Fall 9-12-2018

Programming for the Web: From Soup to Nuts: Implementing a complete GIS web page using HTML5, CSS, JavaScript, Node.js, MongoDB, and Open Layers.

Charles W. Kann III
None

Programming for the Web: From Soup to Nuts: Implementing a complete GIS web page using HTML5, CSS, JavaScript, Node.js, MongoDB, and Open Layers.

Description
This book is designed to be used as a class text but should be easily accessible to programmers interested in Web Programming. It should even be accessible to an advanced hobbyist.

The original goal behind this text was to help students doing research with me in Web based mapping applications, generally using Open Layers. The idea was to provide persistent storage using REST and simple http request from JavaScript to store the data on a server.

When teaching this class, I became painfully aware of just how little students know about Web Programming. They did not know how to format a REST request as a URL, or the methods that could be used when sending the requests to be processed. Even worse, they did not know that REST used URLs, or that there were different request types in URL.

Most could not tell me what the server was, or how it worked. Even fewer could tell me the different types of servers (e.g. Rails type, Sinatra type, EJB, etc), or what types of services they offered. They did not know how to access the server through a simple HTML api such as Postman.

Many did not know HTML, and a lot of those who knew something about HTML knew little beyond how to format text on a web page. There was very little JavaScript knowledge in the class, and most had not even heard of libraries such as JQuery, Vue, or Bootstrap.

These were all junior and senior computer science students, and the types of positions many of them were interviewing for would require this type of knowledge. Yes, they could learn it on the job. And I know this is not the foundational skills that many CS faculty believe students should learn. But there is a lot of good foundational material to be found in Web Programming, and I have had a lot of good feedback from students who have graduated or done an internship where they state even if they do not use all of this material, it is nice to be able to understand what others in the company are talking about.

Because of the experience with the REST interface, I petitioned the CS department of one of the schools I was an adjunct at to offer a class in Web Programming. This book is the result of having taught that class 3 times at 3 different schools. Its purpose is to provide an overview of how to program for the web. It is still largely client side, but that is something I hope to address in a future version of the textbook

This material in this textbook is ubiquitous in industry, and I really believe that there is utility in being able to communicate with others about the concepts without having 5-10 years' experience seeing all of the various pieces of a full stack application for the Web. The book is written to try to be as technology agnostic as possible, trying to emphasize concepts over implementations.

This book is also designed to help students understand how to use Web Programming with interfaces and libraries such as Open Layers, which is a mapping interface that can be run on a Web Browser client.

This book can be used as the main text for a class in Web Programming. It is not intended to be used as the only source of material in such a class, but as the guide to the class. Most of the material can be easily

Programming for the Web:

From Soup

To Nuts

Implementing a complete GIS web page using HTML5, CSS, JavaScript, Node.js, Mongo, and Open Layers.

Charles W. Kann

Last Update: Tuesday, Sept. 11, 2018

This book is available for free download from:
http://chuckkann.com/books/WebDesignFromSoupToNuts.doc.

Contact me at: chuck@chuckkann.com

Phone #: 7170-778-4176

Forward

This book is designed to be used as a class text but should be easily accessible to programmers interested in Web Programming. It should even be accessible to an advanced hobbyist.

The original goal behind this text was to help students doing research with me in Web based mapping applications, generally using Open Layers. The idea was to provide persistent storage using REST and simple http request from JavaScript to store the data on a server.

When teaching this class, I became painfully aware of just how little students know about Web Programming. They did not know how to format a REST request as a URL, or the methods that could be used when sending the requests to be processed. Even worse, they did not know that REST used URLs, or that there were different request types in URL.

Most could not tell me what the server was, or how it worked. Even fewer could tell me the different types of servers (e.g. Rails type, Sinatra type, EJB, etc), or what types of services they offered. They did not know how to access the server through a simple HTML api such as Postman.

