inserted by Rails to ensure that browsers reload the CSS when it changes on the server. Because the hex string is by design unique, your exact version of Listing 4.14 will differ.)

Listing 4.14: The HTML source produced by the CSS includes.

```
<link rel="stylesheet" media="all" href="/assets/application.self-
f0d704deea029cf000697e2c0181ec173a1b474645466ed843eb5ee7bb215794.css?body=1"
data-turbolinks-track="reload" />
```

4.4 Ruby classes

We've said before that everything in Ruby is an object, and in this section we'll finally get to define some of our own. Ruby, like many object-oriented languages, uses *classes* to organize methods; these classes are then *instantiated* to create objects. If you're new to object-oriented programming, this may sound like gibberish, so let's look at some concrete examples.

4.4.1 Constructors

We've seen lots of examples of using classes to instantiate objects, but we have yet to do so explicitly. For example, we instantiated a string using the double quote characters, which is a *literal constructor* for strings:

```
>> s = "foobar"       # A literal constructor for strings using double quotes
=> "foobar"
>> s.class
=> String
```

We see here that strings respond to the method **`class`**, and simply return the class they belong to.

Instead of using a literal constructor, we can use the equivalent *named constructor*, which involves calling the **`new`** method on the class name:[19]

[19]These results will vary based on the version of Ruby you are using. This example assumes you are using Ruby 1.9.3 or later.

```
>> s = String.new("foobar")   # A named constructor for a string
=> "foobar"
>> s.class
=> String
>> s == "foobar"
=> true
```

This is equivalent to the literal constructor, but it's more explicit about what we're doing.

Arrays work the same way as strings:

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

Hashes, in contrast, are different. While the array constructor **`Array.new`** takes an initial value for the array, **`Hash.new`** takes a *default* value for the hash, which is the value of the hash for a nonexistent key:

```
>> h = Hash.new
=> {}
>> h[:foo]            # Try to access the value for the nonexistent key :foo.
=> nil
>> h = Hash.new(0)    # Arrange for nonexistent keys to return 0 instead of nil.
=> {}
>> h[:foo]
=> 0
```

When a method gets called on the class itself, as in the case of **`new`**, it's called a *class method*. The result of calling **`new`** on a class is an object of that class, also called an *instance* of the class. A method called on an instance, such as **`length`**, is called an *instance method*.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. What is the literal constructor for the range of integers from 1 to 10?
2. What is the constructor using the **`Range`** class and the **`new`** method? *Hint*: **`new`** takes two arguments in this context.
3. Confirm using the **`==`** operator that the literal and named constructors from the previous two exercises are identical.

4.4.2 Class inheritance

When learning about classes, it's useful to find out the *class hierarchy* using the **`superclass`** method:

```
>> s = String.new("foobar")
=> "foobar"
>> s.class                          # Find the class of s.
=> String
>> s.class.superclass               # Find the superclass of String.
=> Object
>> s.class.superclass.superclass    # Ruby has a BasicObject base class as of 1.9
=> BasicObject
>> s.class.superclass.superclass.superclass
=> nil
```

A diagram of this inheritance hierarchy appears in Figure 4.1. We see here that the superclass of **`String`** is **`Object`** and the superclass of **`Object`** is **`BasicObject`**, but **`BasicObject`** has no superclass. This pattern is true of every Ruby object: trace back the class hierarchy far enough and every class in Ruby ultimately inherits from **`BasicObject`**, which has no superclass itself. This is the technical meaning of "everything in Ruby is an object".

To understand classes a little more deeply, there's no substitute for making one of our own. Let's make a **`Word`** class with a **`palindrome?`** method that returns **`true`** if the word is the same spelled forward and backward:

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

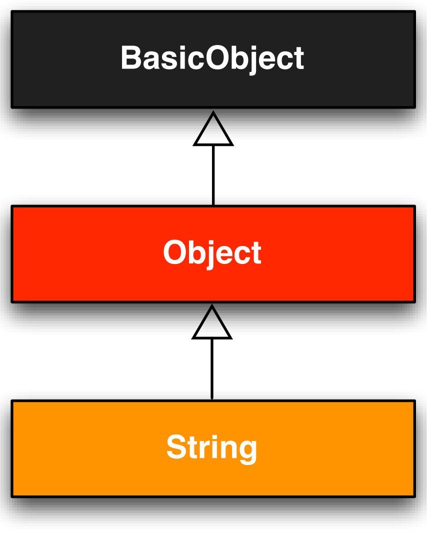

Figure 4.1: The inheritance hierarchy for the **`String`** class.

