CMR
INSTITUTE OF
TECHNOLOGY

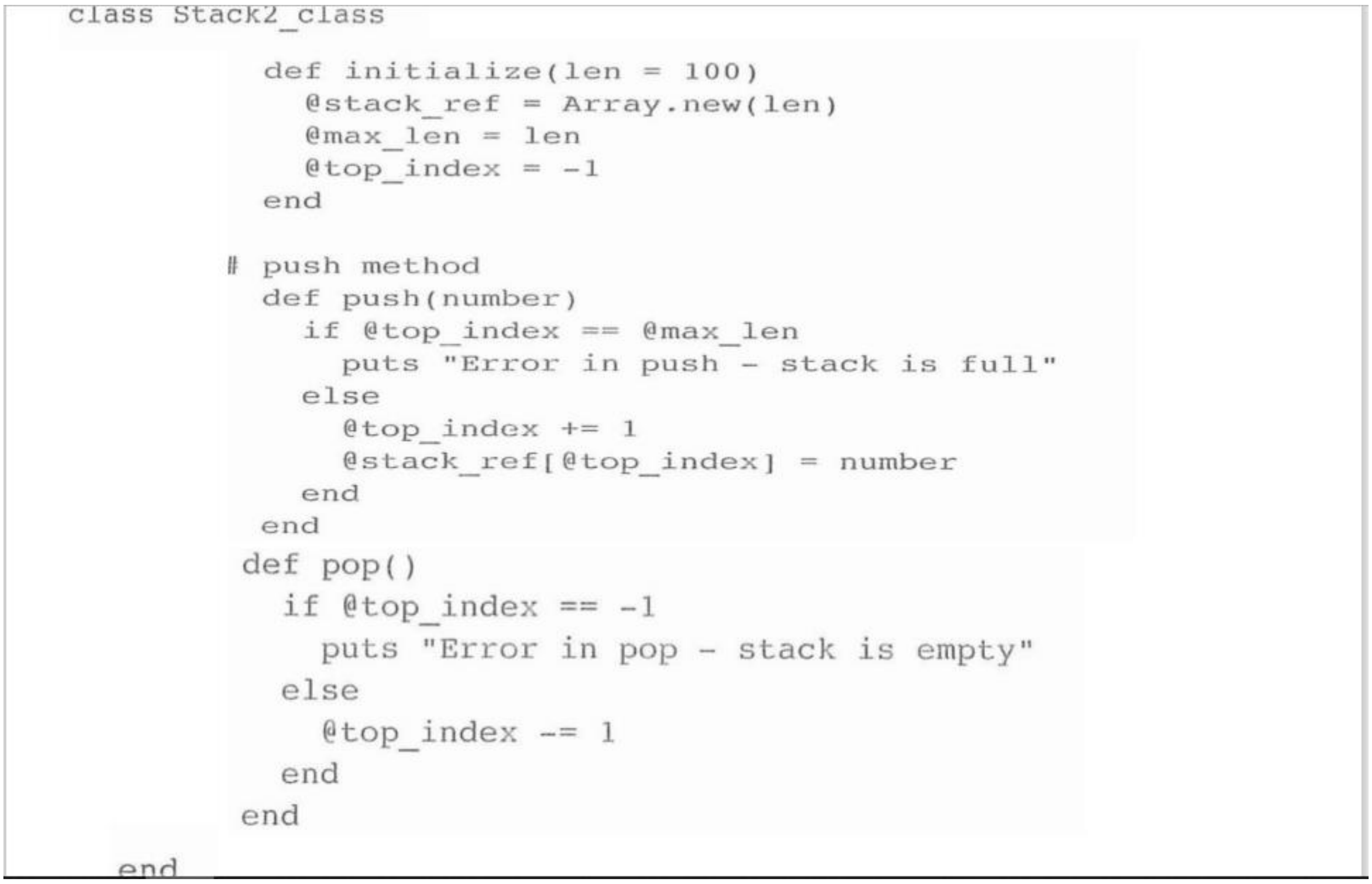
USN

Internal Assessment Test 3 Answer Key– June. 2022									
Sub:	**ADVANCES IN WEB TECHNOLOGIES**					Sub Code:	20MCA41	Branch:	MCA
Date:	18/06/2022	Duration:	90 min's	Max Marks:	50	Sem	IV		

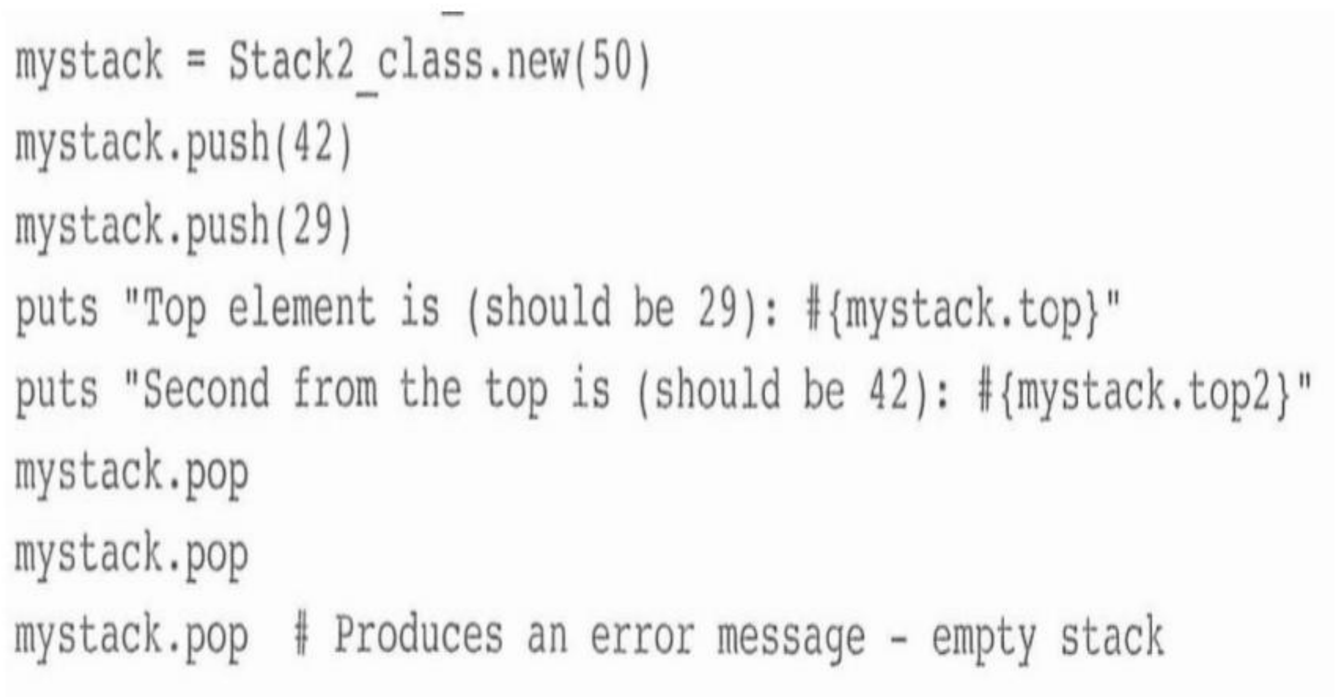
<u>Note : Answer FIVE FULL Questions, choosing ONE full question from each Module</u>

	PART I	MARKS	OBE	
			CO	RBT
1	Write a program to implement a stack like structure in an Array using classes in Ruby. **OR**	[10]	CO3	L4
2	Explain String Methods in Ruby.	[10]	CO3	L2
3	**PART II** a) Explain Creation and Accessing of Arrays in Ruby. b) Write any five built-in methods for Arrays in Ruby with examples. **OR**	[10]	CO3	L2
4	Explain directory structure of rails application using "Hello world" in Rails	[10]	CO4	L3
5	**PART III** Explain rails application to connect with Database **OR**	[10]	CO4	L4
6	a) Explain Overview of Rails with MVC Architecture. b)Compare Controller, Views, Helpers, and Models in Ruby on Rails	[10]	CO3	L2
7	**PART IV** Explain AJAX Patterns with suitable Examples **OR**	[10]	CO2	L2
8	Explain with example for Handling multiple XMLHttpRequest objects in the same page.	[10]	CO2	L4
9	**PART V** a) Explain any two Lists in BOOTSTRAP with examples. b) Explain different ways to display code with Bootstrap **OR**	[10]	CO5	L2
10	a)Explain Table elements supported by BOOTSTRAP with Example. b)Describe Optional Table classes in BOOTSTRAP with Example	[10]	CO5	L2

1) Write a program to implement a stack like structure in an Array using classes in Ruby.

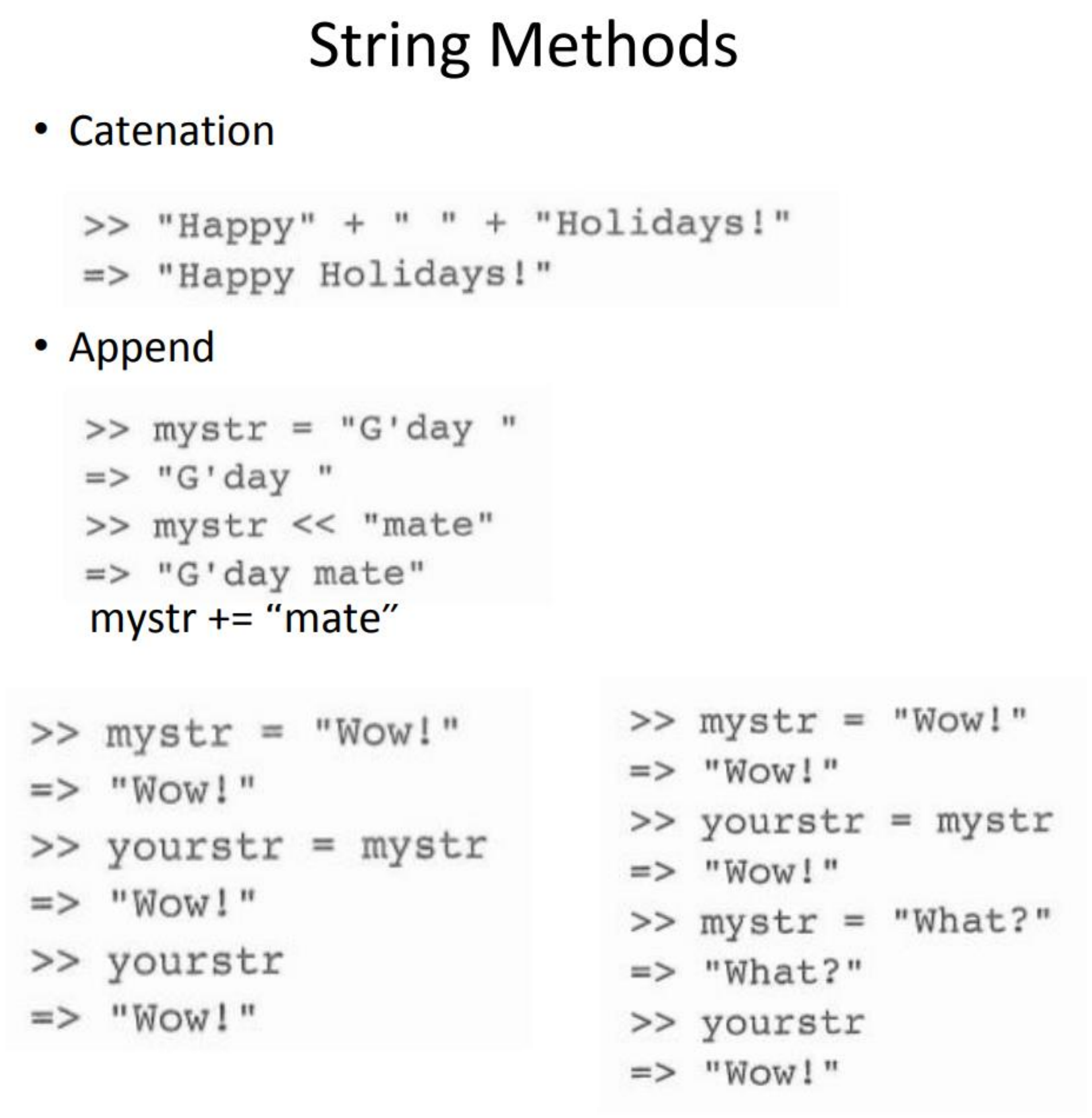


```
class Stack2_class

    def initialize(len = 100)
      @stack_ref = Array.new(len)
      @max_len = len
      @top_index = -1
    end

  # push method
    def push(number)
      if @top_index == @max_len
        puts "Error in push - stack is full"
      else
        @top_index += 1
        @stack_ref[@top_index] = number
      end
    end
   def pop()
     if @top_index == -1
       puts "Error in pop - stack is empty"
     else
       @top_index -= 1
     end
   end
 end
```



```
mystack = Stack2_class.new(50)
mystack.push(42)
mystack.push(29)
puts "Top element is (should be 29): #{mystack.top}"
puts "Second from the top is (should be 42): #{mystack.top2}"
mystack.pop
mystack.pop
mystack.pop  # Produces an error message - empty stack
```

2) Explain String Methods in Ruby.