Many did not know HTML, and a lot of those who knew something about HTML knew little beyond how to format text on a web page. There was very little JavaScript knowledge in the class, and most had not even heard of libraries such as JQuery, Vue, or Bootstrap.

These were all junior and senior computer science students, and the types of positions many of them were interviewing for would require this type of knowledge. Yes, they could learn it on the job. And I know this is not the foundational skills that many CS faculty believe students should learn. But there is a lot of good foundational material to be found in Web Programming, and I have had a lot of good feedback from students who have graduated or done an internship where they state even if they do not use all of this material, it is nice to be able to understand what others in the company are talking about.

Because of the experience with the REST interface, I petitioned the CS department of one of the schools I was an adjunct at to offer a class in Web Programming. This book is the result of having taught that class 3 times at 3 different schools. Its purpose is to provide an overview of how to program for the web. It is still largely client side, but that is something I hope to address in a future version of the textbook

This material in this textbook is ubiquitous in industry, and I really believe that there is utility in being able to communicate with others about the concepts without having 5-10 years' experience seeing all of the various pieces of a full stack application for the Web. The book is written to try to be as technology agnostic as possible, trying to emphasize concepts over implementations.

This book is also designed to help students understand how to use Web Programming with interfaces and libraries such as Open Layers, which is a mapping interface that can be run on a Web Browser client.

This book can be used as the main text for a class in Web Programming. It is not intended to be used as the only source of material in such a class, but as the guide to the class. Most of the material can be easily supplemented by information on the Web.

A suggested 14 week class at an advanced undergraduate level would be as follows:

Week	Chapter	Topics
1	2	Basic HTML, and form definition using HTML
2-3	3	Processing a form in html: 1. Procedural programming, decisions and loops 2. Functions 3. Events 4. Onload event to set event listeners 5. Unobtrusive JavaScript 6. Immediately Invoked Function Expressions (IIFE) 7. JQuery 8. Form processing 9. Lambda functions and functional programming
4	4	CSS
5-6	5	JavaScript objects: 1. JSON 2. Constructor Functions 3. Prototypes 4. Functions and Closures 5. Object Oriented Models in JavaScript
7-9	6	CRUD interfaces 1. Designing an interface 2. Designing the data objects 3. Implementing a simple CRUD interface 4. Persistent storage to local storage Persistent storage using a server 1. Node.js 2. Sails 3. REST 4. HTTP requests
10-11	7	Map applications using Open Layers
11-14		Group or individual projects

This book is currently almost done. The last section of the last chapter is mostly just code for the map application using persistent storage, there really isn't a lot of discussion of the application. However, I need to have the text out this week for a class, and I will probably not get a ton of time to work on it for the semester. I believe the rest is useful enough that I wanted to release it.

I have plans for a number of additions to the book. These include more use of ECMA6+ concepts I think will be useful (modules, arrow functions, etc), an extension to the map application using more unstructured data, and the inclusion of more server-side functionality, including linking and using NoSQL capabilities of MongoDB and possibly JSON in MySQL.

Table of Contents

Part I: The Basic Components of Web Pages

Part I of this text is an introduction to the basic concepts needed for this textbook. Part I of the book provides a first introduction to HTML, CSS, and JavaScript. It covers basic HTML, CSS, and JavaScript syntax, the basic functionality of JQuery, and how to handle events in JavaScript. It also introduces JavaScript lambda functions, and how to implement events in unobtrusive JavaScript by using Immediately Invoke Function Expressions (IFFE) in an onload event function. It will end with a brief introduction to Functional Programming.

Chapter 1 Introduction

Chapter 1.1 About this Book

This book is intended for readers who have an understanding of some computer programming language, such as Java, C#, C++, or Python, and want to learn how to create map applications for the World Wide Web (or just Web). It was written for Junior Computer Science (CS) college students[1]. These students will have had at least 1-2 courses in one programming language, such as the ones mentioned above, as well a class in Data Structures. Most of these students will not be familiar with any technologies for the programming for the Web.