We can use it as follows:

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

If this example strikes you as a bit contrived, good—this is by design. It's odd to create a new class just to create a method that takes a string as an argument. Since a word *is a* string, it's more natural to have our **`Word`** class *inherit* from **`String`**, as seen in Listing 4.15. (You should exit the console and re-enter it to clear out the old definition of **`Word`**.)

Listing 4.15: Defining a **`Word`** class in the console.

```
>> class Word < String             # Word inherits from String.
>>   # Returns true if the string is its own reverse.
>>   def palindrome?
>>     self == self.reverse        # self is the string itself.
>>   end
>> end
=> nil
```

Here `Word < String` is the Ruby syntax for inheritance (discussed briefly in Section 3.2), which ensures that, in addition to the new `palindrome?` method, words also have all the same methods as strings:

```
>> s = Word.new("level")    # Make a new Word, initialized with "level".
=> "level"
>> s.palindrome?            # Words have the palindrome? method.
=> true
>> s.length                 # Words also inherit all the normal string methods.
=> 5
```

Since the `Word` class inherits from `String`, we can use the console to see the class hierarchy explicitly:

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

This hierarchy is illustrated in Figure 4.2.

In Listing 4.15, note that checking that the word is its own reverse involves accessing the word inside the `Word` class. Ruby allows us to do this using the `self` keyword: inside the `Word` class, `self` is the object itself, which means we can use

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

to check if the word is a palindrome. In fact, inside the String class the use of `self.` is optional on a method or attribute (unless we're making an assignment), so

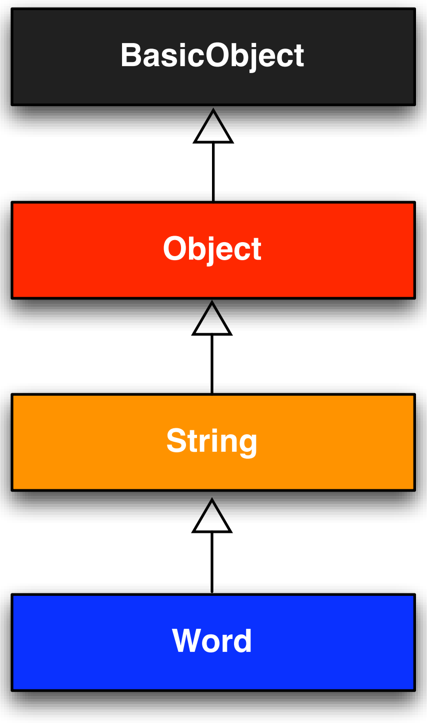

Figure 4.2: The inheritance hierarchy for the (non-built-in) `Word` class from Listing 4.15.

```
self == reverse
```

would work as well.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. What is the class hierarchy for a range? For a hash? For a symbol?
2. Confirm that the method shown in Listing 4.15 works even if we replace **`self.reverse`** with just **`reverse`**.

4.4.3 Modifying built-in classes

While inheritance is a powerful idea, in the case of palindromes it might be even more natural to add the **`palindrome?`** method to the **`String`** class itself, so that (among other things) we can call **`palindrome?`** on a string literal, which we currently can't do:

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

Amazingly, Ruby lets you do just this; Ruby classes can be *opened* and modified, allowing ordinary mortals such as ourselves to add methods to them:

```
>> class String
>>   # Returns true if the string is its own reverse.
>>   def palindrome?
>>     self == self.reverse
>>   end
>> end
=> nil
>> "deified".palindrome?
=> true
```

(I don't know which is cooler: that Ruby lets you add methods to built-in classes, or that `"deified"` is a palindrome.)

