# RUBY
for web designers

patrick j. sparrow

Ruby for Web Designers

Object-Oriented Programming for HTML & CSS Developers

Patrick J. Sparrow

This book is for sale at http://leanpub.com/rubyforwebdesigners

This version was published on 2016-07-04

Tweet This Book!

Please help Patrick J. Sparrow by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm reading Ruby for Web Designers!

The suggested hashtag for this book is #rubyforweb.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#rubyforweb

Contents

Introduction

Ruby for Web Designers is written specifically for designers and developers who know HTML and want to learn how to write code. Each chapter introduces a programming concept by building part of a web page in Ruby. If you have experience with HTML and CSS, the examples in this book will make sense.

About the Book

This book is not a tutorial for building web applications. There are already tons of those. If you want to build a blog as quickly as possible without gaining any general programming knowledge, stop reading now. This book is not for you. If you want to learn more about object-oriented programming, read on.

Rather than show you how to build an application, we're going to take apart a web page and look at how its structure can be used to understand various object-oriented programming and software design concepts. Better yet, we're going to do it using one of the most exciting languages around: Ruby.

Goals

The main goal of this book is to provide a basic understanding of object-oriented programming concepts that can be used as a springboard for continuing your journey into programming. If you gain an appreciation of how programming can be used to solve problems along the way, I have done my job.

Prerequisites

The only real prerequisites for this book are the desire to learn how to code and a familiarity with HTML and CSS.

Some sections of the book reference topics related to front-end web development including the markdown text format and CSS pre-processors. Such references are included to provide additional examples that some web designers will find familiar. Experience with these technologies is not necessary.

Required Technology

Reading this book should serve as a useful tool in itself. However, following along by writing Ruby code on your own computer is strongly recommended. All you need is a computer capable of running Ruby and a text editor.

Let’s get started. Your great aunt will be calling you to fix her printer in no time!

1. Objects

An HTML Document

An HTML document is a collection of elements nested within each other. The elements in the head contain information about the document itself, and the elements in the body contain the text, images, and other pieces of information to be displayed to the viewer.

Here’s a simple HTML document.

```
<html>
 <head>
    <title>I Love HTML!</title>
  </head>
  <body>
    <h1>I Love HTML!</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

When you open it in a web browser, you don’t see any of the HTML code. None of the tags or attributes are printed on the page. Instead, the data from the HTML document is transformed by the browser and presented to the viewer.

HTML is the format required to display content in a web browser. You don’t **need** to write HTML to make a web page. If you have a way to convert your data into HTML before handing it to the browser, you can store it in a spreadsheet, a text file, or your own data format.

The Markdown Format

The markdown text format is a great example of this. Markdown is much easier to read than straight HTML.

> NOTE: This book is written entirely in markdown!

```
# Hello World!
## This is my web page

I hope you like it!
```

Upon converting it to HTML, you get the following markup.

```
<h1>Hello World!</h1>
<h2>This is my web page</h2>
<p>I hope you like it!</p>
```

If you'd like to learn more about markdown, You can use a tool like http://dillinger.io[1] to create a markdown document and convert it into HTML.

In the upcoming chapters, we'll explore another way to represent a web page. Since this is a book about object-oriented programming in Ruby, we'll do it using objects!

Objects

An object is an entity that has its own data and behavior. In Ruby, just about everything is an object.

[1]http://dillinger.io

Numbers

Numbers are objects that can be used in mathematical calculations.

> NOTE: The Ruby interpreter ignores everything after the `#`. These are called comments. They are used to explain code to the reader. In the code samples throughout this book, comments are used to show the result of certain operations.

```
1 + 1             # => 2
42 * 2            # => 84
3.14159.round(2) # => 3.14
```

Strings

Strings are objects that represent a collection of characters. We can reverse them, change their case, and append more characters to them.

```
'hello'.reverse  # => "olleh"
'hello'.upcase   # => "HELLO"
'hello' << '!'   # => "hello!"
```

Whenever we create a string, a new object is created even if the string has the same value as a string that already exists.

```
'hello'.object_id # => 3
'hello'.object_id # => 84
```

Symbols

Since strings are so widely-used in programming, Ruby provides a special, immutable (unchangeable) object that can be used in place of a string. This is object is called a `Symbol`. Symbols must start with a `:`. Whenever we use `:hello`, it always references the same object.

```
:hello.object_id # => 584
:hello.object_id # => 584
```

Using symbols in place of frequently-used strings prevents the creation of extraneous objects. Symbols are used in place of strings throughout this book.

Booleans

Boolean objects represent `true` and `false` values. When we ask an object a yes-or-no question, it answers with a boolean object.

```
3.even? # => false
'hello'.start_with? 'h' # => true
```

Nil

The `nil` object is a special object that represents nothing. When we try to get the location of a letter within a string that does not contain that letter, the result is `nil`.

```
'hello'.index('j') # => nil
```

These are just a few types of objects that Ruby provides. As you'll see in coming chapters, we can even create our own types of objects.

Variables

Variables are memory locations that hold objects. We can name them and use them to access our objects. The standard way of naming variables is snake case: lowercase letters separated by underscores.

```
my_string = 'hello'
my_string.length # => 5
my_string.upcase # => "HELLO"
```

Messages

Objects interact with each other by sending messages. When an object receives a message, it responds by behaving in a certain way. All of the following statements do the same thing.