String Methods

- Catenation

```
>> "Happy" + " " + "Holidays!"
=> "Happy Holidays!"
```

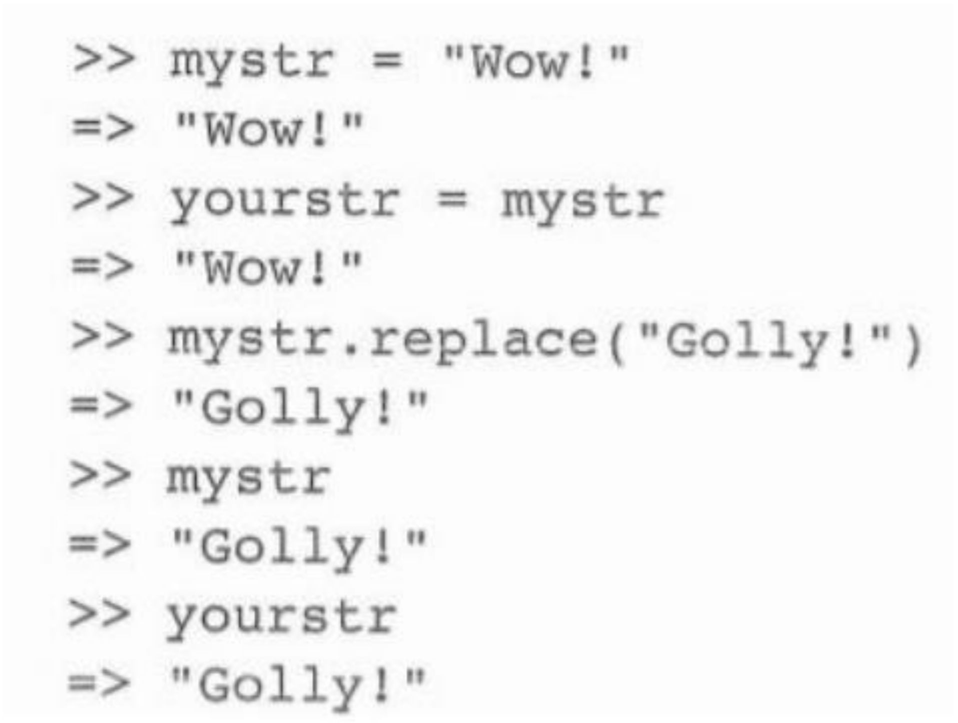
- Append

```
>> mystr = "G'day "
=> "G'day "
>> mystr << "mate"
=> "G'day mate"
```

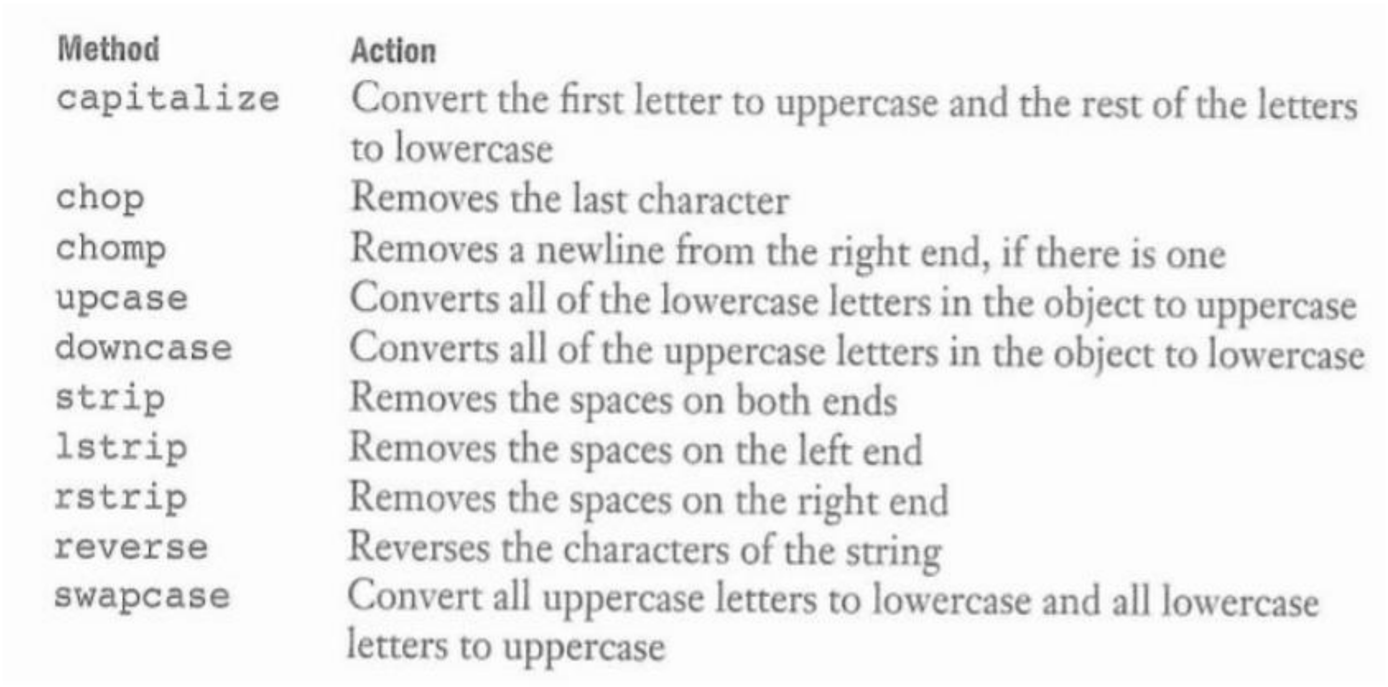
mystr += “mate”

```
>> mystr = "Wow!"
=> "Wow!"
>> yourstr = mystr
=> "Wow!"
>> yourstr
=> "Wow!"
```

```
>> mystr = "Wow!"
=> "Wow!"
>> yourstr = mystr
=> "Wow!"
>> mystr = "What?"
=> "What?"
>> yourstr
=> "Wow!"
```

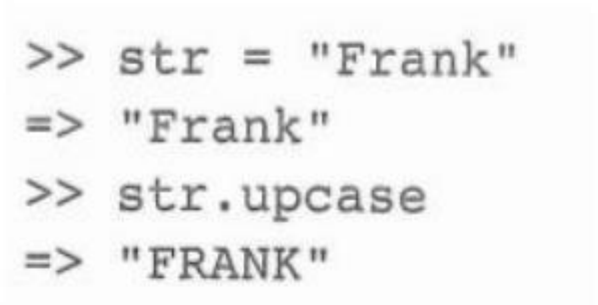


```
>> mystr = "Wow!"
=> "Wow!"
>> yourstr = mystr
=> "Wow!"
>> mystr.replace("Golly!")
=> "Golly!"
>> mystr
=> "Golly!"
>> yourstr
=> "Golly!"
```

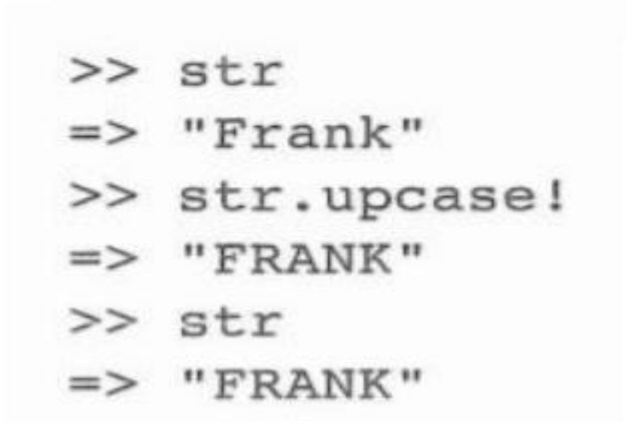


Method	Action
`capitalize`	Convert the first letter to uppercase and the rest of the letters to lowercase
`chop`	Removes the last character
`chomp`	Removes a newline from the right end, if there is one
`upcase`	Converts all of the lowercase letters in the object to uppercase
`downcase`	Converts all of the uppercase letters in the object to lowercase
`strip`	Removes the spaces on both ends
`lstrip`	Removes the spaces on the left end
`rstrip`	Removes the spaces on the right end
`reverse`	Reverses the characters of the string
`swapcase`	Convert all uppercase letters to lowercase and all lowercase letters to uppercase

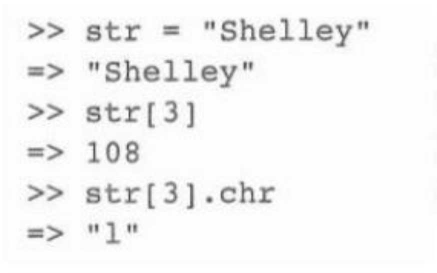
- Bang or mutator methods



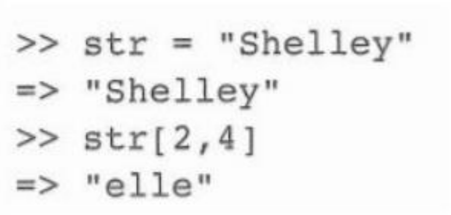
```
>> str = "Frank"
=> "Frank"
>> str.upcase
=> "FRANK"
```



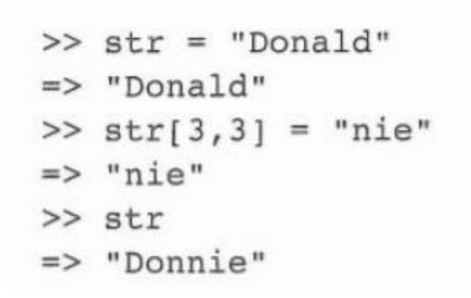
```
>> str
=> "Frank"
>> str.upcase!
=> "FRANK"
>> str
=> "FRANK"
```



```
>> str = "Shelley"
=> "Shelley"
>> str[3]
=> 108
>> str[3].chr
=> "l"
```



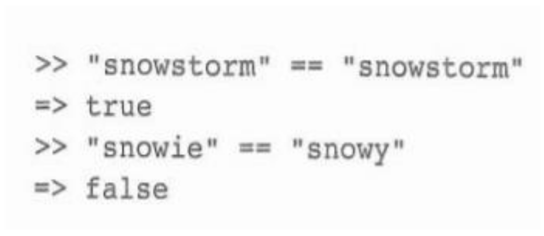
```
>> str = "Shelley"
=> "Shelley"
>> str[2,4]
=> "elle"
```



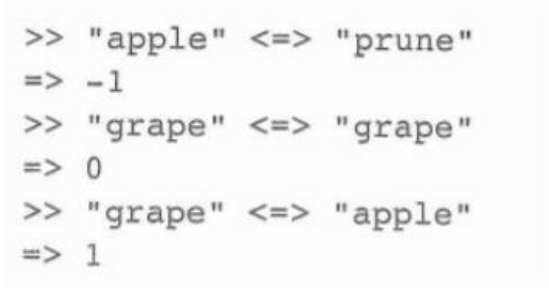
```
>> str = "Donald"
=> "Donald"
>> str[3,3] = "nie"
=> "nie"
>> str
=> "Donnie"
```

S	h	e	l	l	e	y
0	**1**	**2**	**3**	**4**	**5**	**6**

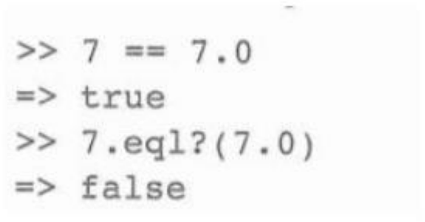
1. ==
2. equal?
3. eql?
4. < = >



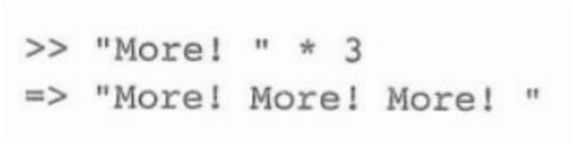
```
>> "snowstorm" == "snowstorm"
=> true
>> "snowie" == "snowy"
=> false
```



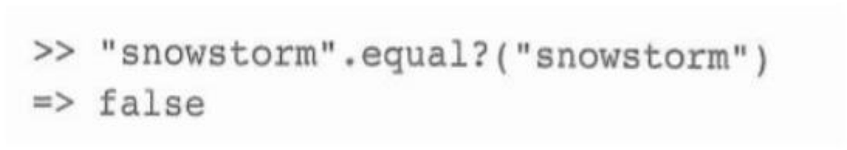
```
>> "apple" <=> "prune"
=> -1
>> "grape" <=> "grape"
=> 0
>> "grape" <=> "apple"
=> 1
```



```
>> 7 == 7.0
=> true
>> 7.eql?(7.0)
=> false
```



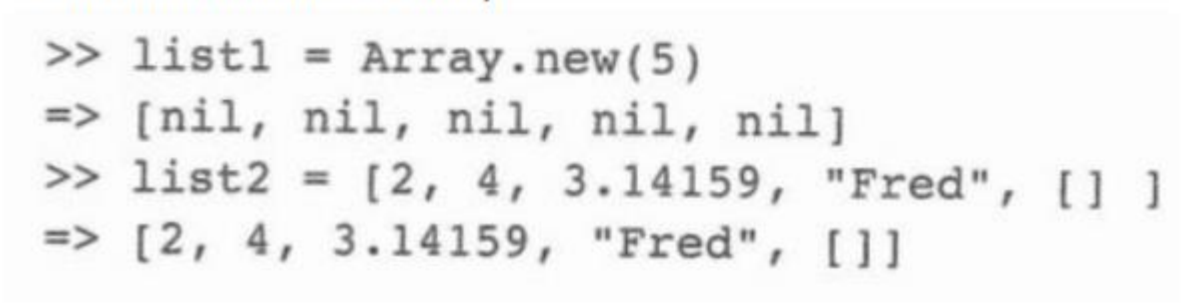
```
>> "More! " * 3
=> "More! More! More! "
```



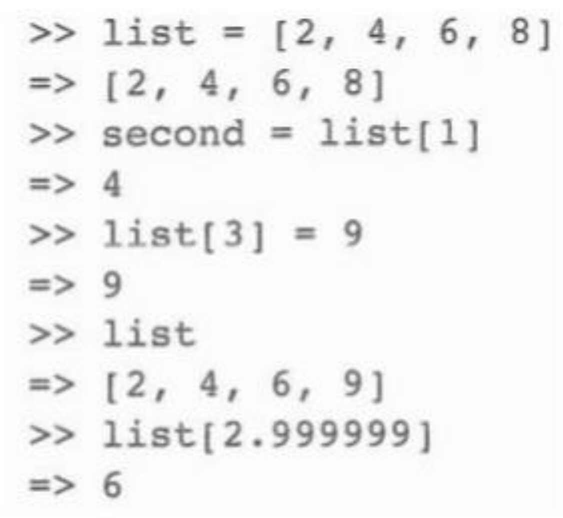
```
>> "snowstorm".equal?("snowstorm")
=> false
```

3a) Explain Creation and Accessing of Arrays in Ruby.
b) Write any five built-in methods for Arrays in Ruby with examples.

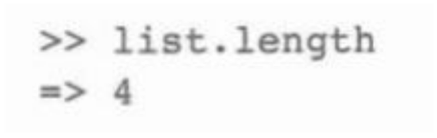
Create and access array



```
>> list1 = Array.new(5)
=> [nil, nil, nil, nil, nil]
>> list2 = [2, 4, 3.14159, "Fred", [] ]
=> [2, 4, 3.14159, "Fred", []]
```



```
>> list = [2, 4, 6, 8]
=> [2, 4, 6, 8]
>> second = list[1]
=> 4
>> list[3] = 9
=> 9
>> list
=> [2, 4, 6, 9]
>> list[2.999999]
=> 6
```



```
>> list.length
=> 4
```

Built-in methods

- Shift , unshift and pop , push
  list=[2, 4, 17, 3]

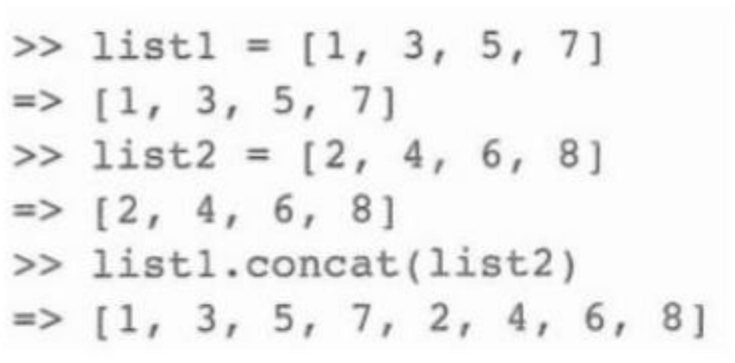
  list.shift
  o/p   [4, 17, 3]

  list.pop
  o/p   [2, 4, 17]

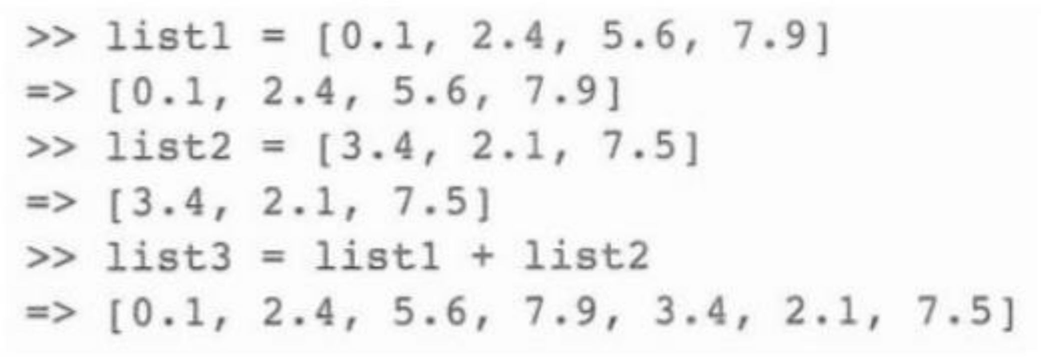
  list.unshift(8)
  o/p   [8, 4, 17, 3]

  list.push(8,5)
  o/p   [2, 4, 17, 8,5]

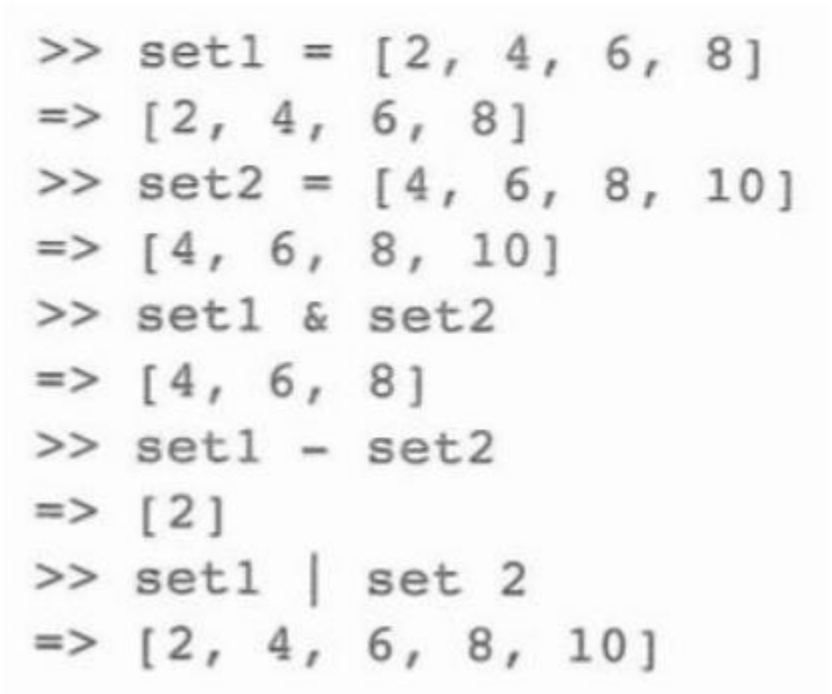
Concat



```
>> list1 = [1, 3, 5, 7]
=> [1, 3, 5, 7]
>> list2 = [2, 4, 6, 8]
=> [2, 4, 6, 8]
>> list1.concat(list2)
=> [1, 3, 5, 7, 2, 4, 6, 8]
```



```
>> list1 = [0.1, 2.4, 5.6, 7.9]
=> [0.1, 2.4, 5.6, 7.9]
>> list2 = [3.4, 2.1, 7.5]
=> [3.4, 2.1, 7.5]
>> list3 = list1 + list2
=> [0.1, 2.4, 5.6, 7.9, 3.4, 2.1, 7.5]
```



```
>> set1 = [2, 4, 6, 8]
=> [2, 4, 6, 8]
>> set2 = [4, 6, 8, 10]
=> [4, 6, 8, 10]
>> set1 & set2
=> [4, 6, 8]
>> set1 - set2
=> [2]
>> set1 | set 2
=> [2, 4, 6, 8, 10]
```

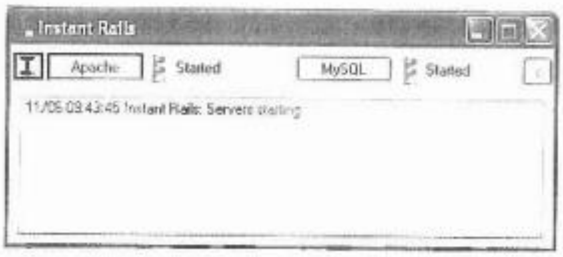
4) Explain directory structure of rails application using “Hello world” in Rails

15.2.1 Static Documents—Hello World in Rails

This section describes how to build a Hello World application in Rails. The purpose of such an exercise is to demonstrate the structure of the simplest possible Rails application, showing what files must be created and where they must reside in the directory structure. InstantRails itself runs in the following directory:

```
C:\myrails\InstantRails-1.7-win\InstantRails
```

where the first directory's name, `myrails` in this example, is chosen by the person who installed InstantRails. In this directory there is an application file named InstantRails. Running this application opens a small window, as shown in Figure 15.1.



```
11/06 09:43:45 Instant Rails: Servers starting
```

Figure 15.1 The InstantRails application window

A click on the black "`I`" on the left border of this window produces a small menu. Click the `Rails Application` entry in this menu, which opens another menu. Clicking on the `Open Ruby Console Window` entry in this menu opens a DOS command window in the following directory:

```
C:\myrails\InstantRails-1.7-win\InstantRails\rails_aps
```

It is at this command line that Rails commands can be given. They cannot be given in a normal DOS command window.

To this base directory, users usually add a new subdirectory for all their Rails applications. We named ours `exercises`. In the new `exercises` directory, the new application `rails1` is created with the following command:

```
>rails rails1
```

```
>ruby script/generate controller say
```

With this command we have chosen the name `say` for the controller for our application.The response produced by the execution of this command follows:

```
exists  app/controllers/
exists  app/helpers/
create  app/views/say
exists  test/functional/
create  app/controllers/say_controller.rb
create  test/functional/say_controller_test.rb
create  app/helpers/say_helper.rb
```

The `exists` lines above indicate files and directories that are verified to already exist. The `create` lines show the newly created directories and files. There are now two Ruby classes in the controllers directory, `application.rb` and `say_controller.rb`, where the `say_controller.rb` class is a subclass of `application.rb`, which provides the default behavior defined in the parent class. The `say_controller.rb` is the specific controller for the `rails1` application. Following is a listing of `say_controller.rb`:

```
class SayController < ApplicationController
end
```

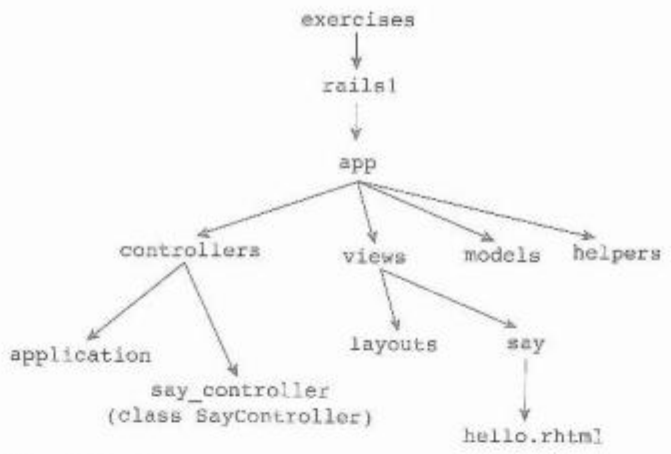
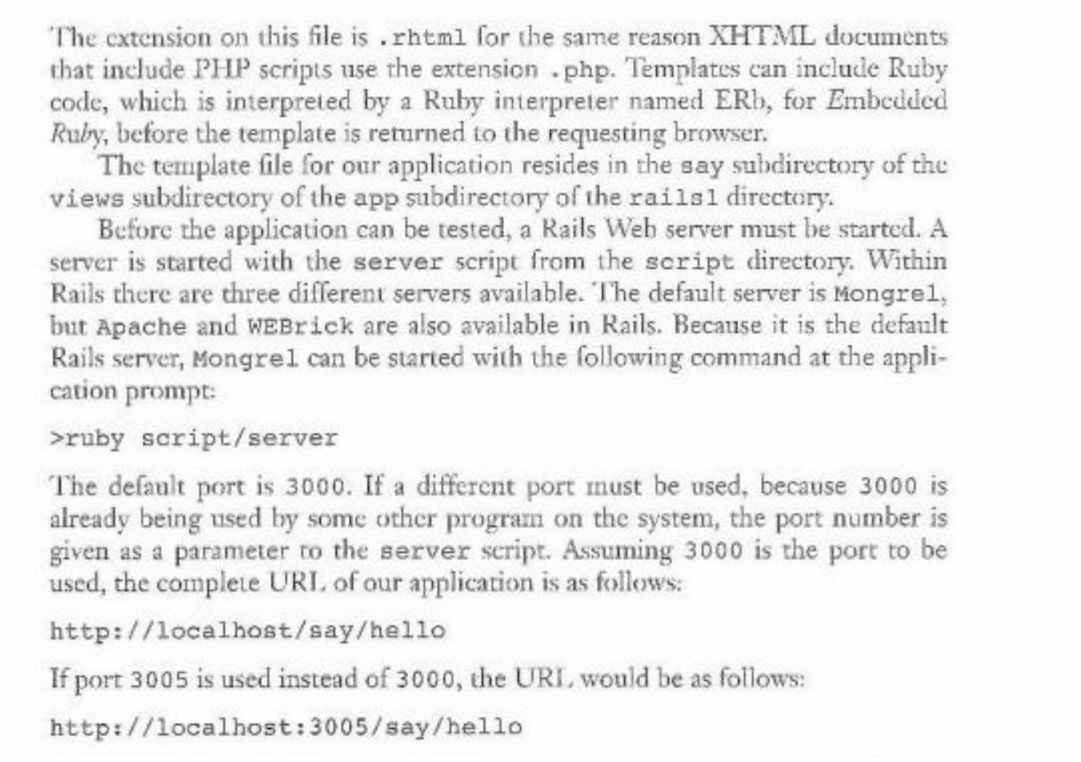


Figure 15.2 Directory structure for the `rails1` application

`SayController` is an empty class, other than what it inherits from `application.rb`. The controller produces, at least indirectly, the response to requests, so a method must be added to it. The method does not need to actually do anything, other than indicate a document that will be the response. The mere existence of the method specifies by its name the response document. So, the action will be nothing more than an empty method definition, whose name will be the same as that of the response document in the `say` subdirectory of `views`. With the empty method, which is called an *action method*, the controller now has the following appearance:

```
class SayController < ApplicationController
  def hello
  end
end
```

```
<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<!-- hello.rhtml - the template for Hello World
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Hello, Rails! </title>
  </head>
  <body>
    <h1> Hello from Rails! </h1>
  </body>
</html>
```


The extension on this file is `.rhtml` for the same reason XHTML documents that include PHP scripts use the extension `.php`. Templates can include Ruby code, which is interpreted by a Ruby interpreter named ERb, for *Embedded Ruby*, before the template is returned to the requesting browser.

The template file for our application resides in the `say` subdirectory of the `views` subdirectory of the `app` subdirectory of the `rails1` directory.

Before the application can be tested, a Rails Web server must be started. A server is started with the `server` script from the `script` directory. Within Rails there are three different servers available. The default server is `Mongrel`, but `Apache` and `WEBrick` are also available in Rails. Because it is the default Rails server, `Mongrel` can be started with the following command at the application prompt:

```
>ruby script/server
```

The default port is `3000`. If a different port must be used, because `3000` is already being used by some other program on the system, the port number is given as a parameter to the `server` script. Assuming `3000` is the port to be used, the complete URL of our application is as follows:

```
http://localhost/say/hello
```

If port `3005` is used instead of `3000`, the URL would be as follows:

```
http://localhost:3005/say/hello
```

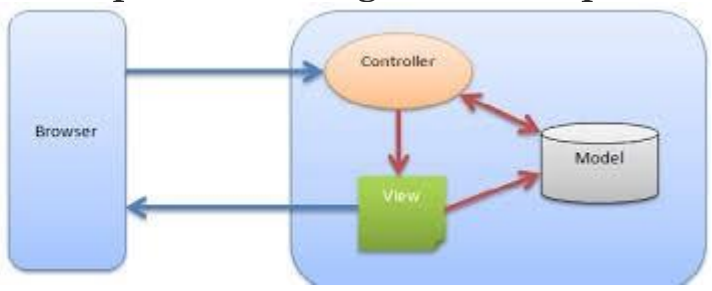
5) Explain rails application to connect with Database.

Step 1: Download "railsinstaller " Which is "railsinstaller-2.2.4 " and double click to install. Before install ensure that internet connection must be available
Step 2: Shortcut "Git bash" is visible on desktop or All programs
Step 3 : Click on "Git bash" The CMD prompt points to $ with "/c/sites " directory
Step 4: create new bookstore using the following syntax
$ rails new bookstore
Step 5: Change the directory the following syntax
$ **cd bookstore**
Step 6: Run the server using following command
$ rails server
Then open Firefox Web browser and type http://localhost:3000Which gives ruby on rails welcome page.
Step 7: Scaffold command helps to represent different data types in book table
rails generate scaffold book acc_num:integer title:string author:string edition:string publisher:string
Step 8: Use rake command to flush "book" table and its contents to **sqlite** database
$ rake db:migrate
Step 9: Use rake command to flush function paths to routes.rb
$ rake routes
Step 10: Run the server again
$ rails server
Then open Firefox Web browser and type http://localhost:3000/books
Add some books to the database by selecting *newbook*.

6) a) Explain Overview of Rails with MVC Architecture.

Ruby on rails is an open source web framework written in the Ruby programming language, and all the applications in Rails are written in Ruby. It is very popular framework for web development due to its amazing features.

This framework has built-in solutions to many common problems that developer face during web development.



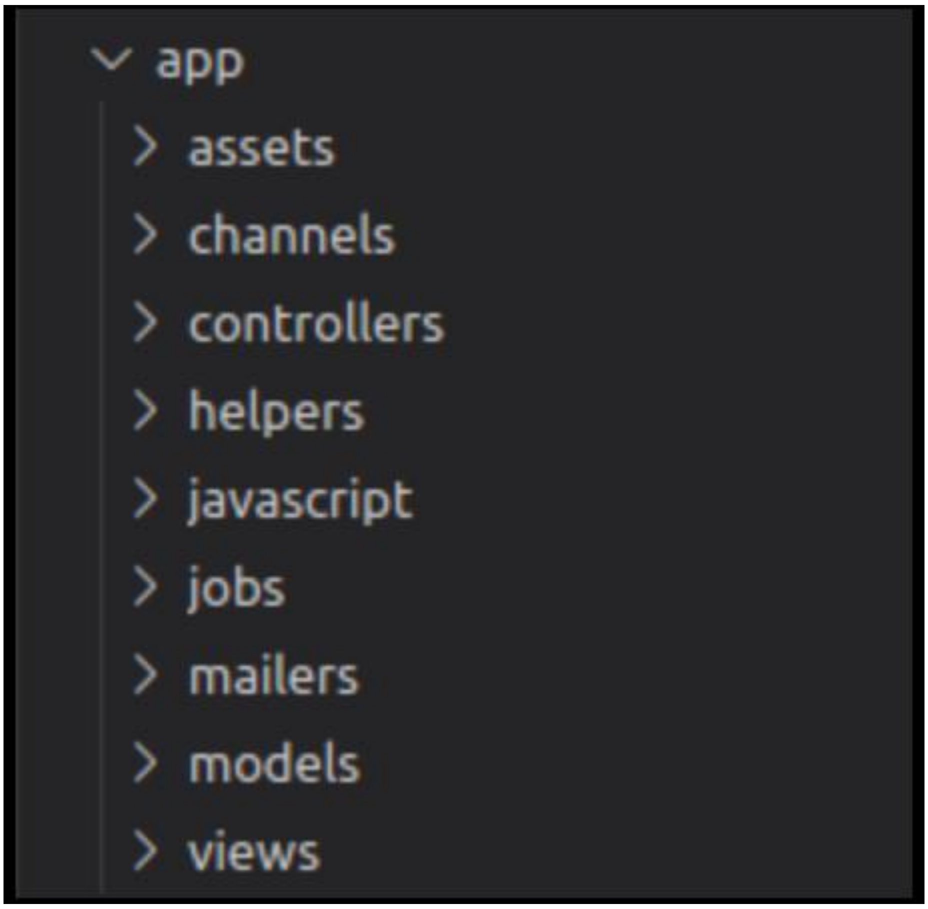
mvc architecture

Ruby on Rails uses the Model-View-Controller (MVC) architectural pattern. MVC is a pattern for the architecture of a software application. It separates an application into the following components:

- Models for handling data and business logic
- Controllers for handling the user interface and application
- Views for handling graphical user interface objects and presentation

This separation results in user requests being processed as follows:

1. The browser sends a request for a page to the controller on the server
2. The controller retrieves the data it needs from the model in order to respond to the request
3. The controller gives the retrieved data to the view
4. The view is rendered and sent back to the client for the browser to display



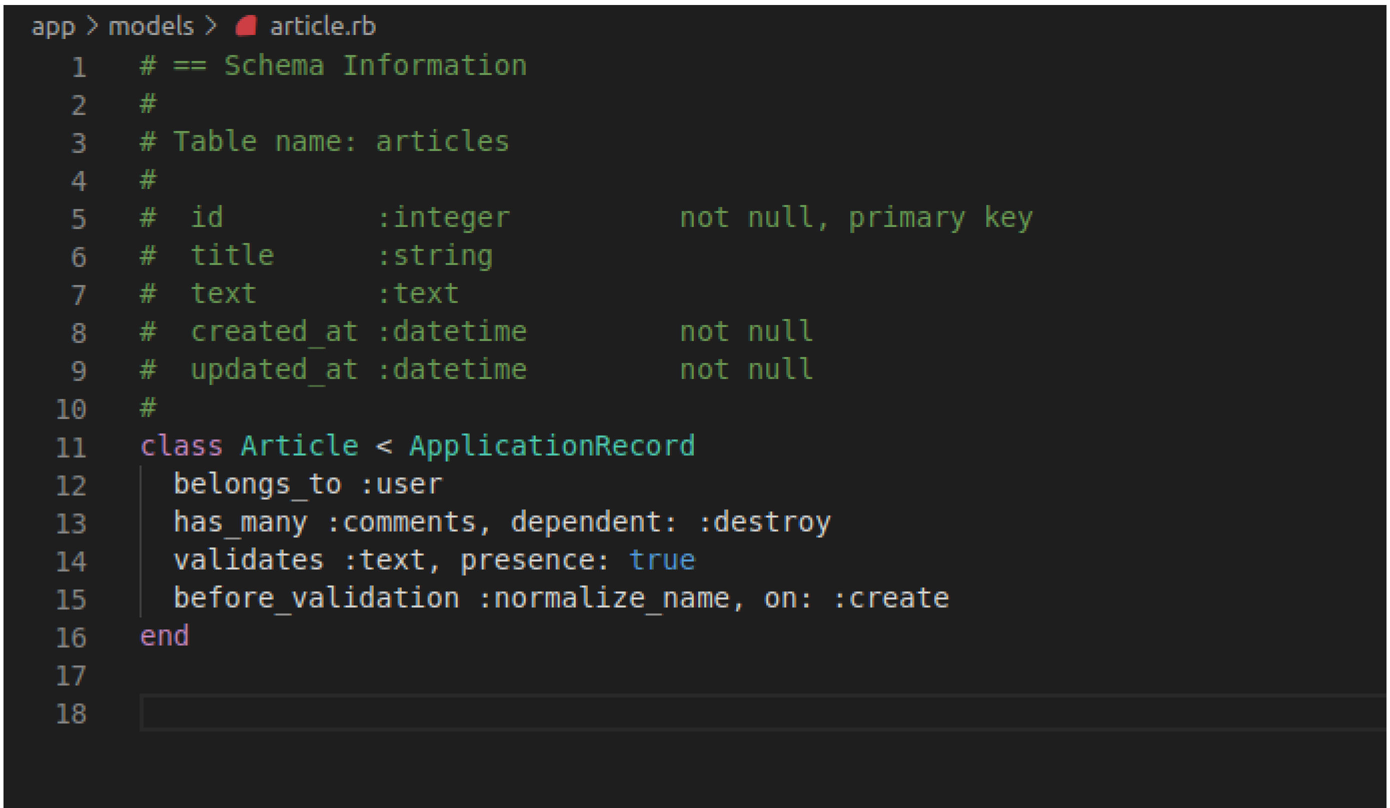
```
∨ app
  > assets
  > channels
  > controllers
  > helpers
  > javascript
  > jobs
  > mailers
  > models
  > views
```

As you can see, each component of the model-view-controller architecture has its place within the app directory—the `models`, `views`, and `controllers` sub directories respectively.

MODELS

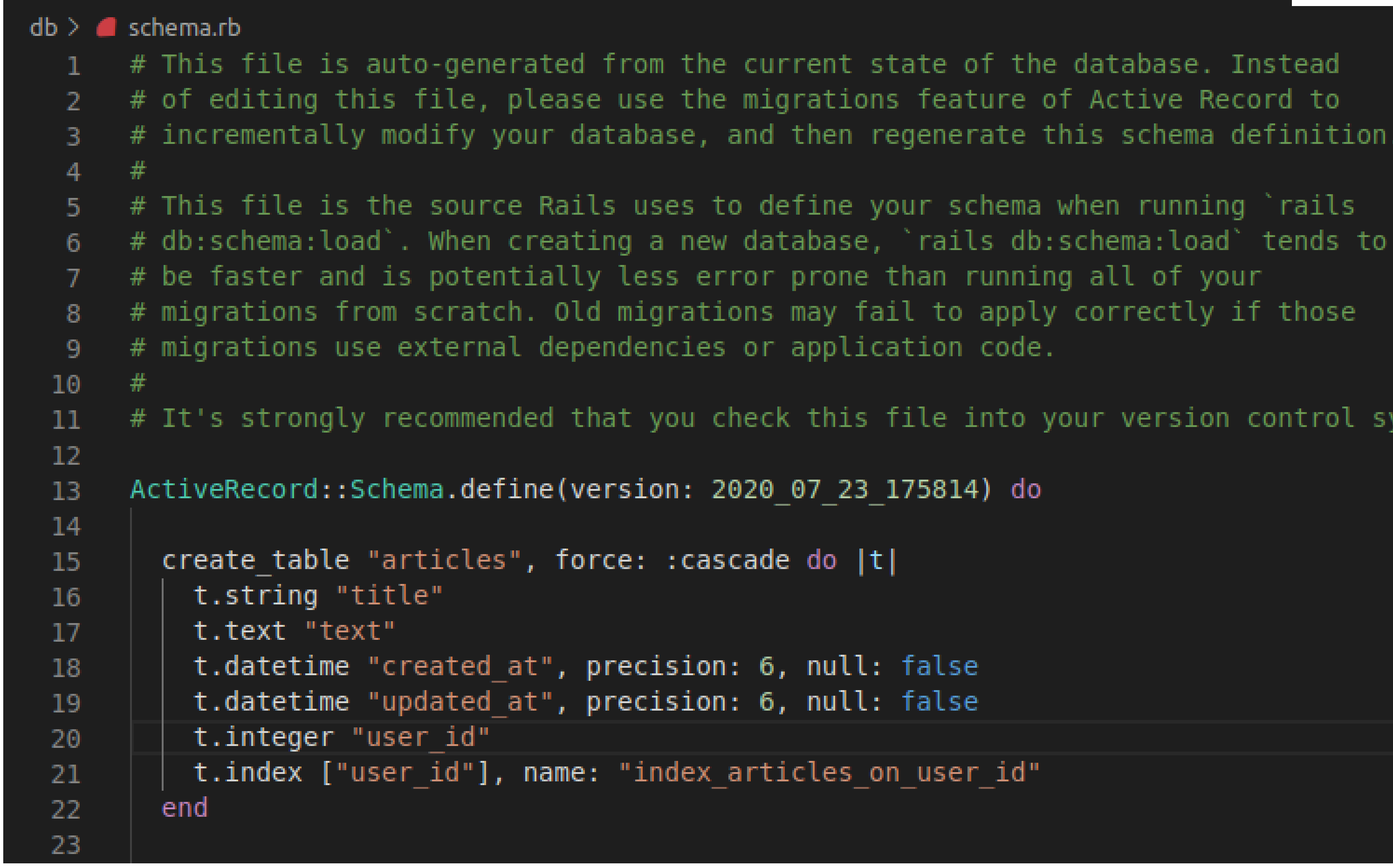
ActiveRecord is the module for handling business logic and database communication. It plays the role of model in our MVC architecture.

Lets consider this example where a user can create articles and each article can have many comments. The image on the left shows the all the validations and associations of article model with other models, where as the image on the write shows the schema of articles.



app > models > article.rb

```ruby
# == Schema Information
#
# Table name: articles
#
#  id         :integer          not null, primary key
#  title      :string
#  text       :text
#  created_at :datetime         not null
#  updated_at :datetime         not null
#
class Article < ApplicationRecord
  belongs_to :user
  has_many :comments, dependent: :destroy
  validates :text, presence: true
  before_validation :normalize_name, on: :create
end

```



db > schema.rb

```ruby
# This file is auto-generated from the current state of the database. Instead
# of editing this file, please use the migrations feature of Active Record to
# incrementally modify your database, and then regenerate this schema definition
#
# This file is the source Rails uses to define your schema when running `rails
# db:schema:load`. When creating a new database, `rails db:schema:load` tends to
# be faster and is potentially less error prone than running all of your
# migrations from scratch. Old migrations may fail to apply correctly if those
# migrations use external dependencies or application code.
#
# It's strongly recommended that you check this file into your version control sy

ActiveRecord::Schema.define(version: 2020_07_23_175814) do

  create_table "articles", force: :cascade do |t|
    t.string "title"
    t.text "text"
    t.datetime "created_at", precision: 6, null: false
    t.datetime "updated_at", precision: 6, null: false
    t.integer "user_id"
    t.index ["user_id"], name: "index_articles_on_user_id"
  end