Modifying built-in classes is a powerful technique, but with great power comes great responsibility, and it's considered bad form to add methods to built-in classes without having a *really* good reason for doing so. Rails does have some good reasons; for example, in web applications we often want to prevent variables from being *blank*—e.g., a user's name should be something other than spaces and other whitespace—so Rails adds a `blank?` method to Ruby. Since the Rails console automatically includes the Rails extensions, we can see an example here (this won't work in plain `irb`):

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

We see that a string of spaces is not *empty*, but it is *blank*. Note also that `nil` is blank; since `nil` isn't a string, this is a hint that Rails actually adds `blank?` to `String`'s base class, which (as we saw at the beginning of this section) is `Object` itself. We'll see some other examples of Rails additions to Ruby classes in Section 9.1.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Verify that "racecar" is a palindrome and "onomatopoeia" is not. What about the name of the South Indian language "Malayalam"? *Hint*: Downcase it first.

2. Using Listing 4.16 as a guide, add a **`shuffle`** method to the **`String`** class. *Hint*: Refer to Listing 4.12.

3. Verify that Listing 4.16 works even if you remove **`self.`**.

Listing 4.16: Skeleton for a **`shuffle`** method attached to the **`String`** class.

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

4.4.4 A controller class

All this talk about classes and inheritance may have triggered a flash of recognition, because we have seen both before, in the Static Pages controller (Listing 3.22):

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

You're now in a position to appreciate, at least vaguely, what this code means: **`StaticPagesController`** is a class that inherits from **`ApplicationController`**, and comes equipped with **`home`**, **`help`**, and **`about`** methods. Since each Rails console session loads the local Rails environment, we can even create a controller explicitly and examine its class hierarchy:[20]

[20]You don't have to know what each class in this hierarchy does. *I* don't know what they all do, and I've been

```
>> 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
```

A diagram of this hierarchy appears in Figure 4.3.

We can even call the controller actions inside the console, which are just methods:

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

Here the return value is **`nil`** because the **`home`** action is blank.

But wait—actions don't have return values, at least not ones that matter. The point of the **`home`** action, as we saw in Chapter 3, is to render a web page, not to return a value. And I sure don't remember ever calling **`StaticPagesController.new`** anywhere. What's going on?

What's going on is that Rails is *written in* Ruby, but Rails isn't Ruby. Some Rails classes are used like ordinary Ruby objects, but some are just grist for Rails' magic mill. Rails is *sui generis*, and should be studied and understood separately from Ruby.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

programming in Ruby on Rails since 2005. This means either that (a) I'm grossly incompetent or (b) you can be a skilled Rails developer without knowing all its innards. I hope for both our sakes that it's the latter.

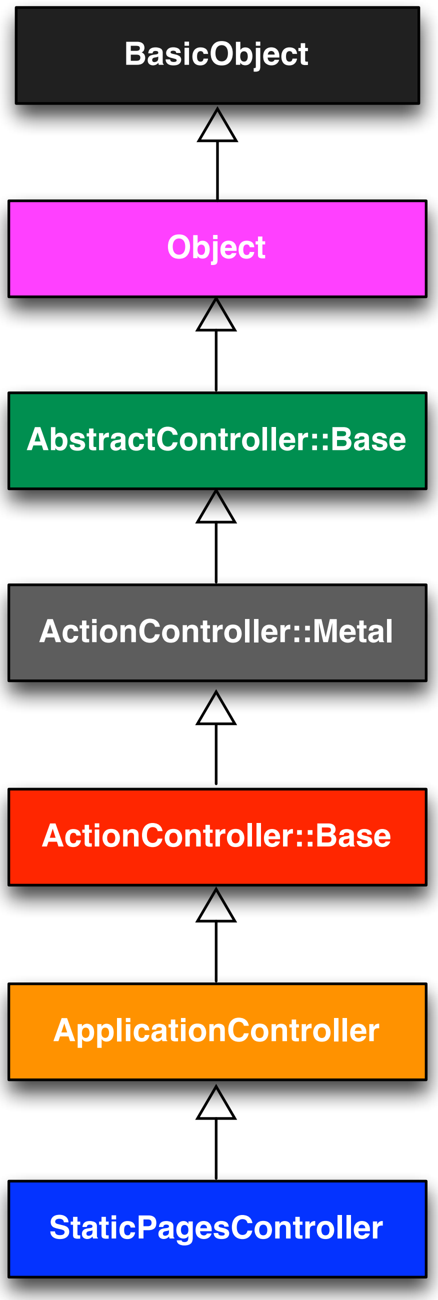

Figure 4.3: The inheritance hierarchy for the Static Pages.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. By running the Rails console in the toy app's directory from Chapter 2, confirm that you can create a **`user`** object using **`User.new`**.