```
'hello'.reverse          # => "olleh"
'hello'.send('reverse') # => "olleh"
'hello'.send(:reverse)  # => "olleh"
```

Because it's much easier to type `object.message` than `object.send('message')`, `send` is only used in special cases.

If we send an object a message that it doesn't understand, Ruby will raise a `NoMethodError`.

```
'hello'.blah # => NoMethodError: undefined method 'blah'
```

2. Classes

Classes are templates for creating objects. Just as CSS classes allow properties to be applied to multiple HTML elements, Ruby classes allow behavior to be shared between multiple objects.

Classes contain the code that tells objects how to respond to messages. The `String` class knows how to reverse a string. The `Fixnum` and `BigNum` classes knows how to add whole numbers together.

Every object is an instance of a class. We can send the `class` message to an object to ask about its class.

```
1.class       # => FixNum
3.14.class    # => Float
'hello'.class # => String
:hello.class  # => Symbol
true.class    # => TrueClass
false.class   # => FalseClass
nil.class     # => NilClass
99999999999999999999.class # => BigNum
```

A Class for HTML Divs

Let's create a Ruby class for one of the most commonly-used HTML elements, the `<div>`.

Here's a sample div:

```
<div>Hello World!</div>
```

The two important things to note here are that divs have an opening and closing tag and that they can have content.

Let's start building the class. In Ruby, the standard way to write class names is CamelCased, with the first letter of each word capitalized and no spaces.

```
class Div
end
```

Although Ruby provides shortcuts for creating instances of the `String`, `FixNum`, `Symbol`, and other commonly-used classes, the standard way to create an instance of a class is to tell the class to create new object. We do this by sending the `new` message to the class.

```
Div.new
```

To make our `Div` objects useful, we need the ability to put content into them and convert them to HTML. To do this, we need to add some methods. Methods are blocks of code that tell objects how to respond to specific messages.

In Ruby, methods are defined using `def` followed by the method name in snake-case format, just like variables. Methods are terminated using the `end` keyword. The last line of a method determines what is returned from the method.

```
def method_that_returns_the_number_one
  1
end
```

The `initialize` Method

We want the ability to give our divs content when we create them.

```
Div.new('Hello World!')
```

When a class receives the `new` message, it creates a new instance of that class and sends the `initialize` message to the new object. To control what happens when the object receives this message, we need to define the `initialize` method and have it accept a parameter for the content. Parameters are variables that hold the data that is passed to a method.

```
class Div
  def initialize(content)
  end
end
```

The `Div` class now expects content to be passed to it when creating a new instance.

```
Div.new('Hello World!')
```

What happens if we try to create a `Div` without content?

```
Div.new
# => ArgumentError: wrong number of arguments (0 for 1)
```

An argument is a piece of data passed to a method. Arguments and parameters are thrown around interchangeably because they are so closely related, but they are not exactly the same. Here's an example to show the difference between the two.

Parameters vs Arguments

When you define a method, the objects that it accepts are the parameters.

```
def a_method(first_parameter, second_parameter)
end
```

When you call that method, the objects that you pass into it are the arguments.

```
a_method('first argument', :second_argument)
```

The arguments sent to the method are placed into the parameter variables.

The `initialize` method is expecting an argument for the `content` parameter. Since we didn't provide one, Ruby raised an exception. Since empty divs are common in HTML documents, we need to fix this problem. We can tell `initialize` that the default the value of `content` should be an empty string. Then, when `Div.new` is called without any arguments, the `content` parameter will contain an empty string.

```
def initialize(content = '')
end
```

We can now create empty and non-empty divs without raising exceptions.

```
Div.new
Div.new('Hello World!')
```

Instance Variables

Unfortunately, we aren't storing the content anywhere. So, after `initialize` is executed, the content is gone forever. To store the content, we need to put it into an instance variable. Instance variables hold data that belongs to an object. They start with a `@`.

Let's modify the `initialize` method so it stores the content.

```
def initialize(content = '')
  @content = content
end
```

Now, when we create a new `Div` and pass in some content, the content is stored for later use.

The `to_s` Method

The `to_s` method (short for "to string") determines how an object responds when converted into a string.

Let's add a `to_s` method to our `Div` class and have it return a string containing the HTML markup for a div.

```
def to_s
  '<div></div>'
end
```

`print` and `puts` are two methods that Ruby provides for outputting strings. The difference between them is that `puts` includes a new line at the end of the string while `print` does not.

Now, we can run `puts Div.new` and it will print out `<div></div>`.

String Interpolation

The last step in making our `Div` class useful is to include the value of the `@content` variable between the opening and closing tags. We can do this via string interpolation. Interpolation is a fancy term for executing code in a string. In Ruby, this is done by wrapping the code in `#{}` inside the string.

> NOTE: This only works for strings wrapped in double quotes (“”). Strings wrapped in single quotes (‘’) will not be interpolated.

Let’s update our `to_s` method to include the content.

```
def to_s
 "<div>#{@content}</div>"
end
```

Now, when we print a `Div`, the content we passed into `Div.new` is included in the output.

```
puts Div.new('Hello World!') # => "<div>Hello World!</div>"
puts Div.new # => “<div></div>”
```

The Code

Here’s the complete `Div` class along with an example of how to use it.

```ruby
class Div
  def initialize(content = '')
    @content = content
  end

  def to_s
    "<div>#{@content}</div>"
  end
end

puts Div.new('Hello World!') # => "<div>Hello World!</div>"
```