```

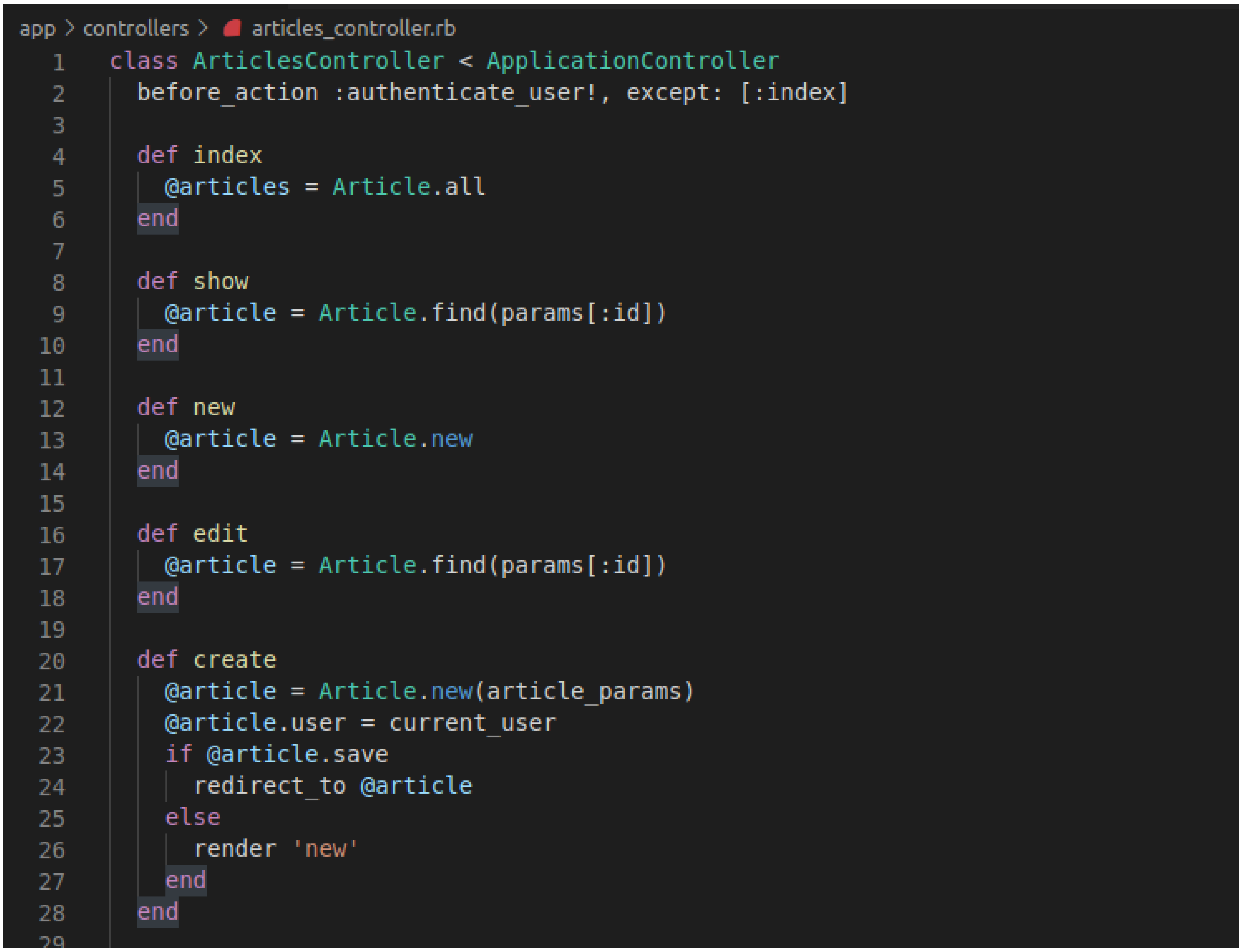
ActiveRecord is designed to handle all of an application's tasks that relate to the database, including:

- establishing a connection to the database server
- retrieving data from a table
- storing new data in the database.

CONTROLLERS

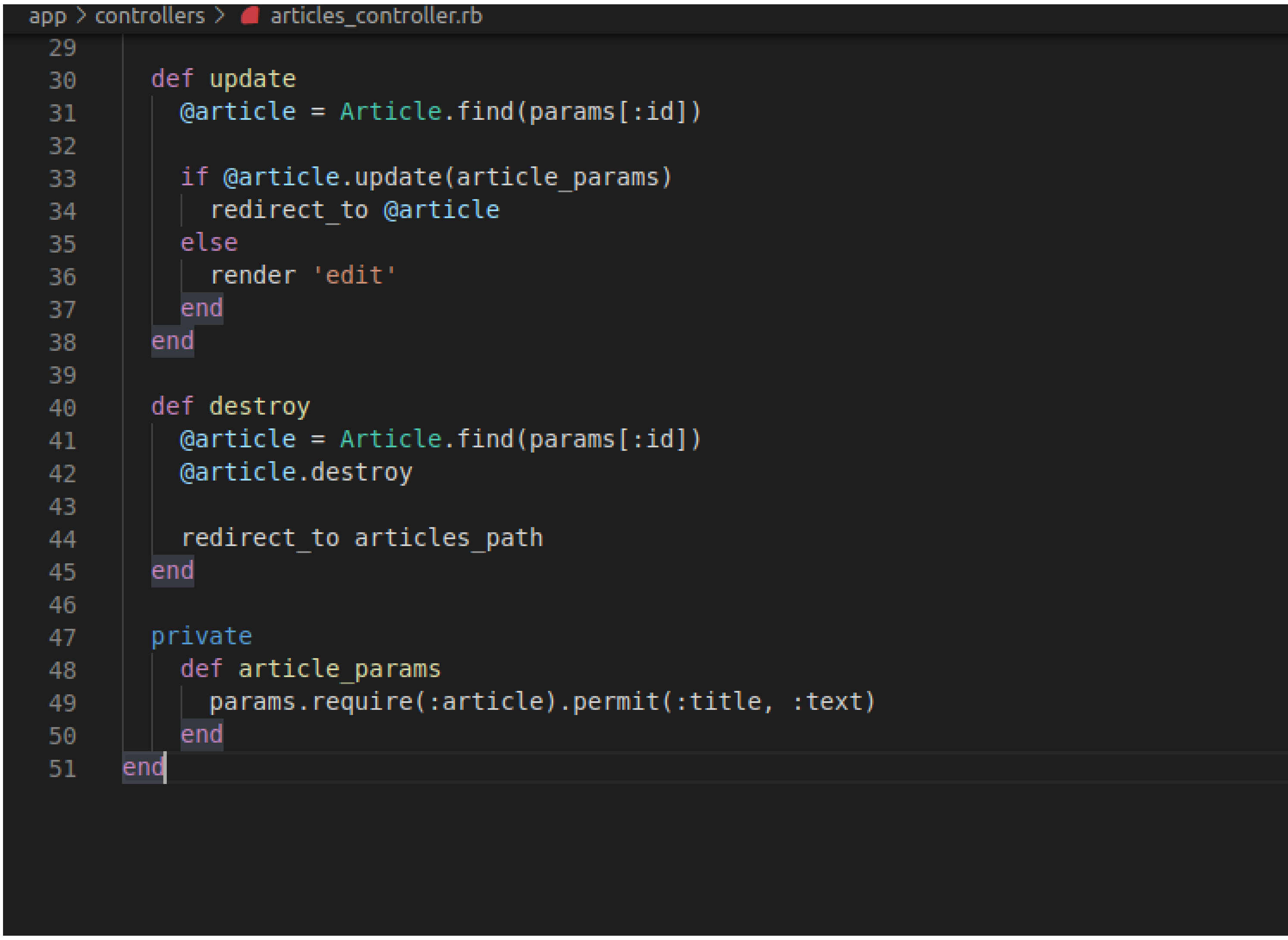
The ActionController is a module which handles the application logic of your program, acting as glue between the application's data, the presentation layer, and the web browser.

The following images are example of how controllers are created for a model. Here I have created controllers for articles model.



app > controllers > articles_controller.rb

```ruby
class ArticlesController < ApplicationController
  before_action :authenticate_user!, except: [:index]

  def index
    @articles = Article.all
  end

  def show
    @article = Article.find(params[:id])
  end

  def new
    @article = Article.new
  end

  def edit
    @article = Article.find(params[:id])
  end

  def create
    @article = Article.new(article_params)
    @article.user = current_user
    if @article.save
      redirect_to @article
    else
      render 'new'
    end
  end
```



```ruby

  def update
    @article = Article.find(params[:id])

    if @article.update(article_params)
      redirect_to @article
    else
      render 'edit'
    end
  end

  def destroy
    @article = Article.find(params[:id])
    @article.destroy

    redirect_to articles_path
  end

  private
    def article_params
      params.require(:article).permit(:title, :text)
    end
end
```

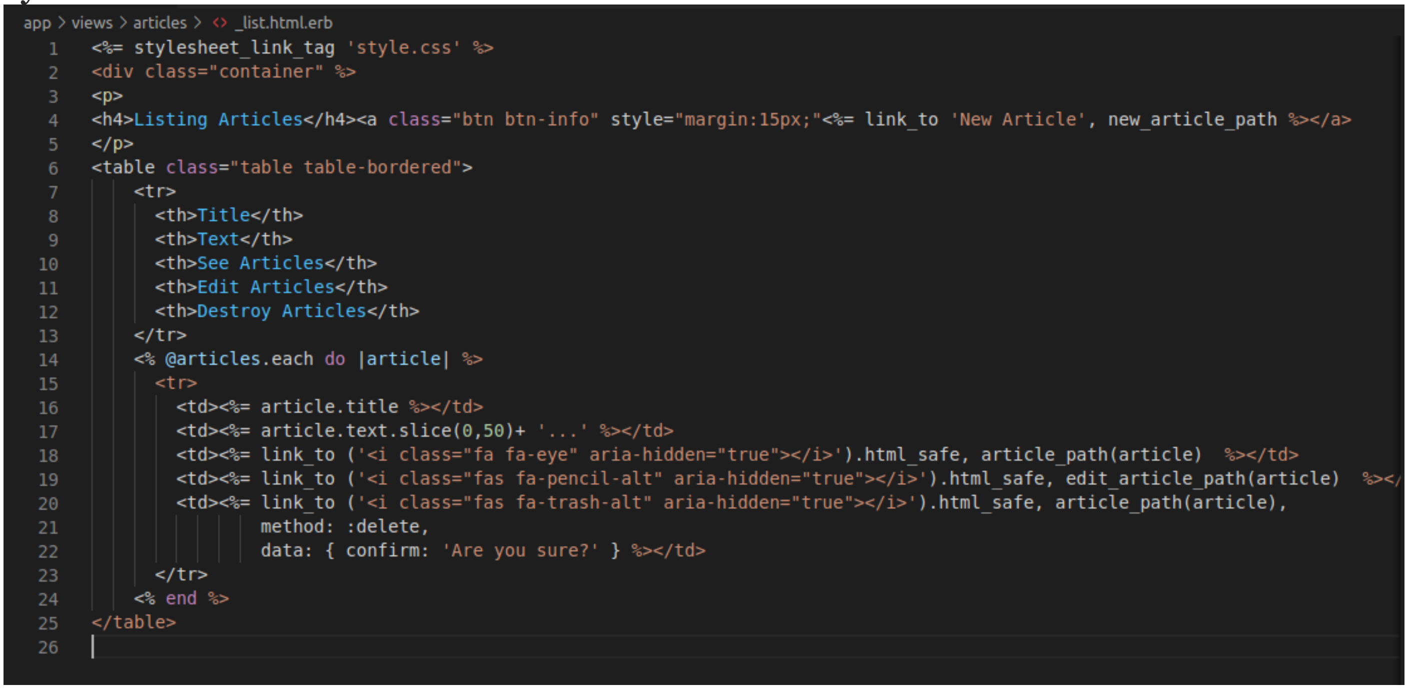
In this role, a controller performs a number of tasks including:

- deciding how to handle a particular request.
- retrieving data from the model to be passed to the view
- gathering information from a browser request and using it to create or update data in the model.

VIEWS

The principle of MVC is that a view should contain presentation logic only. This principle holds that the code in a view should only perform actions that relate to displaying pages in the application; none of the code in a view should perform any complicated application logic, nor store or retrieve any data from the database. In Rails, everything that is sent to the web browser is handled by a view.

A view need not actually contain any Ruby code at all — it may be the case that one of your views is a simple HTML file; however, it's more likely that your views will contain a combination of HTML and Ruby code, making the page more dynamic. The Ruby code is embedded in HTML using embedded Ruby syntax.



app > views > articles > _list.html.erb

```erb
<%= stylesheet_link_tag 'style.css' %>
<div class="container" %>
<p>
<h4>Listing Articles</h4><a class="btn btn-info" style="margin:15px;"<%= link_to 'New Article', new_article_path %></a>
</p>
<table class="table table-bordered">
    <tr>
      <th>Title</th>
      <th>Text</th>
      <th>See Articles</th>
      <th>Edit Articles</th>
      <th>Destroy Articles</th>
    </tr>
    <% @articles.each do |article| %>
      <tr>
        <td><%= article.title %></td>
        <td><%= article.text.slice(0,50)+ '...' %></td>
        <td><%= link_to ('<i class="fa fa-eye" aria-hidden="true"></i>').html_safe, article_path(article)  %></td>
        <td><%= link_to ('<i class="fas fa-pencil-alt" aria-hidden="true"></i>').html_safe, edit_article_path(article)  %></
        <td><%= link_to ('<i class="fas fa-trash-alt" aria-hidden="true"></i>').html_safe, article_path(article),
                method: :delete,
                data: { confirm: 'Are you sure?' } %></td>
      </tr>
    <% end %>
</table>

```

This views generates a list of all the articles created by user.

6b) Compare Controller, Views, Helpers, and Models in Ruby on Rails

Models:

Where the actual data lives. This is the direct link to the database, and is where data should be manipulated and altered.

For the most part, when you are dealing with code that relates to your data (such as saving, updating, or manipulating an object entry), place it in the model. These methods can easily be called upon and re-used in a DRY fashion from the controller or view.

Views:

What the end user sees when they interact with the app. This should have HTML and CSS specific code, along with some Ruby passed in through ERB tags.

A view should contain as little to no processing or calculating. This should be done through the model or controller, though at times that may not be fitting, and should be added to a Helper model.

Controllers:

What manages the communication between the View and the Model. The controller takes the data from the Model and prepares it for implementation within the view.

The controller should handle all of the prep work for handling what in the view the user will need to see. This can include setting instance variables of models, calling model methods to manipulate specific data, deciding if a user is logged in, and what to allow or not to allow within the view. The controller will take requests, and apply logic within the requests method to prepare the method data for the upcoming view. They should be skinny, and you should keep as much data manipulating functionality to the model as possible.

Helpers:

Helper methods don't manipulate data like methods within a model, but instead are a great way to extract common presentation logic from multiple views. Helpers are methods that are available to your views and encapsulate a common bit of code. They can be seen within a Controller, but are commonly housed within the

/app/helpers

folder. Because Helper methods are generally organized by controller (remember they help with refactoring heavily repeated bits of code), they would typically be housed under the file name:

object_helper.rb

If you need to build a function that will make creating the view easier, or more DRY, that is a great place to use a Helper. For example, if you are iterating over a list of objects frequently to display a particular piece of information, or see something within your views that is heavily repeated, those can go into a Helper. Helpers can be a great way to keep your views DRY, easier to update, and less bug prone.

7) Explain AJAX Patterns with suitable Examples

Submission Throttling

- Submission throttling solves the issue of when to send user data to the server.
- In a traditional web site or web application, each click makes a request back to the server so that the server is always aware of what the client is doing.
- In the Ajax model, the user interacts with the site or application without additional requests being generated for each click.
- In a traditional web solution data send back to the server every time when user action occurs. Thus, when the user types a letter, that letter is sent to the server immediately. The process is then repeated for each letter typed.
- The problem with this approach is that it has the possibility to create a large number of requests in a short amount of time, which may not only cause problems for the server but may cause the user interface to slow down as each request is being made and processed.
- The Submission Throttling design pattern is an alternative approach to solve this problematic issue.

- Using Submission Throttling, you buffer the data to be sent to the server on the client and then send the data at predetermined times.
    - Submission Throttling typically begins either when the web site or application first loads or because of a specific user action.
    - Then, a client-side function is called to begin the buffering of data.
    - Every so often, the user's status is checked to see if user is idle or not. If the user is still active, data continues to be collected.
    - When the user is idle, means not performing any action, it's time to decide whether to send the data. This determination varies such as, data is send only when it reaches a certain size, or data is sending every time when the user is idle.
    - After the data is sent, the application typically continues to gather data until either a server response or some user action signals to stop the data collection.
- **Example:** ***Google Suggest*** feature does this brilliantly. It doesn't send a request after each character is typed. Instead, it waits for a certain amount of time and sends all the text currently in the text box. The delay from typing to sending has been fine-tuned to the point that it doesn't seem like much of a delay at all. Submission Throttling, in part, gives Google Suggest its speed.

Predictive Fetch

- The Predictive Fetch pattern is a relatively simple idea that can be somewhat difficult to implement: the Ajax application guesses what the user is going to do next and retrieves the appropriate data.
- In a perfect world, it would be wonderful to always know what the user is going to do and make sure that the next data is readily available when needed. In reality, however, determining future user action is just a guessing game depending on user’s intentions.
- Suppose user is reading an online article that is separated into three pages. It is logical to assume that if user is interested in reading the first page, then user also interested in reading the second and third page.
- So if the first page has been loaded for a few seconds (which can easily be determined by using a timeout), it is probably safe to download the second page in the background. Likewise, if the second page has been loaded for a few seconds, it is logical to assume that the reader will continue on to the third page.
- As this extra data is being loaded and cached on the client, the reader continues to read and barely even notices that the next page comes up almost instantaneously after clicking the Next Page link.
- **Example:** This approach is taken by many web-based e-mail systems, including ***Gmail*** and ***AOL Webmail***; During the writing of an e-mail, its general e-mail is for someone whom user knows, so it's logical to assume that the person is already in user's address book. To help out user, it may be wise to pre-load user's address book in the background and offer suggestions.

Fallback Patterns

- We pre-suppose that everything goes according to plan on the server-side: the request is received, the necessary changes are made, and the appropriate response is sent to the client. But what happens if there's an error on the server? Or worse yet, what if the request never makes it to the server?
- When developing Ajax applications, it is imperative that you plan ahead for these problems and describe how your application should work if one of these should occur.

Cancel Pending Requests

- If an error occurs on the server, meaning a status of something other than 200 is returned, you need to decide what to do. Chances are that if a file is not found (404) or an internal server error occurred (302), trying again in a few minutes isn't going to help since both of these require an administrator to fix the problem.
- The simplest way to deal with this situation is to simply cancel all pending requests. You can set a flag somewhere in your code that says, "don't send any more requests."
- This solution has maximum impact on the **Periodic Refresh Pattern**.

Try Again

- Another option when dealing with errors is to silently keep trying for either a specified amount of time or a particular number of tries.
- Once again, unless the Ajax functionality is key to the user's experience, there is no need to notify user about the failure. It is best to handle the problems behind the scenes until it can be resolved.
- In general, the Try Again pattern should be used only when the request is intended to occur only once, as in a **Multi-Stage Download**.

8) Explain with example for handling multiple XMLHttpRequest objects in the same page.

```
<html>
 <head>
 <title>Sending Data to the Server</title>
 <script language = "javascript">
 var XMLHttpRequestObject1 = false;
 if (window.XMLHttpRequest) {
 XMLHttpRequestObject1 = new XMLHttpRequest();
 } else if (window.ActiveXObject) {
 XMLHttpRequestObject1 = new
 ActiveXObject("Microsoft.XMLHTTP");
 }
 function getData1(dataSource, divID)
 {
 if(XMLHttpRequestObject1) {
 var obj = document.getElementById(divID);
 XMLHttpRequestObject1.open("GET", dataSource);
 XMLHttpRequestObject1.onreadystatechange = function()
 {
 if (XMLHttpRequestObject1.readyState == 4 &&
 XMLHttpRequestObject1.status == 200) {
 obj.innerHTML = XMLHttpRequestObject1.responseText;
 }
 }
 XMLHttpRequestObject1.send(null);
 }
 }
 var XMLHttpRequestObject2 = false;
 if (window.XMLHttpRequest) {
 XMLHttpRequestObject2 = new XMLHttpRequest();
 } else if (window.ActiveXObject) {
 XMLHttpRequestObject2 = new
 ActiveXObject("Microsoft.XMLHTTP");
 }
 function getData2(dataSource, divID)
 {
 if(XMLHttpRequestObject2) {
 var obj = document.getElementById(divID);
 XMLHttpRequestObject2.open("GET", dataSource);
 XMLHttpRequestObject2.onreadystatechange = function()
 {
 if (XMLHttpRequestObject2.readyState == 4 &&
 XMLHttpRequestObject2.status == 200) {
 obj.innerHTML = XMLHttpRequestObject2.responseText;
 }
 }
 XMLHttpRequestObject2.send(null);
 }
 }
 </script>
 </head>
 <body>
 <h1>Sending Data to the Server</h1>
 <form>
 <input type = "button" value = "Fetch message 1"
 onclick = "getData1('dataresponder1.php?data=1', 'targetDiv')">
 <input type = "button" value = "Fetch message 2"
 onclick = "getData2('dataresponder1.php?data=2', 'targetDiv')">
 </form>
 <div id="targetDiv">
 <p>The fetched message will appear here.</p>
 </div>
 </body>
</html>
```

9) a) Explain any two Lists in BOOTSTRAP with examples.
b) Explain different ways to display code with Bootstrap

Unordered list

If you have an ordered list that you would like to remove the bullets from, add `class="unstyled"` to the opening `<ul>` tag (see Figure 2-9):

```
<h3>Favorite Outdoor Activities</h3>
<ul>
        <li>Backpacking in Yosemite</li>
        <li>Hiking in Arches
                <ul>
                        <li>Delicate Arch</li>
                        <li>Park Avenue</li>
                </ul>
        </li>
        <li>Biking the Flintstones Trail</li>
</ul>
```

Ordered list

An ordered list is a list that falls in some sort of sequential order and is prefaced by numbers rather than bullets (see Figure 2-10). This is handy when you want to build a list of numbered items like a task list, guide items, or even a list of comments on a blog post:

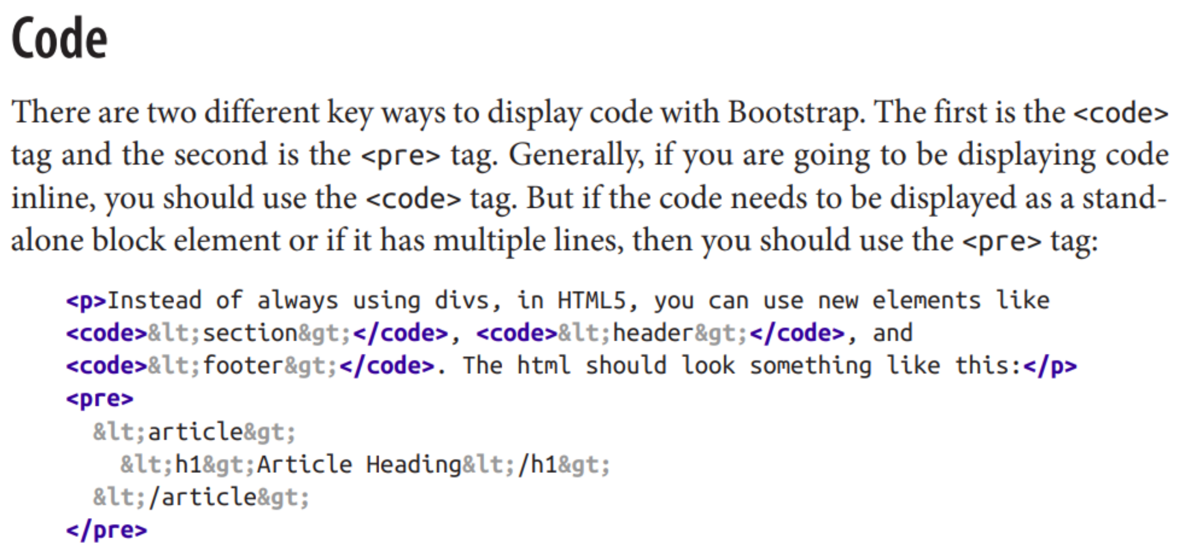
```
<h3>Self-Referential Task List</h3>
<ol>
        <li>Turn off the internet.</li>
        <li>Write the book.</li>
        <li>... Profit?</li>
</ol>
```

The third type of list you get with Bootstrap is the definition list. The definition list differs from the ordered and unordered list in that instead of just having a block-level `<li>` element, each list item can consist of both the `<dt>` and the `<dd>` elements. `<dt>` stands for "definition term," and like a dictionary, this is the term (or phrase) that is being defined. Subsequently, the `<dd>` is the definition of the `<dt>`.

A lot of times in markup, you will see people using headings inside an unordered list. This works, but may not be the most semantic way to mark up the text. A better method would be creating a `<dl>` and then styling the `<dt>` and `<dd>` as you would the heading and the text (see Figure 2-11). That being said, Bootstrap offers some clean default styles and an option for a side-by-side layout of each definition:

```
<h3>Common Electronics Parts</h3>
<dl>
        <dt>LED</dt>
        <dd>A light-emitting diode (LED) is a semiconductor light source.</dd>
        <dt>Servo</dt>
        <dd>Servos are small, cheap, mass-produced actuators used for radio
    control and small robotics.</dd>
</dl>
```

9 b)



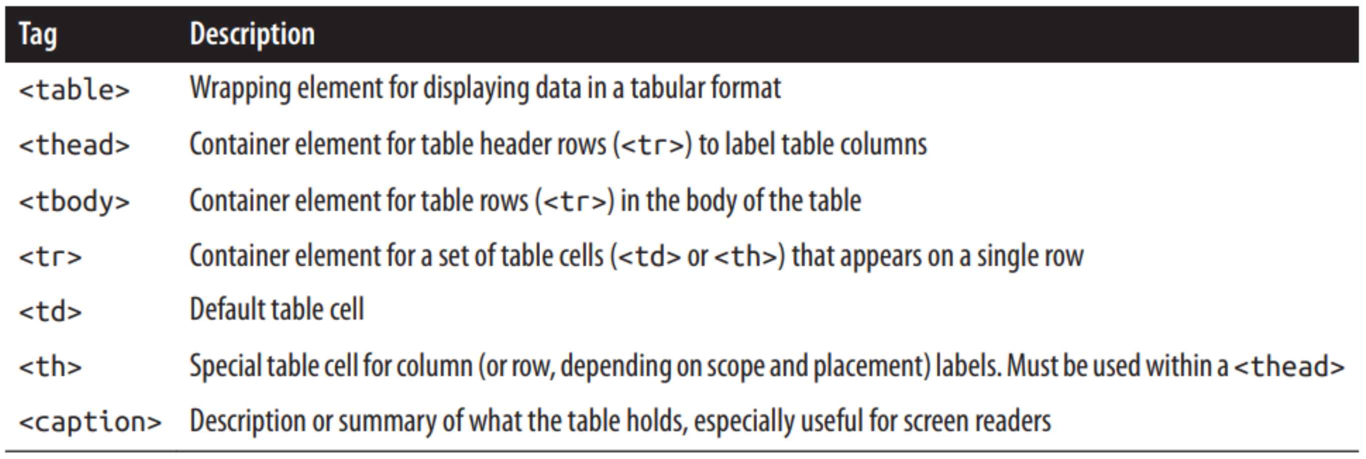
Code

There are two different key ways to display code with Bootstrap. The first is the `<code>` tag and the second is the `<pre>` tag. Generally, if you are going to be displaying code inline, you should use the `<code>` tag. But if the code needs to be displayed as a stand-alone block element or if it has multiple lines, then you should use the `<pre>` tag:

```
<p>Instead of always using divs, in HTML5, you can use new elements like
<code>&lt;section&gt;</code>, <code>&lt;header&gt;</code>, and
<code>&lt;footer&gt;</code>. The html should look something like this:</p>
<pre>
  &lt;article&gt;
    &lt;h1&gt;Article Heading&lt;/h1&gt;
  &lt;/article&gt;
</pre>
```

10) a)Explain Table elements supported by BOOTSTRAP with Example.
b)Describe Optional Table classes in BOOTSTRAP with Example

Table 2-1. Table elements supported by Bootstrap



Tag	Description
`<table>`	Wrapping element for displaying data in a tabular format
`<thead>`	Container element for table header rows (`<tr>`) to label table columns
`<tbody>`	Container element for table rows (`<tr>`) in the body of the table
`<tr>`	Container element for a set of table cells (`<td>` or `<th>`) that appears on a single row
`<td>`	Default table cell
`<th>`	Special table cell for column (or row, depending on scope and placement) labels. Must be used within a `<thead>`
`<caption>`	Description or summary of what the table holds, especially useful for screen readers

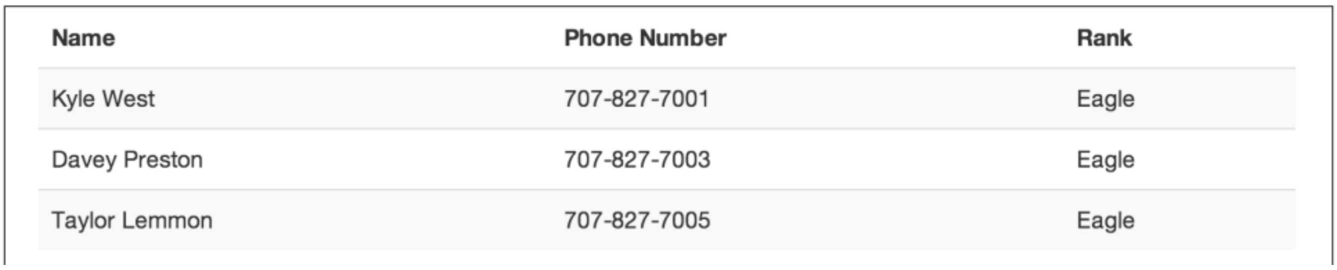
```
<table class="table">
  <caption>...</caption>
  <thead>
    <tr>
      <th>...</th>
      <th>...</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>...</td>
      <td>...</td>
    </tr>
  </tbody>
</table>
```

Optional Table Classes

Along with the base table markup and the `.table` class, there are a few additional classes that you can use to style the markup. These four classes are: `.table-striped`, `.table-bordered`, `.table-hover`, and `.table-condensed`.

Striped table

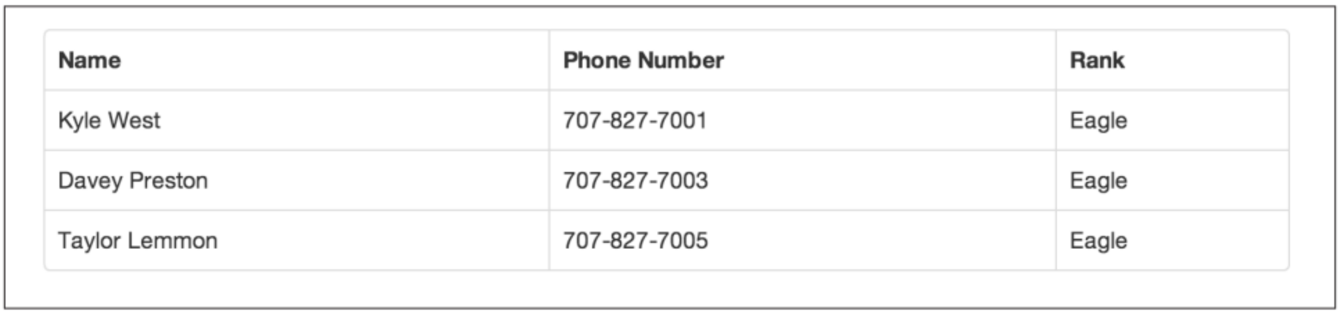
By adding the `.table-striped` class, you will get stripes on rows within the `<tbody>` (see Figure 2-14). This is done via the CSS `:nth-child` selector, which is not available on Internet Explorer 7–8.



Name	Phone Number	Rank
Kyle West	707-827-7001	Eagle
Davey Preston	707-827-7003	Eagle
Taylor Lemmon	707-827-7005	Eagle

Bordered table

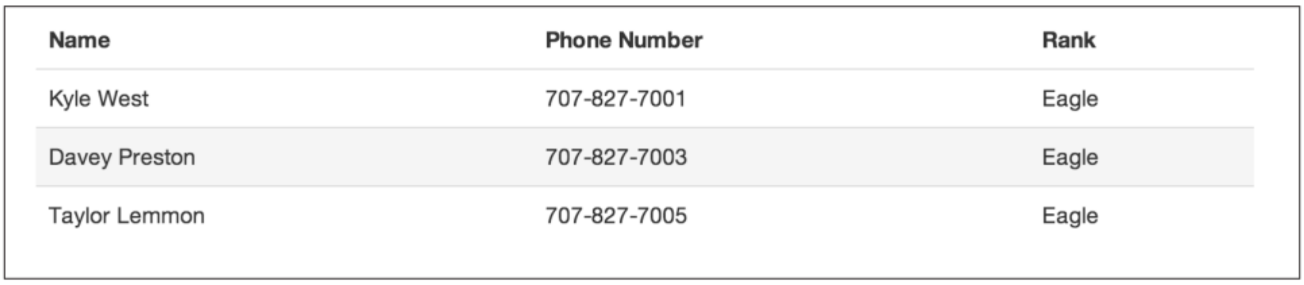
If you add the `.table-bordered` class, you will get borders surrounding every element and rounded corners around the entire table, as shown in Figure 2-15.



Name	Phone Number	Rank
Kyle West	707-827-7001	Eagle
Davey Preston	707-827-7003	Eagle
Taylor Lemmon	707-827-7005	Eagle

Hover table

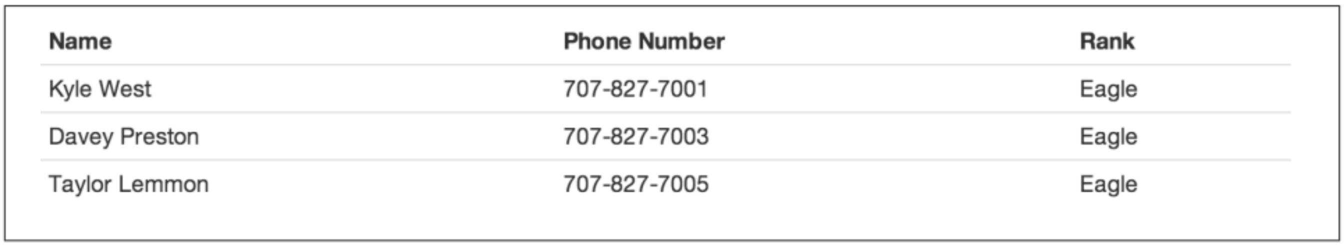
Figure 2-16 shows the `.table-hover` class. A light gray background will be added to rows while the cursor hovers over them.



Name	Phone Number	Rank
Kyle West	707-827-7001	Eagle
Davey Preston	707-827-7003	Eagle
Taylor Lemmon	707-827-7005	Eagle

Condensed table

If you add the `.table-condensed` class, as shown in Figure 2-17, row padding is cut in half to condense the table. This is useful if you want denser information.



Name	Phone Number	Rank
Kyle West	707-827-7001	Eagle
Davey Preston	707-827-7003	Eagle
Taylor Lemmon	707-827-7005	Eagle