2. Determine the class hierarchy of the **`user`** object.

4.4.5 A user class

We end our tour of Ruby with a complete class of our own, a **`User`** class that anticipates the User model coming up in Chapter 6.

So far we've entered class definitions at the console, but this quickly becomes tiresome; instead, create the file **`example_user.rb`** in your application root directory and fill it with the contents of Listing 4.17.

Listing 4.17: Code for an example user.
`example_user.rb`

```ruby
class User
  attr_accessor :name, :email

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

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

There's quite a bit going on here, so let's take it step by step. The first line,

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

creates *attribute accessors* corresponding to a user's name and email address. This creates "getter" and "setter" methods that allow us to retrieve (get) and assign (set) **`@name`** and **`@email`** *instance variables*, which were mentioned briefly in Section 2.2.2 and Section 3.4.2. In Rails, the principal importance of instance variables is that they are automatically available in the views, but in general they are used for variables that need to be available throughout a Ruby class. (We'll have more to say about this in a moment.) Instance variables always begin with an `@` sign, and are `nil` when undefined.

The first method, `initialize`, is special in Ruby: it's the method called when we execute `User.new`. This particular `initialize` takes one argument, `attributes`:

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

Here the `attributes` variable has a *default value* equal to the empty hash, so that we can define a user with no name or email address. (Recall from Section 4.3.3 that hashes return `nil` for nonexistent keys, so `attributes[:name]` will be `nil` if there is no `:name` key, and similarly for `attributes[:email]`.)

Finally, our class defines a method called `formatted_email` that uses the values of the assigned `@name` and `@email` variables to build up a nicely formatted version of the user's email address using string interpolation (Section 4.2.1):

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

Because `@name` and `@email` are both instance variables (as indicated with the `@` sign), they are automatically available in the `formatted_email` method.

Let's fire up the console, `require` the example user code, and take our User class out for a spin:

```
>> require './example_user'     # This is how you load the example_user code.
=> true
>> example = User.new
```

```
=> #<User:0x224ceec @email=nil, @name=nil>
>> example.name                    # nil since attributes[:name] is nil
=> nil
>> example.name = "Example User"           # Assign a non-nil name
=> "Example User"
>> example.email = "user@example.com"      # and a non-nil email address
=> "user@example.com"
>> example.formatted_email
=> "Example User <user@example.com>"
```

Here the `'.'` is Unix for "current directory", and `'./example_user'` tells Ruby to look for an example user file relative to that location. The subsequent code creates an empty example user and then fills in the name and email address by assigning directly to the corresponding attributes (assignments made possible by the `attr_accessor` line in Listing 4.17). When we write

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

Ruby is setting the `@name` variable to `"Example User"` (and similarly for the `email` attribute), which we then use in the `formatted_email` method.

Recalling from Section 4.3.4 we can omit the curly braces for final hash arguments, we can create another user by passing a hash to the `initialize` method to create a user with pre-defined attributes:

```
>> 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>"
```

We will see starting in Chapter 7 that initializing objects using a hash argument, a technique known as *mass assignment*, is common in Rails applications.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. In the example User class, change from **`name`** to separate first and last name attributes, and then add a method called **`full_name`** that returns the first and last names separated by a space. Use it to replace the use of **`name`** in the formatted email method.

2. Add a method called **`alphabetical_name`** that returns the last name and first name separated by comma-space.

3. Verify that **`full_name.split`** is the same as **`alphabetical_name.split(', ').reverse`**.

4.5 Conclusion

This concludes our overview of the Ruby language. In Chapter 5, we'll start putting it to good use in developing the sample application.

We won't be using the **`example_user.rb`** file from Section 4.4.5, so I suggest removing it:

```
$ rm example_user.rb
```

Then commit the other changes to the main source code repository and merge into the **`master`** branch, push up to GitHub, and deploy to Heroku:

```
$ git commit -am "Add a full_title helper"
$ git checkout master
$ git merge rails-flavored-ruby
```

As a reality check, it's a good practice to run the test suite before pushing or deploying:

```
$ rails test
```

Then push up to GitHub: