rake test` 실행한 결과 일부를 표시하였는데, 테스트를 통과하고 실패한 개수를 보여주었으나, 이제부터는 페이지를 절약하기 위해 테스트 통과 및 실패 개수를 생략하겠다.)

애플리케이션 스타일시트를 포함하는 코드라인과 같이, 목록 4.2의 코드는 노련한 레일스 개발자의 눈에 간결해 보일 수도 있지만, 언듯 이해하기 어려운 루비 언어의 개념이 *많다*. 모듈, 메소드 정의, 메소드 파라미터 생략, 주석, 지역 변수 할당, 불린형(논리형), 흐름 제어, 문자열 끼워넣기, 반환값. 이번 장에서는 이러한 개념을 설명한다.

4.2 문자열과 메소드

루비 학습에 주로 사용할 도구는 *레일스 콘솔*인데, 레일스 애플리케이션과 상호 작용하는 커맨드라인 툴이며 2.3.3에서 처음 소개한 바 있다. 콘솔 자체는 인터랙티브 루비(`irb`)를 토대로 만들어서, 모든 루비 언어를 바로 사용할 수 있다. (4.4.4절에서 알게 되겠지만, 콘솔은 레일스 환경변수를 읽어 온다.)

클라우드 IDE를 사용하는 경우, irb 설정시 포함하도록 추천하는 파라미터 두 개는 아래를 참고하기 바란다. `nano` 텍스트 편집기를 사용해서, 홈 디렉토리에 있는 `.irbrc` 파일에 아래의 목록 4.8을 복사해서 붙여넣는다.

```
$ nano ~/.irbrc
```

목록 4.8은 irb 프롬프트를 짧게 처리하고 제멋대로 들여쓰는 자동 들여쓰기 기능을 끈다.

목록 4.8: irb 설정 추가. ~/.irbrc

```
IRB.conf[:PROMPT_MODE] = :SIMPLE
IRB.conf[:AUTO_INDENT_MODE] = false
```

목록 4.8 설정을 추가한 후, 아래와 같이 커맨드라인에서 콘솔을 실행한다.

```
$ rails console
Loading development environment
>>
```

아무런 옵션을 지정하지 않으면 *개발 환경*모드로 콘솔을 시작하고 개발환경은 레일스에서 세가지로 구분한다(세 가지 중 다른 것은 *테스트*환경과 *운영*환경). 이 장에서 환경 모드를 구분하지 않으나, 보다 자세한 내용은 7.1.1절에서 다루겠다.

콘솔은 훌륭한 학습 도구이므로, 마음껏 구석 구석 사용해 보라. 걱정할 필요 없다. (아마도) 멈추지 않을 것이다. 콘솔을 사용할 때, 문제가 발생할 경우 Ctrl-C를 입력하거나 Ctrl-D 를 누르면 콘솔을 빠져 나갈 수 있다. 이 장의 나머지 부분에서는, 루비 API를 참조하는 것이 도움이 될 수 있다. 루비 API는 많은 정보가 들었 있다.(*너무* 많은 건지도 모른다.) 예를 들면, 루비의 문자열을 더 배우고 싶으면 `String` 클래스에 대해서 루비 API를 참조할 수 있다.

4.2.1 주석

루비 *주석*은 파운드 기호 `#` (“해시 마크” 또는 “넘버’ 라고도 함’)로 시작하고 해당 코드라인의 맨 끝까지 주석처리한다. 주석은 루비가 무시하지만 사람들이 편하게 읽을 수 있다.(저자 포함!). 아래의 코드에서

```
# 페이지마다 타이틀을 반환한다.
def full_title(page_title = '')
  .
  .
  .
end
```

첫 번째 코드라인 주석은 아래 함수의 목적을 나타낸다.

원래 콘솔 세션에서는 주석을 포함하지 않지만 여기에서는 설명할 목적으로 아래와 같이 주석을 삽입하겠다.

```
$ rails console
>> 17 + 42   # 정수 덧셈
=> 59
```

이번 절을 따라하면서 콘솔에서 직접 키보드를 치거나 단축키로 복사해서 붙여 넣을 때, 주석을 생략해도 좋다. 콘솔은 항상 주석을 처리하지 않는다.

4.2.2 문자열

*문자열*은 웹 애플리케이션의 가장 중요한 자료 구조라고 보는데 웹 페이지는 결국 서버가 브라우저로 보내는 여러 문자들로 구성되기 때문이다. 콘솔에서 문자열을 살펴보자.

```
$ rails console
>> ""          # 빈 문자열
=> ""
>> "foo"       # 비어 있지 않은 문자열
=> "foo"
```

이들은 큰 따옴표 `"` 를 써서 만든 *문자열 리터럴* (*리터럴 문자열*이라고도 함)이다. 콘솔은 코드라인마다 처리하여 결과를 출력하는데, 이 경우에 문자열 리터럴은 단지 문자열을 출력한다.

문자열을 `+` 연산자로 합칠 수 있다.

```
>> "foo" + "bar"     # 문자열 합치기
=> "foobar"
```

`"foo"` 더하기 `"bar"` 를 처리한 결과 `"foobar"` 라는 문자열이다.**("라는 문자열이 만들어졌다." 가 더 자연스러운 것 같습니다.)**[2]

문자열을 만드는 다른 방법은 *내삽법(중간에 끼어넣기, interpolation)*이라고 해서 `#{}` 이라는 특별한 구문을 사용한다:[3]

```
>> first_name = "Michael"     # 변수에 할당한다
=> "Michael"
>> "#{first_name} Hartl"      # 문자열을 끼워 넣는다
=> "Michael Hartl"
```

여기서 `"Michael"` 이라는 값을 `first_name` 이라는 변수에 *할당*한 뒤 `"#{first_name} Hartl"` 에 문자열을 끼워 넣었다. 물론 두 개의 문자열을 하나의 변수 이름으로 할당할 수 있다.

```
>> first_name = "Michael"
=> "Michael"
>> last_name = "Hartl"
=> "Hartl"
>> first_name + " " + last_name     # 문자열을 공백으로 붙여 넣기
=> "Michael Hartl"
>> "#{first_name} #{last_name}"     # 문자열에 끼워 넣는 다른 방법
=> "Michael Hartl"
```

마지막 두 코드라인은 같은 내용인데, 저자는 문자열에 삽입하는 방법을 선호한다. 빈칸 `" "` 을 붙여 넣는 것이 불편해 보이기 때문이다.

문자열 출력

문자열을 *출력*하려면, `puts` 라는 루비 함수를 사용한다("풋 에스", "풋 스트링" 이라고 읽음):

```
>> puts "foo"     # 문자열 출력
foo
=> nil
```

`puts` 메소드는 *부작용*이 있다. `puts "foo"` 는 화면에 문자열을 출력하고 존재하지 않는다고 반환한다. `nil` 은 "아무것도 아니다"라는 특별한 루비 값이다. (이후에는 지면을 절약하기 위해 `=> nil` 부분을생략할 것이다.)

앞선 예시에서 보듯이, `puts` 메소드를 사용하면 자동으로 개행문자 \n 가 추가된다. 그러나 비슷한 종류의 `print` 메소드의 경우는 개행 문자를 추가하지 않는다.

```
>> print "foo"    # 문자열 출력.(puts 메소드와 달리 다음 줄로 커서를 옮기지 않음)
foo=> nil
>> print "foo\n"  # puts "foo" 실행결과와 같다
foo
=> nil
```

작은 따옴표 문자열

지금까지 모든 예시에서 *큰 따옴표*로 문자열을 사용했지만, 루비는 *작은 따옴표 문자열*을 지원한다. 많은 경우에서, 문자열의 두 가지 유형은 결과적으로 동일한 것이다.

```
>> 'foo'          # 작은 따옴표를 쓴 문자
=> "foo"
>> 'foo' + 'bar'
=> "foobar"
```

그런데 중요한 차이점이 있다. 루비는 작은 따옴표로 둘러싼 문자열 안에 내삽법을 허용하지 않는다.

```
>> '#{foo} bar'     # 작은 따옴표를 쓴 문자열에 끼워 넣을 수 없다.
=> "\#{foo} bar"
```

주의할 점은 콘솔이 큰 따옴표를 사용한 문자열로 값을 반환하는 것인데, 이 때 큰 따옴표 문자열은 `#{` 와 같은 특수 문자를 *이스케이프(예외 처리)*하기 위해 백슬래시를 써야 한다.

큰 따옴표 문자열은 작은 따옴표 문자열로 할 수 있는 모든 기능을 가지면서 내삽법까지 사용할 수 있는데, 작은 따옴표 문자열은 어떤 측면에서 중요할까? 작은 따옴표 문자열은 글자 그대로, 즉, 타이핑하는 글자들을 있는 있는 그대로 문자열 안에 포함할 때 종종 유용하다. 예를 들어, "백슬래시" 글자는 시스템에서 대부분 특별하게 쓰이는데, 개행 문자를 \n 과 같이 쓴다. 이와 같이 변수에 백슬래시가 붙은 글자를 넣을 경우는, 작은 따옴표가 더 용이하다.

```
>> '\n'       # '백슬래시 n' 특수 문자 표현
=> "\\n"
```

앞선 예시의 `#{` 글자와 같이, 루비는 백슬래시를 한번 더 붙여서 백슬래시 자체를 이스케이프(예외 처리)해야 한다. 따라서, 큰 따옴표 문자열 안에서 백슬래시는 백슬래시 *두 개*로 표기해야 한다. 이처럼 짧은 예시에서는, 타이핑 수를 별로 줄이

지 못해도, 이스케이프(예외 처리)할 글자가 많을 경우에는 정말 요긴하게 사용할 수 있다.

```
>> '개행문자 (\n)와 탭 (\t)은 백슬래시 문자 \를 붙여 쓴다.'
=> "개행문자 (\\n)와 탭 (\\t)은 백슬래시 문자 \\를 붙여 쓴다.."
```

결국, 작은 따옴표와 큰 따옴표를 바꿔서 써도 문제가 없어야 하는 경우에, 소스코드에서 일정한 패턴없이 큰/작은 따옴표를 혼재해서 사용하는 것을 종종 볼 수 있다는 것은 주목할만하다. 할 수 있는 것은 이 말 밖에 없다. “루비에 잘 오셨습니다!" (그 만큼 루비에는 자유로움이 있다는 의미. 역자 주)

4.2.3 객체와 메시지 보내기

루비에서는 모든 것이, 문자열을 포함해서 심지어 `nil` 도, *객체*다. 이것의 기술적 의미를 4.4.2절에서 알게 되겠지만, 누구나 책에서 그 정의를 읽었다고 해도 객체를 완전히 이해할 수는 없다. 객체에 대한 통찰력을 높이기 위해서는 다양한 사례를 겪어봐야 한다.

객체가 *하는 일을* 설명하기는 더 쉬운데, 그것은 메시지에 대해서 응답을 보내는 것이다. 예를 들어 문자열 같은 객체는 `length` 라는 메시지를 처리하여, 문자열의 글자 수를 반환한다.

```
>> "foobar".length       # 길이 "length"를 요청하는 메시지를 문자열 객체에게 보낸다.
=> 6
```

일반적으로, 객체로 보내는 메시지는 *메소드*라고 하며, 해당 객체 안에서 정의한 함수다. [4] 문자열은 `empty?` 라는 메시지를 받아서 메소드를 처리한다.

```
>> "foobar".empty?
=> false
>> "".empty?
=> true
```

물음표가 `empty?` 메소드의 맨 뒤에 있다는 점을 주의한다. 이러한 표기는 루비의 관습으로 반환하는 값이 *불린(논리값)* (`true` 또는 `false`)임을 알려준다. 불린(논리값)은 *흐름 제어*시 특히 유용하다.

```
>> s = "foobar"
>> if s.empty?
>>   "The string is empty"
>> else
>>   "The string is nonempty"
>> end
=> "The string is nonempty"
```

조건을 하나 더 붙이려면, `elsif` (`else` + `if`)를 쓸 수 있다.

```
>> if s.nil?
>>   "The variable is nil"
>> elsif s.empty?
>>   "The string is empty"
>> elsif s.include?("foo")
>>   "The string includes 'foo'"
>> end
=> "The string includes 'foo'"
```

불린(논리값)은 `&&` ("and"), `||` ("or"), 그리고 `!` ("not") 연산자를 조합하여 사용할 수 있다.

```
>> x = "foo"
=> "foo"
>> y = ""
=> ""
>> puts "Both strings are empty" if x.empty? && y.empty?
=> nil
>> puts "One of the strings is empty" if x.empty? || y.empty?
"One of the strings is empty"
=> nil
>> puts "x is not empty" if !x.empty?
"x is not empty"
=> nil
```

루비의 모든 것은 객체라는 명제에 따라, `nil` 역시 객체이기 때문에, 사용할 수 있는 메소드가 있다. 한가지 예시는 `to_s` 메소드인데 모든 객체를 문자열로 변환할 수 있다.

```
>> nil.to_s
=> ""
```

이렇게 보면 비어있는 문자열이지만, `nil` 객체에 보내는 메시지를 *체이닝(chaining, '.'을 이어서 쓰기)*하면 몇가지를 검증할 수 있다.

```
>> nil.empty?
NoMethodError: undefined method `empty?' for nil:NilClass
>> nil.to_s.empty?      # 메시지 체이닝
=> true
```

여기서 보듯이 `nil` 이라는 객체는 `empty?` 메소드를 처리하지 못하지만, `nil.to_s` 는 처리할 수 있다.

`nil` 인지 테스트하는 특별한 메소드가 있는데, 쉽게 생각할 수 있는 것들이다.

```
>> "foo".nil?
=> false
>> "".nil?
=> false
>> nil.nil?
=> true
```

아래와 같은 코드는

```
puts "x is not empty" if !x.empty?
```

`if` 키워드를 다른 형태로 사용하는 예를 보여준다. `if` 뒤의 문장이 참일 때만 루비가 스크립트를 처리하는 문장을 작성할 수 있다. 참이 아닐 때 스크립트를 처리하는 `unless` 키워드도 같은 방식으로 동작한다.

```
>> string = "foobar"
>> puts "The string '#{string}' is nonempty." unless string.empty?
The string 'foobar' is nonempty.
=> nil
```

`false` 를 제외하고, 불린(논리값) 상태를 평가해야 할 상황에서 거짓값을 나타내는, 유일한 루비 객체라는 점에서, `nil` 객체가 특별하다는 점은 주목할만한 가치가 있다. `!!` ("뱅 뱅"으로 읽음)을 앞에 붙여서 실행해보면, 객체의 값을 두 번 바꾸어 결국 불린(논리값) 값으로 처리하게 된다는 것을 알 수 있다.

```
>> !!nil
=> false
```

특히, 다른 모든 객체는 0 일지라도 *참* 값을 나타낸다.

```
>> !!0
=> true
```

4.2.4 메소드 정의

목록 3.6의 `home` 액션과 목록 4.2의 `full_title` 헬퍼 메소드를 정의할 때와 같이 콘솔에서도 메소드를 정의할 수 있다. (콘솔에서 메소드를 정의하는 방법은 데모를 보여줄 때 편리하지만, 타이핑하기 번거러워서 대개 파일형태로 불러온다.) 예를 들어, 하나의 인수를 가지는 `string_message` 라는 함수를 정의하여 인수의 유무에 따라 각기 다른 메시지를 반환하는 함수를 정의해 보자.

```
>> def string_message(str = '')
>>   if str.empty?
>>     "It's an empty string!"
>>   else
>>     "The string is nonempty."
>>   end
>> end
=> :string_message
>> puts string_message("foobar")
The string is nonempty.
>> puts string_message("")
It's an empty string!
>> puts string_message
It's an empty string!
```

위에서 보듯이, 인수가 없어도 함수를 실행할 수 있고 심지어 괄호를 생략할 수도 있다. 이러 일이 가능한 이유는, 함수의 인수가

```
def string_message(str = '')
```

기본값을 가지기 때문인데, 이 경우에는 빈 문자열이다. 따라서 `str` 인수를 꼭 쓰지 않아도 되며, 인수를 생략하면 인수는 미리 정해둔 기본 값을 갖는다.

루비에서 함수는 *return* 구문을 생략할 수 있고, 대신에 마지막에 처리한 문장을 반환한다. 이 경우에는, 메소드 인수인 `str` 이 비어 있는지에 따라서 메시지 문자열 두 개 중에서 하나만을 반환한다. 루비는 return 구문을 명시적으로 표시할 수 있다. 아래의 함수는 위와 동일하다.

```
>> def string_message(str = '')
>>   return "It's an empty string!" if str.empty?
>>   return "The string is nonempty."
>> end
```

(오류를 잘 찾는 독자는 두 번째 `return` 은 실제로 불필요하다는 것을 눈치챘을 것이다. 함수의 마지막 문장인, `"The string is nonempty."` 문자열은 `return` 키워드가 없어도 반환되지만, `return` 을 두 군데에서 표시하여 좋게 보이게 한 것 뿐이다.)

함수 인수의 이름은 호출하는 것과 무관하다고 이해하는 게 중요하다. 즉, 위의 첫번째 예시에서 `str` 이라는 이름을 `the_function_argument` 와 같이 다른 유효한 이름으로 바꿔도 동일하게 동작한다.

```
>> def string_message(the_function_argument = '')
>>   if the_function_argument.empty?
>>     "It's an empty string!"
>>   else
>>     "The string is nonempty."
>>   end
>> end
=> nil
>> puts string_message("")
It's an empty string!
>> puts string_message("foobar")
The string is nonempty.
```

4.2.5 타이틀 헬퍼로 돌아가서

이제 목록 4.2의 `full_title` 헬퍼 메소드를 이해하게 되었고,[5] 이 목록에 주석을 표시하여 목록 4.9과 같이 보이도록 한다.

목록 4.9: `title_helper` 에 주석을 표시함. app/helpers/application_helper.rb

```
module ApplicationHelper

  # 해당 페이지에 전체 타이틀을 반환한다.                    # 문서 주석
  def full_title(page_title = '')                          # 인수(arg)를 생략할 수 있는 메소드
 정의(def)
    base_title = "Ruby on Rails Tutorial Sample App"       # 변수 할당
    if page_title.empty?                                   # 불린(논리값) 검사
      base_title                                           # return 키워드는 생략 가능
    else
      "#{page_title} | #{base_title}"                      # 문자열에 끼워 넣기
    end
  end
end
```

함수 정의(선택적 인수를 포함), 변수 할당, 불린(논리값) 검사, 흐름 제어, 문자열 내삽법 등과 같은 요소들을 함께 사용하여 컴팩트한 헬퍼 메소드를 만들고 사이트 레이아웃에서 사용한다. 마지막 요소는 `module ApplicationHelper` 다. 모듈은 서로 관련 있는 메소드를 패키지로 묶어주고, `include` 구문으로 루비 클래스에서 *함께 사용할 수 있다*. 루비 코드를 작성할 때, 모듈을 쓰려면 직접 넣어줘야 하지만, 레일스의 헬퍼 메소드의 경우에는 레일스가 알아서 넣어준다. 결과적으로 `full_title` 메소드는 모든 뷰 안에서 추가조치없이 바로 사용할 수 있다.

4.3 그 밖의 자료 구조

웹앱이란 결국 문자열에 관한 것이지만, 실제로 그런 문자열을 *만들 때에는* 다른 형태의 자료구조가 필요하다. 이번 절에서, 레일스 애플리케이션 작성하는데 중요한 몇가지 루비 자료 구조를 배울 것이다.

4.3.1 배열과 범위

배열은 순서가 있는 요소들의 목록이다. 아직 *레일스 튜토리얼*에서 살펴보지 않았지만, 배열을 이해하면 해시(4.3.3절)와 레일스 데이타 모델링 부분(2.3.3 절에서 `has_many` 관계를 언급했고, 11.1.3절에서 더 자세히 다룬다)을 이해하는데 밑거름이 된다.

지금까지 문자열을 이해하는데 많은 시간을 할애했다. `split` 메소드를 사용하면 문자열을 배열로 자연스럽게 변경할 수 있다.

```
>>  "foo bar     baz".split     # 문자열을 3개로 분리하여 배열로 만든다.
=> ["foo", "bar", "baz"]
```

실행결과 3개의 문자열 요소를 가지는 배열이 만들어졌다. 기본적으로 `split` 메소드는 문자열을 공백문자마다 분리하여 배열로 나누지만, 다른 문자를 사용하여 분리할 수도 있다.

```
>> "fooxbarxbazx".split('x')
=> ["foo", "bar", "baz"]
```

많은 컴퓨터 언어의 통상적인 관념처럼 루비 배열은 *zero-offset*, 순번이 0부터 시작하여 배열 첫번째 요소의 인덱스가 0, 두번째 요소는 1 이 되는 것이다.

```
>> a = [42, 8, 17]
=> [42, 8, 17]
>> a[0]                 # 루비는 대괄호로 배열 요소를 읽는다.
=> 42
>> a[1]
=> 8
>> a[2]
=> 17
>> a[-1]                # 인덱스는 음수도 된다!
=> 17
```

위와 같이 루비는 대괄호를 사용하여 배열 요소를 읽는다. 또한 루비는 이 대괄호 표기법 뿐만 아니라, 더 자주 불러오는 배열요소에 대해서는 별도의 메소드를 제공한다.[6]

```
>> a                    # 'a' 를 저장한다.
=> [42, 8, 17]
>> a.first
=> 42
>> a.second
=> 8
>> a.last
=> 17
>> a.last == a[-1]      # 등호 두개를 붙여서 비교합니다. ==
=> true
```

마지막 코드라인에서는 다른 언어에서도 공통으로 볼 수 있는 동일 비교 연산자 ==, 그리고 이와 연관해서 `!=` (“같지 않음”) 같은 연산자도 있다.

```
>> x = a.length       # 문자열처럼 배열도 'length' 메소드가 있다.
=> 3
>> x == 3
=> true
>> x == 1
=> false
>> x != 1
=> true
>> x >= 1
=> true
>> x < 1
=> false
```

`length` (위의 첫번째 코드라인) 메소드 외에도 여러가지 다른 메소드들도 사용할 수 있다.

```
>> a
=> [42, 8, 17]
>> a.empty?
=> false
>> a.include?(42)
=> true
>> a.sort
=> [8, 17, 42]
>> a.reverse
=> [17, 8, 42]
>> a.shuffle
=> [17, 42, 8]
>> a
=> [42, 8, 17]
```

위 메소드는 모두 `a` 자체의 값을 바꾸지 않는다는 것에 주의해야 한다. 배열의 값을 *바꾸려면* “뱅” 메소드(느낌표를 “뱅”으로 읽음)를 사용해야 한다.

```
>> a
=> [42, 8, 17]
>> a.sort!
=> [8, 17, 42]
>> a
=> [8, 17, 42]
```

배열 마지막에 요소를 넣을 때는 `push` 메소드 외에도 << 연산자를 사용할 수 있다.

```
>> a.push(6)                    # 배열에 6 을 푸시한다.
=> [42, 8, 17, 6]
>> a << 7                       # 배열에 7 을 푸시한다.
=> [42, 8, 17, 6, 7]
>> a << "foo" << "bar"          # 배열에 체인 방식으로 푸시한다.
=> [42, 8, 17, 6, 7, "foo", "bar"]
```

마지막 예에서 보듯이 푸시 두개를 연이어 쓸 수 있고, 다른 언어의 배열과 달리 루비 배열은 자료형이 달라도 같은 배열에 요소로 추가할 수 있다. (여기서는 정수와 문자열)

앞서 `split` 메소드가 문자열을 배열로 바꾸는 예를 보았다. 반대로 배열을 문자열로 바꾸는 방법으로 `join` 메소드가 있다.

```
>> a
=> [42, 8, 17, 7, "foo", "bar"]
>> a.join                          # 공백없이 모두 요소를 연결한다.
=> "428177foobar"
>> a.join(', ')                    # 콤마와 공백으로 모든 요소를 연결한다.
=> "42, 8, 17, 7, foo, bar"
```

배열과 매우 비슷한 자료구조로 *범위(Range)*가 있고 범위를 배열로 바꾸는 `to_a` 메소드를 사용하면 범위를 쉽게 이해할 수 있다.

```
>> 0..9
=> 0..9
>> 0..9.to_a               # 앗 이런, 숫자 9의 to_a 메소드를 실행했다.
NoMethodError: undefined method `to_a' for 9:Fixnum
>> (0..9).to_a             # 괄호를 써서 범위의 to_a 메소드를 실행한다.
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

범위를 `0..9` 로 표기하는 것은 맞지만, 위와 같이 두번째 코드라인의 메소드를 실행할 때는 범위표기를 괄호로 둘러싸야 한다.

범위는 배열 요소를 가져올 때도 유용하다.

```
>> a = %w[foo bar baz quux]          # %w 표기를 사용하여 문자열 배열을 생성함
=> ["foo", "bar", "baz", "quux"]
>> a[0..2]
=> ["foo", "bar", "baz"]
```

특히 유용한 방법은 -1을 인덱스 범위의 끝으로 지정하면 배열의 시작부터 끝까지 모든 요소를 선택할 수 있어 배열의 크기를 *표시하지* 않아도 된다.**(이미 이 사용법을 알고 있다면 이해가 되는 문장이나, 초보자가 이해하기에 설명이 부족? 하다고 생각됩니다.)**

```
>> a = (0..9).to_a
=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>> a[2..(a.length-1)]                # 배열의 크기를 명시적으로 표시
=> [2, 3, 4, 5, 6, 7, 8, 9]
>> a[2..-1]                          # 인덱스를 -1 로 표시하는 방법
=> [2, 3, 4, 5, 6, 7, 8, 9]
```

범위는 문자에도 사용할 수 있다.

```
>> ('a'..'e').to_a
=> ["a", "b", "c", "d", "e"]
```

4.3.2 블록

배열과 범위에는 블록을 파라미터로 받아서 사용하는 메소드가 많다. *블록*은 루비의 가장 강력한 기능 중 하나이지만 한편으로는 가장 혼란스러운 기능이기도 합니다.

```
>> (1..5).each { |i| puts 2 * i }
2
4
6
8
10
=> 1..5
```

위의 코드는 `each` 메소드가 범위 `(1..5)`에 대하여 하나씩 블록 `{ |i| puts 2 * i }`을 처리하도록 해준다. `|i|` 에서 변수명을 둘러싼 수직 막대는 루비 구문에서 블록 변수를 나타내고, 블록을 어떻게 처리할 지는 메소드에 달여 있다. 여기서는 범위 `each` 메소드는 하나의 지역변수(`i`)로 블록을 처리할 수 있기 때문에, 범위 내의 각 값에 대해서 각각 블록을 실행한다.

중괄호를 사용하여 블록을 표시하는 방법 외에도 아래와 같은 다른 방법도 있다.

```
>> (1..5).each do |i|
?>   puts 2 * i
>> end
2
4
6
8
10
=> 1..5
```

블록은 한 줄을 넘을 수 있고, 대체로 그렇다. *레일스 튜토리얼*에서는 통념상 중괄호는 짧고 한줄짜리 블록에서만 쓰고 `do..end` 구문은 한 줄로 작성하기엔 길거나 블록 내용이 여러 줄일 경우에 사용할 것이다.

```
>> (1..5).each do |number|
?>   puts 2 * number
>>   puts '--'
>> end
2
--
4
--
6
--
8
--
10
--
=> 1..5
```

여기서는 변수명으로 어떤 이름도 사용할 수 있다는 것을 강조하기 위해서 `i` 대신 `number` 를 사용하였다.

실제 프로그래밍 경험이 없다면 블록을 이해하는데는 왕도가 없다. 블록을 자주 접하게 되면 결국 익숙해 질 것이다.[7] 다행히 인간은 구체적인 사례로부터 일반화를 잘 하기 때문에, `map` 메소드를 사용하는 2가지 사례를 살펴보자.

```
>> 3.times { puts "Betelgeuse!" }   # 3.times 는 블록 변수가 없다.
"Betelgeuse!"
"Betelgeuse!"
"Betelgeuse!"
=> 3
>> (1..5).map { |i| i**2 }          # ** 는 '거듭제곱'.
=> [1, 4, 9, 16, 25]
>> %w[a b c]                        # %w 는 문자열 배열로 만들어 준다.
=> ["a", "b", "c"]
>> %w[a b c].map { |char| char.upcase }
=> ["A", "B", "C"]
>> %w[A B C].map { |char| char.downcase }
=> ["a", "b", "c"]
```

위의 예에서 `map` 메소드는 해당 블록을 범위 내의 각 값에 적용하여 그 결과 값을 배열로 반환하였다. 마지막 두 예에서 `map` 메소드는 블록 내에서 블록 변수에 대해서 특정한 메소드를 실행했으며, 다음과 같이 단축폼을 사용할 수도 있다.

```
>> %w[A B C].map { |char| char.downcase }
=> ["a", "b", "c"]
>> %w[A B C].map(&:downcase)
=> ["a", "b", "c"]
```

(*심볼*을 사용하여 단축 폼을 사용할 경우 코드가 낯설어 보이는데, 심볼에 대해서는 4.3.3 절에서 살펴보겠다.) 이런 것은 루비온레일스에 처음부터 적용되었고, 사람들이 그 점을 매우 만족했기 때문에 루비 코어에 반영되었다고 한다.

블록의 다음 사례로, 목록 4.4 의 테스트에서 볼 수 있다:

```
test "should get home" do
  get :home
  assert_response :success
  assert_select "title", "Ruby on Rails Tutorial Sample App"
end
```

코드를 자세히 몰라도 되지만(그리고 *저자* 역시 자세히는 모른다), 본문에 `do` 키워드가 나타나면 블록이라는 것을 알 수 있다. `test` 메소드는 인자로 문자열(여기서는 설명문)과 블록을 받고, 블록 안의 내용을 테스트 한다.

한편, 이 시점에서 1.5.4 절의 도메인을 임의로 만드는 루비 코드를 되짚고 넘어가겠다.

```
('a'..'z').to_a.shuffle[0..7].join
```

실행 과정을 나눠서 살펴본다:

```
>> ('a'..'z').to_a                    # 알파벳 배열
=> ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o",
"p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]
>> ('a'..'z').to_a.shuffle             # 알파벳 순서를 섞는다.
=> ["c", "g", "l", "k", "h", "z", "s", "i", "n", "d", "y", "u", "t", "j", "q",
"b", "r", "o", "f", "e", "w", "v", "m", "a", "x", "p"]
>> ('a'..'z').to_a.shuffle[0..7]      # 처음부터 8번째 요소까지 가져온다.
=> ["f", "w", "i", "a", "h", "p", "c", "x"]
>> ('a'..'z').to_a.shuffle[0..7].join  # 배열의 요소를 묶어서 문자열로 만든다.
=> "mznpybuj"
```

4.3.3 해시와 심볼

해시는 본질적으로 배열이지만 각 요소를 접근하는 인덱스를 정수 외에 다른 것으로도 사용할 수 있다. (실제로, 어떤 언어에서는, 펄(Perl) 언어를 꼬집어 말하면, 이와 같은 이유로 해시를 *연관 배열*이라고 말한다.) 대신에, 거의 모든 객체가 해시 인덱스(또는 *키*)가 될 수 있다. 예를 들어, 문자열을 키로 사용할 수 있다.

```
>> user = {}                          # {} 는 비어 있는 해시.
=> {}
>> user["first_name"] = "Michael"     # 키는 "first_name", 값은 "Michael"
=> "Michael"
>> user["last_name"] = "Hartl"        # 키는 "last_name", 값은 "Hartl"
=> "Hartl"
>> user["first_name"]                 # 배열처럼 요소를 가져온다.
=> "Michael"
>> user                               # 해당 해시를 출력한다
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}
```

해시는 중괄호로 감싸고 키-값 쌍을 요소로 추가한다. 키-값 쌍이 없는 괄호(`{}`)는 비어 있는 해시다. 주의할 것은 해시의 중괄호가 블록의 중괄호가 아니라는 점이다. (물론, 이 점 때문에 혼란스러울 수 있다.) 해시가 배열처럼 보이지만 중요한 차이점 한 가지는, 해시의 경우 일반적으로 특정 순서로 자신의 요소를 유지하지 않는다는 점이다.[8] 순서가 중요하다면, 배열을 사용한다.

꺾쇠괄호를 사용하여 한번에 한개 항목씩 추가하여 해시를 정의하기 보다는, 키와 값들을 "해시로켓" 이라고 하는 `=>` 로 구분해서 리터럴로 표시하는 것이 쉽다.

```
>> user = { "first_name" => "Michael", "last_name" => "Hartl" }
=> {"last_name"=>"Hartl", "first_name"=>"Michael"}
```

여기에서는 루비의 관례에 따라 해시의 시작과 끝 괄호에 공백을 넣었다. 그러나 콘솔은 관례와 무관하게 출력한다. (빈칸이 왜 관례인지 묻지 않길 바란다. 아마도 초창기 영향력 있는 루비 프로그래머가 앞뒤로 넣은 공백이 보기에 좋아서 관례로 굳어진 듯 하다.)

여기까지는 문자열을 해시키로 사용했지만, 레일스는 *심볼*을 더 자주 사용한다. 심볼은 문자열처럼 보이지만, 인용부호로 감싸지 않고 앞에만 콜론을 붙인다. 예를 들어, `:name` 은 심볼이다. 심볼은 군더더기 없는 문자열로 생각할 수 있다.[9]

```
>> "name".split('')
=> ["n", "a", "m", "e"]
>> :name.split('')
NoMethodError: undefined method `split' for :name:Symbol
>> "foobar".reverse
=> "raboof"
>> :foobar.reverse
NoMethodError: undefined method `reverse' for :foobar:Symbol
```

심볼은 특별한 루비 자료형이고 다른 언어에서는 보기 드물어 낯설지만 레일스에서는 자주 사용하기 때문에 금방 익숙해진다. 문자열과 달리 모든 글자를 심볼로 사용할 수 없다.

```
>> :foo-bar
NameError: undefined local variable or method `bar' for main:Object
>> :2foo
SyntaxError
```

심볼을 알파벳 글자로 시작하고 단어에서 사용할 수 있는 문자로 제한한다면, 오류는 발생하지 않는다.

해시 키로 사용하는 심볼에 대하여, `user` 해시는 아래와 같이 정의할 수 있다.

```
>> user = { :name => "Michael Hartl", :email => "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
>> user[:name]              # :name 에 해당하는 값을 가져온다.
=> "Michael Hartl"
>> user[:password]          # 정의하지 않은 키의 값을 가져 온다.
=> nil
```

마지막 예에서 정의하지 않은 키의 해시 값은 그냥 `nil` 이다.

해시가 심볼을 키로 사용하는 것이 보편적이라서, 루비 1.9는 새로운 구문을 지원하여 다음과 같은 특별한 경우가 가능하다.

```
>> h1 = { :name => "Michael Hartl", :email => "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
>> h2 = { name: "Michael Hartl", email: "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
>> h1 == h2
=> true
```

두 번째 구문은 심볼/해시로켓을 조합하여 사용하는 대신 키 이름 뒤에 콜론과 값을 차례대로 둔다.

```
{ name: "Michael Hartl", email: "michael@example.com" }
```

이러한 구조는 다른 언어(자바스크립트와 같은)의 해시 표기 방법에 더 가까워서 레일스 커뮤니티에서 자주 사용하는 사람들이 늘어가고 있다. 해시 구문 두 가지 모두 여전히 일반적으로 사용하기 때문에, 반드시 구분할 수 있어야 한다. 안타깝게도, 머릿속이 복잡해질 수 있는데, 특히 `:name` 심볼은 자체만으로(독립된 심볼로써)도 유효한 구문이지만 `name:` 은 자체만으로는 유효하지 않기때문이다. 마지막 줄에서 `:name =>` 과 `name:` 은 사실상 *리터럴 해시* 안에서만 같아서,

```
{ :name => "Michael Hartl" }
```

그리고

```
{ name: "Michael Hartl" }
```

는 똑같지만, 그 외의 경우에는 `:name` (콜론이 앞에 붙은) 형식과 같이 심볼을 표기한다.

해시 값으로는 사실상 거의 모든 것이 될 수 있고, 목록 4.10와 같이 다른 해시가 해시 값으로 사용될 수도 있다.

목록 4.10: 중첩 해시.

```
>> params = {}        # 'params' 라는 해시를 정의한다 ('parameters'의 줄임말).
=> {}
>> params[:user] = { name: "Michael Hartl", email: "mhartl@example.com" }
=> {:name=>"Michael Hartl", :email=>"mhartl@example.com"}
>> params
=> {:user=>{:name=>"Michael Hartl", :email=>"mhartl@example.com"}}
>>  params[:user][:email]
=> "mhartl@example.com"
```

이러한 유형의 해시-속의-해시, 또는 *중첩 해시*는 레일스에서 매우 자주 사용하며, 7.3 절을 시작할 때 볼 수 있다.

배열과 범위 같이, 해시와 동작하는 `each` 메소드가 있다. 예를 들어, `flash` 라는 이름의 해시가 `:success` 와

`:danger` 라는 두 가지 조건에 대해 키를 가진다고 생각해 보자.

```
>> flash = { success: "It worked!", danger: "It failed." }
=> {:success=>"It worked!", :danger=>"It failed."}
>> flash.each do |key, value|
?>   puts "Key #{key.inspect} has value #{value.inspect}"
>> end
Key :success has value "It worked!"
Key :danger has value "It failed."
```

주의할 것은 배열의 `each` 메소드는 블록 변수가 한 개지만, 해시의 `each` 메소드는 블록 변수가두 개(*키*와 *값*)다. 따라서, 해시의 `each` 메소드가 한번에 키-값 *한 쌍* 해시를 반복처리한다.

마지막 예에서 유용한 `inspect` 메소드를 사용했는데, 이 메소드는 해당 객체를 호출할 때 리터럴 자체를 그대로 표현하는 문자열을 반환한다.

```
>> puts (1..5).to_a            # 배열을 문자열로 출력한다.
1
2
3
4
5
>> puts (1..5).to_a.inspect    # 배열을 리터럴로 출력한다.
[1, 2, 3, 4, 5]
>> puts :name, :name.inspect
name
:name
>> puts "It worked!", "It worked!".inspect
It worked!
"It worked!"
```

한편, `inspect` 메소드를 사용해서 객체를 출력하는 것이 일반적이나 이 때 단축형인 `p` 함수를 대신 사용할 수도 있다:[10]

```
>> p :name             # 'puts :name.inspect' 와 같다.
:name
```

4.3.4 CSS 재검토

목록4.1 로 돌아가서 캐스캐이딩 스타일 시트를 포함하기 위해 레이아웃에서 사용한 코드 라인을 다시 살펴보자.

```
<%= stylesheet_link_tag 'application', media: 'all',
                                       'data-turbolinks-track' => true %>
```

이제는 위의 코드가 무엇인지 거의 알 것 같다. 4.1 절에서 간략히 살펴본 것처럼 레일스는 스타일시트를 포함하기 위해서

특별한 함수를 정의한 후

```
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track' => true
```

이 함수를 호출한다. 그러나 여기에 몇가지 궁금한 점이 있다. 먼저, 괄호는 어디에 있을까? 루비에서 괄호는 생략할 수 있다.

```
# 함수 호출할 때 괄호는 생략할 수 있다.
stylesheet_link_tag('application', media: 'all',
                                   'data-turbolinks-track' => true)
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track' => true
```

다음으로, 인수 `media` 는 해시가 분명한데 중괄호는 어디에 있을까? 함수를 호출할 때 *맨 끝의* 인수가 해시일 경우, 중괄호를 생략할 수 있으며, 따라서 아래 두 줄은 동일한 것이다.

```
# 마지막 해시 인수의 중괄호는 생략할 수 있다.
stylesheet_link_tag 'application', { media: 'all',
                                     'data-turbolinks-track' => true }
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track' => true
```

다음으로, `data-turbolinks-track` 라는 키/값 쌍은 왜 예전 방식으로 해시로켓 구문을 사용하지 않을까? 새로운 해시 구문을 사용하는 것이

```
data-turbolinks-track: true
```

유효하지 않은 이유는 하이픈 때문이다. (4.3.3 절에서 보면 하이픈은 심볼로 쓸 수 없다.) 따라서 예전 문법을 사용할 수 밖에 없으며, 다음과 같이 표시한다.

```
'data-turbolinks-track' => true
```

마지막으로, 아래와 같은 코드가

```
stylesheet_link_tag 'application', media: 'all',
                                   'data-turbolinks-track' => true
```

마지막 인수인 해시 내에서 줄이 바뀌어도 문법적으로 틀지지 않는 이유는 무엇일까? 루비는 개행문자든 여러 개 공백문자든 다르게 구분하지 않기 때문이다. 여기서 코드를 한 줄에 쓰지 않은 [11] *이유는* 소스 코드를 라인당 80자 이내로 작성하여 가독성을 유지하기 위해서다.[12]

그럼, 아래의 코드라인을 보면,

```
stylesheet_link_tag 'application', media: 'all',
                                  'data-turbolinks-track' => true
```

`stylesheet_link_tag` 함수를 호출할 때 두 개의 인수를 사용한다. 첫번째 문자열은 스타일시트의 경로를 가리키고, 두 개의 요소를 포함하는 해시는 미디어 타입을 지정하고 레일스 4의 새로운 기능인 터보링크를 사용하도록 한다. <%= %> 괄호 때문에 ERB가 결과를 템플릿 안에 삽입해 준다. 웹브라우저로 페이지 소스보기를 하면 스타일시트(목록 4.11)를 포함하는 HTML 태그를 볼 수 있어야 한다. (CSS 파일 이름 뒤에 붙은 몇가지 항목을 볼 수 있는데, `?body=1` 와 같다. 이것은 레일스가 추가한 것으로, 브라우저가 CSS파일을 캐시하여 저장하더라도 서버에 있는 CSS 파일 내용이 바뀌면 브라우저가 다시 읽어오도록 해 준다.)

목록 4.11: CSS가 만든 HTML 소스.

```
<link data-turbolinks-track="true" href="/assets/application.css" media="all"
rel="stylesheet" />
```

실제 CSS 파일을 보기 위해 이 주소(http://localhost:3000/assets/application.css)로 이동하면, 내용이 (주석 몇 줄을 제외하고) 비어 있는 것을 알 수 있다. 5 장에서 이 파일을 변경할 것이다.

4.4 루비 클래스

앞서 언급했듯이 루비에서 모든 것은 객체다. 이번 절에서는 마침내 직접 객체를 정의해 볼 것이다. 많은 객체 지향 언어처럼 루비는 *클래스*를 사용하여 메소드를 구성하며, 이 클래스를 *실체화*하여 객체를 생성하게 된다. 객체 지향 프로그램을 처음 접하는 독자의 이해를 돕기 위해 몇 가지 예를 들어 보겠다.

4.4.1 생성자

클래스를 사용하여 객체를 실체화하는 많은 예를 보기만 했지만, 직접 명시적으로 그렇게 해야만 한다. 예를 들어, 문자열을 실체화할 때 문자열의 *리터럴 생성자*로 이중 따옴표 문자를 사용했다.

```
>> s = "foobar"        # 문자열이 리터럴 생성자로 이중 따옴표를 사용한다.
=> "foobar"
>> s.class
=> String
```

예를 보면 문자열의 `class` 메소드를 실행하면, 바로 해당 클래스를 반환한다.

리터럴 생성자를 사용하는 방법 대신에, *네임드(이름을 가지는) 생성자*를 사용할 수 있는데, 해당 클래스 이름 뒤에 `new` 메소드를 호출한다.[13]

```
>> s = String.new("foobar")   # 문자열의 네임드 생성자
=> "foobar"
>> s.class
=> String
>> s == "foobar"
=> true
```

리터럴 생성자와 동일하지만 어떤 기능을 하는지 명시적으로 알 수 있다.

배열도 문자열처럼 생성한다.

```
>> a = Array.new([1, 3, 2])
=> [1, 3, 2]
```

그러나, 해시를 생성하는 방법은 다르다. 배열 생성자 `Array.new` 메소드는 배열의 초기값을 인수로 넘겨 받는 반면, `Hash.new` 메소드는 아직 정의되지 않은 키에 대한 *기본 값을* 넘겨 받는다.

```
>> h = Hash.new
=> {}
>> h[:foo]             # 키 :foo의 값를 읽어 온다.
=> nil
>> h = Hash.new(0)     # 키가 없을 경우 nil 대신 0을 반환하도록 정한다.
=> {}
>> h[:foo]
=> 0
```

`new` 메소드처럼 클래스 이름 뒤에 메소드를 붙여서 호출하는 경우를 *클래스 메소드*라고 부른다. 클래스의 `new` 메소드를 호출하면 클래스 객체를 반환하는데, 클래스 *인스턴스*라고 한다. `length` 메소드와 같이 인스턴스 뒤에 메소드를 붙여서 호출하면 *인스턴스 메소드*라고 부른다.

4.4.2 클래스 상속

클래스에 대해 배울 때, *클래스 계층 구조*를 파악하는 방법으로 `superclass` 메소드를 사용할 수 있다.

```
>> s = String.new("foobar")
=> "foobar"
>> s.class                          # s의 클래스를 확인.
=> String
>> s.class.superclass               # String 클래스의 상위 클래스를 확인.
=> Object
>> s.class.superclass.superclass  # Ruby 1.9 는 베이스 클래스가 BasicObject.
=> BasicObject
>> s.class.superclass.superclass.superclass
=> nil
```

그림 4.1에는 상속 계층 구조의 다이어그램이 있다. `String` 의 상위 클래스는 `Object` 이고 `Object` 의 상위 클래스는 `BasicObject` 이지만, `BasicObject` 는 상위 클래스가 없다. 이런 패턴은 모든 루비 객체에 적용된다. 클래스 계층 구조로 되돌아가 보면 루비의 모든 클래스는 최상위 클래스인 `BasicObject` 에서 상속받는다. 이것은 "루비의 모든 것은 객체"라는 기술적 의미다.

그림 4.1: `String` 클래스의 상속 계층.

클래스를 좀 더 깊이 이해하려면 직접 클래스를 만드는 것만한 것이 없다. `Word` 클래스에 `palindrome?` 메소드를 만들어 보겠다. 이메소드는 단어의 철자를 앞으로 뒤로 읽어도 같은 단어일 경우 `true` 를 반환한다.

```
>> class Word
>>   def palindrome?(string)
>>     string == string.reverse
>>   end
>> end
=> :palindrome?
```

아래와 같이 사용할 수 있다.

```
>> w = Word.new                 # Word 객체를 생성한다.
=> #<Word:0x22d0b20>
>> w.palindrome?("foobar")
=> false
>> w.palindrome?("level")
=> true
```

이 예가 다소 억지스러운 느낌을 준다면 제대로 된 것이다. 일부러 그런 것이다. 단지 문자열을 인수로 갖는 메소드를 생성하기 위해 새로운 클래스를 만들어 한다면 자연스럽지 못하다. 목록 4.12와 같이, 단어는 문자열*이기 때문에*, `Word` 클래스가 `String` 클래스를 *상속*받는 것은 당연하다. (이미 정의한 `Word` 클래스를 루비 콘솔에서 지우려면 콘솔을 종료한 후 다시 들어가야 한다.)

목록 4.12: 콘솔에서 `Word` 클래스를 정의한다.

```
>> class Word < String              # Word 클래스는 String에서 상속 받는다.
>>   # 문자열을 거꾸로 해도 같으면 true 를 반환한다.
>>   def palindrome?
>>     self == self.reverse        # self 는 문자열을 가리킨다.
>>   end
>> end
=> nil
```

여기서 `Word < String` 는 루비의 상속 구문이고 (3.2절 참고), 단어는 `palindrome?` 메소드 외에도, 문자열의 모든 메소드를 갖게 된다.

```
>> s = Word.new("level")    # Word를 새로 만들고, "level"로 초기화한다.
=> "level"
>> s.palindrome?            # 단어를 가리키는 Word 클래스는 palindrome? 메소드가 있다.
=> true
>> s.length                 # 단어를 가리키는 Word 클래스는 또한 일반적인 문자열 메소드를 상속받는
다.
=> 5
```

`Word` 클래스는 `String` 클래스에서 상속받았기 때문에, 콘솔에서 클래스 계층을 살펴볼 수 있다.

```
>> s.class
=> Word
>> s.class.superclass
=> String
>> s.class.superclass.superclass
=> Object
```

계층 구조는 그림 4.2와 같다.

Figure 4.2: 목록 4.12의 `Word` 클래스의 상속 계층 구조.

목록 4.12에서, 단어가 거꾸로 읽어도 같은지 확인하기 위해 `Word` 클래스 내에서 해당 단어에 접근 할 수 있어야 함을 주목한다. 루비에서 `self` 키워드를 사용하여 이 문제를 해결할 수 있다 . `Word` 클래스 안에서, `self` 는 객체 자신을 가리키기 때문에 아래와 같이

```
self == self.reverse
```

단어가 앞뒤로 바꿔도 같은지 확인한다. [14] 실제로, String 클래스 안에서 `self.` 는 메소드와 속성 앞에서 생략해도 된다. 따라서

```
self == reverse
```

처럼 쓸 수 있다.

4.4.3 내장 클래스 변경하기

상속이 강력하게 추천되지만, 회문(回文, 펠린드롬)의 경우에는 `palindrome?` 메소드를 `String` 클래스 메소드로 추가하는 것이 훨씬 자연스러워 보이는데, 이렇게 하면 문자열 리터럴에 대해서 다른 메소드 중에서도 `palindrome?` 메소드를 호출할 수 있게 된다. 지금 당장은 호출할 수 없다.

```
>> "level".palindrome?
NoMethodError: undefined method `palindrome?' for "level":String
```

놀랍게도, 루비는 다음과 같이 할 수 있도록 해준다. 루비 클래스는 *다시 열어* 변경할 수 있기 때문에, 누구나 메소드를 기존 클래스에 추가할 수 있다.

```
>> class String
>>   # 해당 문자열이 자신의 reverse 값과 같은 경우 true를 반환한다.
>>   def palindrome?
>>     self == self.reverse
>>   end
>> end
=> nil
>> "deified".palindrome?
=> true
```

(루비에서 내장 클래스에 메소드를 추가할 수 있다는 것과 `"deified"` 란 단어가 회문이라는 것 중에 어떤 것이 더 멋진 것인지 모르겠다.)

내장 클래스를 변경하는 기술은 강력하지만, 능력이 많으면 그 만큼 많은 책임이 뒤따르듯이, *타당한* 이유없이 내장 클래스에 메소드를 추가하는 것은 좋지 않다. 레일스는 몇가지 그럴만한 이유가 있다. 예를 들어, 웹 애플리케이션에서 변수가 *블랭크*가 되면 안되는 경우가 종종 있는데, 예를 들어, 사용자 이름은 공백 또는 다른 화이트스페이스 외의 다른 값을 가져야 한다. 따라서 레일스는 `blank?` 메소드를 루비에 추가했다. 레일스 콘솔에서는 레일스가 확장한 기능을 자동으로 불러오기 때문에, 아래와 같은 결과를 볼 수 있다. (그냥 `irb` 를 실행하면 결과가 다르게 나온다.)

```
>> "".blank?
=> true
>> "      ".empty?
=> false
>> "      ".blank?
=> true
>> nil.blank?
=> true
```

공백 문자열은 *비어있진* 않지만 *블랭크*다. `nil` 이 블랭크임을 주의해라. `nil` 은 문자열이 아니기 때문에, 레일스가 실제로 `blank?` 메소드를 `String` 클래스의 부모 클래스인, (이번 절의 시작에서 본 것처럼) `Object` 클래스에 추가했다는 것을 암시해 준다. 8.4 절에서 루비 클래스에 레일스 메소드를 추가한 사례를 볼 것이다.

4.4.4 콘트롤러 클래스

클래스와 상속에 관한 이야기를 하면, 이전에 본적이 있기 때문에, 목록 3.18의 StaticPages 콘트롤러에 대한 기억이 번득 떠오른다.

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

이제 어렴풋하게 나마 위의 코드가 무엇인지 눈에 들어 올 수 있다. `StaticPagesController` 는 `ApplicationController` 을 상속받고, `home` , `help` , 그리고 `about` 메소드를 정의한다. 레일스 콘솔 세션마다 로컬 레일스 환경을 로드하기 때문에, 콘트롤러를 직접 생성하고 해당 클래스의 계층 구조를 찾아 볼 수 있다:[15]

```
>> controller = StaticPagesController.new
=> #<StaticPagesController:0x22855d0>
>> controller.class
=> StaticPagesController
>> controller.class.superclass
=> ApplicationController
>> controller.class.superclass.superclass
=> ActionController::Base
>> controller.class.superclass.superclass.superclass
=> ActionController::Metal
>> controller.class.superclass.superclass.superclass.superclass
=> AbstractController::Base
>> controller.class.superclass.superclass.superclass.superclass.superclass
=> Object
```

이 계층 구조의 다이어그램은 그림 4.3과 같다..

그림 4.3: Static Pages 컨트롤러의 상속 계층 구조

콘트롤러 액션조차 메소드이기 때문에 콘솔에서 호출할 수 있다:

```
>> controller.home
=> nil
```

여기서 반환값이 `nil` 인 것은 `home` 액션이 비어 있기 때문이다.

그러나 액션의 반환값이 없다는 것은 그리 중요하지 않다. `home` 액션의 중요한 점은 3장에서 처럼 웹페이지를 렌더링하는 것이지 값을 반환하는 것이 아니다. 어디서 `StaticPagesController.new` 를 호출하는지 모른다고해서 그게 무슨 큰일이라도 되겠는가?

그게 무슨 큰 일이겠냐는 말은 레일스는 루비로 *작성*했지만, 레일스가 곧 루비가 아니라는 것이다. 레일스 클래스 일부는 보통 루비 객체처럼 사용하지만, 그러나 어떤 클래스는 단지 레일스라는 마법같은 방앗간에서 제분용으로 사용하는 곡식(그리스트)일 뿐이다. 레일스는 매우 독특해서(*수이 제너리스*), 루비와 별도로 공부해야 한다.

4.4.5 사용자 클래스

루비 여행의 마지막으로 `User` 클래스를 작성하고, 6장에서 User 모델을 다시 언급할 것이다.

지금까지는 콘솔에서 클래스를 정의하여 입력했지만, 금방 귀찮아진다;. 대신에 `example_user.rb` 라는 파일을 애플리케이션 루트 디렉토리에 만들고 목록 4.13의 내용을 입력한다.

목록 4.13: 사용자 예시 코드 example_user.rb

```
class User
  attr_accessor :name, :email

  def initialize(attributes = {})
    @name  = attributes[:name]
    @email = attributes[:email]
  end

  def formatted_email
    "#{@name} <#{@email}>"
  end
end
```

여기서 조금 더 설명할 것이 있기 때문에, 단계별로 나누도록 하겠다. 첫번째 코드라인에서,

```
  attr_accessor :name, :email
```

사용자의 이름과 이메일 주소에 해당하는 *속성 접근자*를 생성한다. 속성 접근자는 “getter” 와 “setter” 메소드를 생성하여 `@name` 과 `@email` 이라는 *인스턴스 변수*를 가져오고(get) 할당(set)하는데, 인스턴스 변수에 대해서는 2.2.2절과 3.6절을 참조하기 바란다. 레일스에서 인스턴스 변수가 매우 중요한 것은 뷰에서 변수를 사용할 수 있다는 것이지만, 보통은 루비 클래스 내에서 필요로 할 때 사용한다. (좀 더 설명하면,) 인스턴스 변수는 항상 `@` 기호로 시작하고, 정의하지 않으면 `nil` 값을 반환한다.

첫 번째로, `initialize` 메소드는 루비의 특별한 메소드다. `User.new` 메소드를 실행할 때 호출되기 때문이다. `initialize` 메소드는 `attributes` 라는 인수가 한 개 있다.

```
  def initialize(attributes = {})
    @name  = attributes[:name]
    @email = attributes[:email]
  end
```

여기서 `attributes` 변수는 *기본 값이* 빈 해시라서 이름과 주소가 없는 사용자를 정의할 수 있다. (4.3.3 절을 참조하

면 해시는 키가 없을 때 `nil` 을 반환하므로, `attributes[:name]` 이 `nil` 일때는 `:name` 이라는 키가 없을 때이고, `attributes[:email]` 도 같다.)

끝으로, 클래스에서 `formatted_email` 이라는 메소드를 정의하여 `@name` 과 `@email` 변수에 할당한 값을 문자열 내삽법(4.2.2절)으로 사용자 이메일 주소의 포맷 버전을 만든다.

```
  def formatted_email
    "#{@name} <#{@email}>"
  end
```

`@name` 과 `@email` 모두 인스턴스 변수 (`@` 기호로 표시)이므로, `formatted_email` 메소드에서 사용할 수 있다.

콘솔을 실행하여 사용자 예시 코드를 `require` 하고 User 클래스를 살펴보자.

```
>> require './example_user'      # 이렇게 하여 example_user 코드를 불러 들인다.
=> true
>> example = User.new
=> #<User:0x224ceec @email=nil, @name=nil>
>> example.name                  # attributes[:name] 가 nil이라서 nil이 된다.
=> nil
>> example.name = "Example User"            # nil이 아닌 이름으로 대입한다
=> "Example User"
>> example.email = "user@example.com"       # nil이 아닌 이메일 주소를 대입한다
=> "user@example.com"
>> example.formatted_email
=> "Example User <user@example.com>"
```

여기서 `'.'` 은 유닉스에서 “현재 디렉토리”이며, `'./example_user'` 은 현재 위치에서 example user 파일을 상대 경로로 찾는다. 뒤 따라 오는 코드는 변수 이름이 example인 user를 만들고 아직 내용은 비어있기때문에 이름과 이메일 주소에 해당하는 속성을 직접 할당한다.(목록 4.13에서 `attr_accessor` 코드을 사용했기 때문에 변수에 값을 할당할 수 있음). 다음과 같이

```
example.name = "Example User"
```

루비는 `@name` 변수에 `"Example User"` (`email` 도 유사하게)로 할당하고 `formatted_email` 메소드를 호출한다.

4.3.4절에서 맨 뒤에 오는 해시 인수는 중괄호를 생략해도 되므로, 다음과 같이 `initialize` 메소드 뒤에 해시 인수를 넘겨 주는 식으로 속성 값을 미리 할당한 user 객체를 생성하므로써 또 다른 사용자를 만들 수 있다.

```
>> user = User.new(name: "Michael Hartl", email: "mhartl@example.com")
=> #<User:0x225167c @email="mhartl@example.com", @name="Michael Hartl">
>> user.formatted_email
=> "Michael Hartl <mhartl@example.com>"
```

7 장에서 객체를 해시 인수로 초기화하는 것을 보게 되는데, 이러한 기법은 레일스 애플리케이션에서 흔히 볼 수 있으며 *대량할당*으로 알려져 있다.

4.5 결론

이 정도로 루비 언어에 대한 전반적인 내용을 정리한다. 5장부터는 여기서 배운 것을 바탕으로 sample 애플리케이션 개발을 시작할 것이다.

4.4.5 절부터는 `example_user.rb` 파일을 사용하지 않을 것이기 때문에 파일을 삭제한다.

```
$ rm example_user.rb
```

파일 삭제 후 메인 소스 코드 저장소에 지금까지 작성한 파일을 커밋하고, 비트버킷에 푸시한 후 허로쿠로 배포한다.

```
$ git status
$ git commit -am "Add a full_title helper"
$ git push
$ bundle exec rake test
$ git push heroku
```

4.5.1 이번 장에서 배운 것

- 루비는 문자열을 조작할 수 있는 메소드가 많이 있다.
- 루비의 모든 것은 객체다.
- 루비는 `def` 키워드로 메소드를 정의한다.
- 루비는 `class` 키워드로 클래스를 정의한다.
- 레일스 뷰는 HTML 중간에 임베디드 루비(ERb)로 넣을 수 있다.
- 내장 루비 자료구조로는 배열, 범위, 해시가 있다.
- 루비 블록은 (다른 무엇보다도) 유연한 구조여서 배열같은 열거형 자료 구조에서 코드를 반복 실행할 수 있다.
- 심볼은 문자열처럼 이름으로 사용하지만, 별도의 자료구조가 없다.
- 루비는 객체 상속을 지원한다.
- 내장 루비 클래스를 열어서 변경할 수 있다.
- “deified”라는 단어는 일종의 회문이다.

4.6 연습문제

- www.railstutorial.org를 방문하여 레일스 튜토리얼을 구매하면 연습문제에 대한 해답을 구할 수 있다.
- 레일스 튜토리얼 이메일 리스트에 등록하면 보너스로 첫번째 toy 앱에 대한 레일스 튜토리얼 치트시트 카드를 얻게 된다.

1. 목록 4.14에서 보이는 물음표 위치를 다른 메소드로 변경하여 주어진 문자열의 글자를 임의로 섞는 함수를 `split`, `shuffle`, 및 `join` 메소드로 작성한다.

2. 목록 4.15처럼 `String` 클래스에 `shuffle` 메소드를 추가한다.
3. 이름과 성을 `:first` 와 `:last` 키로 가지는 해시 3개 `person1`, `person2`, and `person3` 를 만든다. `params[:father]` 은 `person1`, `params[:mother]` 은 `person2`, `params[:child]` 은 `person3` 인 `params` 해시를 만든다. `params[:father][:first]` 은 올바른 값인지 검증하는 예를 보여준다.
4. 루비 API 온라인 버전에서 해시 클래스의 `merge` 메소드를 읽는다. 아래의 코드를 실행한 결과는 무엇인가?

```
{ "a" => 100, "b" => 200 }.merge({ "b" => 300 })
```

목록 4.14: 문자열을 섞는 함수 코드

```
>> def string_shuffle(s)
>>   s.?('').?.?
>> end
>> string_shuffle("foobar")
=> "oobfra"
```

목록 4.15: `String` 클래스에 추가한 `shuffle` 메소드

```
>> class String
>>   def shuffle
>>     self.?('').?.?
>>   end
>> end
>> "foobar".shuffle
=> "borafo"
```

1. 헬퍼 메소드를 특정 컨트롤러에서만 사용한다면, 해당 헬퍼 파일에 두어야 한다. 예를 들어, Static Pages 컨트롤러의 헬퍼 메소드는 `app/helpers/static_pages_helper.rb` 파일 내에 위치해야 한다. 여기서는 사이트 페이지마다 `full_title` 헬퍼를 사용할 것이기 때문에, 레일스는 이러한 경우를 위해 특별한 헬퍼 파일을 사용한다. `app/helpers/application_helper.rb` . ↥
2. “foo” 와 “bar”에 대한 유래를 알고자 한다면—특히 “foobar” *와 무관한* “FUBAR”에 대해서—약어 파일에서 “foo” 항목을 찾아본다. ↥
3. 프로그래머가 펄(Perl) 또는 PHP에 익숙하다면 `"foo $bar"` 와 같은 표현식에서 달러 기호가 붙은 변수를 자동으로 삽입(내삽법)하는 것과 비교해야 한다. ↥
4. 이번 장에서 *함수*와 *메소드*를 임의로 혼용하여 사용하고 있다는 것을 미리 알려 둔다. 둘 다 같은 것인데, 루비 내의 모든 것이 객체이므로, 모든 메소드는 함수이며 모든 함수는 메소드이기 때문이다. ↥
5. 여전히 이해하지 못하는 *한* 가지가 남아 있는데 레일스가 내부의 모든 구성요소들을 연결하는 방식이다. 즉, URL을 액션으로 매핑해 주고, 뷰와 다른 곳에서도 `full_title` 헬퍼를 사용할 수 있게 한다. 흥미로운 주제여서 깊이 살펴보면 좋겠지만 레일스를 *사용*할 때 레일스 *동작* 원리를 몰라도 된다. (더 깊이 이해하려면, 오비 페르난데스(Obie Fernandez)의 *Rails 4 Way*를 추천한다.) ↥
6. `second` 메소드는 원래 루비에 포함되지 않았지만, 레일스가 확장한 것이다. 한편, 레일스 콘솔에서는 레일스가 루비에 확장한 기능도 자동으로 불러들이기때문에 이 메소드를 실행할 수 있다. ↥
7. 반면에, 프로그래밍 전문가는 블록이 *클로저*라는 사실을 알게 되므로써 이득을 볼 수 있는데, 클로저는 자료를 넘겨 받

을 수 있는 일회성의 익명 함수를 만한다. ↥

8. 루비 1.9 버전부터는 해시에 요소를 넣은 순서대로 저장하지만, 이 순서에 의존하여 해시 작업을 하려는 것은 현명하지 못할 것이다. ↥
9. 군더더기가 없는 심볼은 서로 비교하기 편하다. 즉, 문자열은 한 글자씩 비교해야하지만 심볼은 숫자처럼 한번에 비교할 수 있다. 심볼이 해시의 키로 주로 사용하는 이유다. ↥
10. 실제로 미묘하게 다른데, `p` 는 출력할 객체를 반환하지만 `puts` 는 항상 `nil` 을 반환한다. 독자 카타르지나 시웨크(Katarzyna Siwek)가 알려주었다. ↥
11. 개행문자는 줄의 맨 뒤에 있기에 줄이 새로 시작됨을 뜻한다. 코드에서, 특수문자 \n 를를 쓴다. ↥
12. 한줄의 칸 수를 *하나씩 세면* 귀찮고 짜증나기도 하지만 대다수 텍스트 편집기의 화면에 칸 수를 보여주는 기능이 있어서 편하다. 예를 들어, 그림1.5으로 돌아가서 오른 쪽에 수직 선이 가늘게 보이는데 코드가 한 줄에 80자씩 작성할 수 있다. 클라우드 통합개발환경(1.2.1)절) 은 기본으로 80자 제한을 보여주는 선을 나타낸다. 텍스트 메이트를 사용하면 View > Wrap Column > 78 로 가면 이 기능을 볼 수 있다. 서브라임텍스트는 View > Ruler > 78 또는 View > Ruler > 80 을 사용할 수 있다. ↥
13. 이러한 결과는 여러분이 사용하는 루비의 버전에 따라 달라질 수 있다. 여기서는 루비 1.9.3 이상을 사용한다고 가정한다. ↥
14. 루비 클래스에서 한가지 더 살펴보기 위해 `self` 키워드에 대해 레일스 팁 블로그의 “루비의 클래스와 인스턴스 변수” 포스트를 참고. ↥
15. 계층 구조를 따라 존재하는 클래스가 어떤 것인지 꼭 알 필요는 없다. *저자*는 이런 클래스들이 무엇을 하는지 모든 채, 2005년부터 루비온레일스를 프로그래밍했다. 이 말은 (a) 자신의 기술력이 매우 부족한 편이던지 (b) 내부구조를 잘 모르는 노련한 레일스 개발자가 되던지 둘 중에 하나다. 후자이기를 바란다. ↥

```
</div>
```