Some non-programmers will argue that all is needed for web pages is a Content Management Systems (CMS) such as Word Press, Joomla, Scala, or any number of tools targeted at non-programmers. While it is possible to create perfectly reasonable web sites with a CMS, these tools are designed to manage content and are thus very limited. The core technologies for Web browsers are Hyper Text Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript[2], and to access any real power for a web application requires that a programmer be familiar with these technologies. Even end users using a CMS often need to access HTML, CSS, or JavaScript for some specific functionality.

The need to know the core technologies of the web, HTML, CSS, and JavaScript, is also necessary for programmers who are using other languages for Web development such as C#, Python, or Java. Even though these languages provide templating engines, the output from these engines is HTML, CSS, and JavaScript. To understand the input and output of templating engines, programmers should know about the target Web core technologies that these templating tools target.

The style of the book will not be like a traditional textbook. It will have problems at the end of each chapter, but those are so the reader can test their knowledge of the subjects presented and learn the concepts at a deeper level than I can present in the written portion. The tone of the book is also more conversational, and the format more like a tutorial than a textbook. This is in keeping with my style of interacting with students.

This text will follow material to develop one application, a simple Map Viewer in Open Layers, from *soup to nuts*. The phrase "soup to nuts" is an American English idiom which means "the whole thing". It comes from the idea that a traditional American meal starts with a soup course and ends with sweets and nuts. This book was written with this paradigm in mind. It intends to present just enough material that a student can learn the languages, tools, and concepts needed to implement a mapping application. The mapping application presented is the starting point for students doing research with me on mapping projects.

[1] The specific purpose for writing this book was to prepare students who wanted to do mapping projects in JavaScript. While writing this book, it has been used in various stages of development to teach classes on Enterprise Computing.

[2] The general consensus of the Web development community is to move away from JavaScript to Web Assembly. At the time this was written (July 2018), the specification for Web Assembly is still being developed (https://webassembly.org/roadmap/). Even once it is completed, there will be support for JavaScript in Web Assembly. So, JavaScript will remain a core technology for the Web for a long time.

Chapter 1.2 What you will learn

The purpose of this text is to present more than just how to create a web site. It specifically looks to introduce students to the following topics:

1. Implementation of a complete Web Create-Read-Update-Delete (CRUD) application, from *soup to nuts*. It will start defining the basic functionality of an application, and in a step wise fashion implement the final application. All the technologies and steps needed to do this will be covered, though not in detail.

   This book is designed to make up for what it lacks in depth with what it includes in breath.

2. A basic overview of all the technologies needed to implement a web page. After completing this book, you will be able to implement simple CSS, HTML, JavaScript, JQuery, a CRUD interface, a simple REpresentational State Transfer (REST) interface, and use OpenLayers and the other tools to create a simple map program that goes beyond what most programmers can do in Google Maps.

   This book is not intended to be about any one technology. After completing this book, the reader will still have to learn technologies they will use in detail. However the reader should know a little about each of the technologies, and how they all fit together. In most organizations this is important since most programmers are not responsible for all parts of a large project but must know how their part fits in to the bigger picture.

3. How to use events in JavaScript programs. You will know that the HTML DOM exists, and that it contains a lot of useful information. You will be able to use some standard JavaScript libraries like JQuery. You will be able to use OpenLayers to implement GIS and other image-based layering programs. You will be able to build a Node.js server with a framework. You will know what a NoSQL database is, and some of the reasons to choose between a NoSQL and Relational database.

   This book will not prepare anyone to interview for positions in these technologies, but you will have a background that will make learning them easier. And you will understand the environment surrounding these tools.

4. How to design, write and use mapping applications, which is an area of interest to me, and why this book was written. Juxtaposing symbolic information in a data space to highlight the interplay that the information is fascinating to me.

   The reader with note that what I refer to as a map is far from the more traditional view of a map. The one I normally use comes from Wikipedia and defines a map as "a symbolic depiction highlighting relationships between elements of some space.[3]" Note that nowhere does this definition mention geography, which is what most people think about when discussing a map.

   Even within geography, there are multiple different types of maps. For example:

[3] https://en.wikipedia.org/wiki/Map

a. Printed maps, such as would be found in an Atlas. These will be called referred to as static geographic maps.

b. Graphic Information Systems (GIS), such as Google Maps. These maps allow a user to examine a geographical space from multiple zoom levels, and search for and place markers. They will be referred to as GIS and are more interactive than their static cousins.

c. Maps not intended to be to scale and are thus not true linear geographical maps. These maps will be referred to as non-linear maps. Examples are subway and metro maps, bus routes, directions from a friend, and many historic maps. These maps are arguably more common and less well defined than linear maps.

Constraining maps to geography is far too limiting. Maps can be pictures, where each person in the image is at a point on the map. Maps can be networks, representations of data, ways to solve a problem, or possible outcomes of events such as what must happen for a team to make the playoffs. When implementing maps in this textbook the OpenLayers opensource program will be used. OpenLayers works with any GIS maps server for geographic data, but by its design of layers on images offers far more possibilities than GIS mapping software products.

Chapter 1.3 What is in this Book

As to how to use this book, it is divided into sections, and each of those sections is further divided into chapters. The overall layout of the books is as follows:

I. The Basics Components of Web Pages

This section covers the basic elements needed to create a web application. An overview of HTML, CSS, and JavaScript, up to the ability to create and process a simple form, will be covered. The topics will be:

Chapter 3 HTML

1. HTML Text, Tags, and Attributes
2. Standard HTML Tags
3. Document Structure Tags and Simple Web Page
4. Creating Forms in HTML

Chapter 4 JavaScript

5. How to insert JavaScript into an html file.
6. Getting input and output from JavaScript.
7. Commenting in JavaScript.
8. Variables and arrays.
9. The procedural constructs if, for, and while
10. Functions and iterators
11. Events and event processing using callback functions.
12. Principals and usage of Unobtrusive JavaScript and IFFE.

Chapter 5 CSS

13. CSS syntax
14. Styling sections of a web page
15. Styling a JavaScript form
16. CSS selectors
17. CSS and separate file management.

II. JavaScript Objects and a CRUD application

III. Implementing a Web Server using Node.js and MongoDb

IV. Mapping Application

The material in Part I of this text tends to be fairly simple for Junior/Senior CS students. I would expect that a reasonable Junior CS student could cover it in 2-3 weeks. The material in Part II is generally new to the students, and will require slightly more time, probably 3-4 weeks. The server material is mostly generated programmatically, so if the only the AJAX part of the application is covered without a lot of discussion about the server implementation, this section should take 2-3 weeks. The GIS portion using Open Layers should be able to be covered in 2-3 weeks. If an aggressive schedule is maintained through the material, there should be 3-6 weeks at the end of the semester for students to do a project of their choosing.

What you will learn

In this chapter, you will learn:

1. HTML text, tags, and attributes
2. HTML container tags
3. Standard HTML tags
4. HTML document structure
5. How to include an image
6. How to create form elements:
    a. Text Boxes
    b. Radio Buttons
    c. Check Boxes
    d. Buttons

Chapter 2 HTML

This chapter is designed as a brief overview of HTML. HTML is the language used to *mark-up* (or layout) Web pages. It consists of tags which are embedded in strings of text. These tags are instructions in a web page to control things such as formatting. For example, the *emphasis* (`<em>`) tag is used to provide emphasis to a string, and the *strong* (`<strong>`) tag is used to bold text.

The evolution of HTML has caused it to be much more than a program that can format documents. It can be used to include information for other languages. For example, the `<script>` tag can be used include JavaScript source code within the current document, and the `<style>` tag can be used to include an external file containing Cascading Style Sheets (CSS). It can be integrated with these other languages to then be used as an infrastructure for write complex programs, such as form-based systems, mapping systems, and other useful programs that can be run from a browser.

This chapter will cover the basics of creating simple HTML web pages and creating the HTML portion of an interactive form. In subsequent sections of Part I of this book CSS will be used to style the form, and JavaScript will be used to provide interactivity with the form and to process the form.

Chapter 2. 1 HTML Text, Tags, and Attributes

HTML was derived from Standard Generalized Markup Language (SGML). SGML was designed as a *markup* language, to allow a writer to markup (or annotate) a document. Markup languages have been around since at least the 1970's, when the author used one on a DECSYSTEM-10 to format school papers. Perhaps the most popular pure markup language still in use is LaTeX, which is used for mathematical, scientific, and engineering documents.

The idea behind a markup language is that a document could be *marked-up* with tags to tell a program processing the input how to render the text. For example, the following HTML code:

```
</center>The <i>quick</i> brown fox jumped over the <strong>lazy</strong>
dog</center>
```

would be rendered as:

```
                    The quick brown fox jumped over the lazy dog
```

Markup languages were the precursors of word processing programs that became popular in the 1980's with the PC revolution. The word processing programs, as they used a What-You-See-Is-What-You-Get (WYSIWYG) interface, which is much easier for a novice computer user to interface with than a markup language. WYSIWYG editors eventually and took over the market for word processing, with a few exceptions such as LaTeX, as mentioned earlier.

SGML was originally a traditional markup language, and hyperlinks between documents were added to create HTML. In the beginning the purpose was to link physics papers together in a web of documents. HTML started with with a browser introduced at CERN in 1990, HTML has expanded far beyond the wildest vision of its creators, but still maintains its markup character.

HTML still consists of text and tags. Over time, HTML has evolved from its roots, and is no longer seen simply as a way to format a document. The HTML language is now used to define the content (or contextual meaning) of items on a web page, and the tags have evolved to represent this new role. Most of the original tags specifying how to format text, such as *bolding* (`<b>`), *centering* (`<center>`) or *italicizing* (`<i>`), are now considered obsolete and their use is discouraged. Bolding is now done by the content tag *strong* `<strong>`, and italicizing is done by the content tag *emphasis* <em>. Formatting is done based on the content tags using CSS, and interactivity is defined using embedded JavaScript programming using the <script> tag. HTML has become much more than a simple markup language, but to understand HTML it is important to understand its roots as a markup language.

The tags in HTML are key words defined between a less than sign (<) and a greater than sign (>), though when using HTML, it is more common to call them *angle brackets*. Between the angle brackets are HTML tags and attributes. For example, the HTML tag to bold text is the word *strong*, so to bold text the *`<strong>`* tag would be used. The tag represents actions to be taken by the program that processes the marked-up text.

HTML tags are not case sensitive, so the tags `<i>` and `<I>` are equivalent.

Most tags are applied to a block of text, and apply to the block of the text they enclose. All tags are closed using a slash (*/tag*), as in the example above where *`<strong>lazy</strong>`* caused the word *lazy* to be bolded.

The tags shown above are called *block*[4] tags in that the first tag (e.g. `<i>`) specifies where to begin italicizing the text, and the closing tag (e.g. `</i>`) specifies where to stop indenting the text. The text or other information between the two tags is called a *block*.

[4] The terms block and container tags are often used interchangeably in HTML. This text will make a distinction between a block and container tag. Semantically it is easier to refer to a block tag as referring to an attribute to apply to all the elements in the block, such as italicizing or bolding the text in the above example. Other tags, such as the *div* or *table* tag contain other HTML elements and will be called container tags.

Sometimes a tag, such as a break tag (`<br>`) or image tag (`<image>`) are *empty*, in the sense that they simply run a command, and do not apply an attribute to the text. In the case of the `<br>` tag, the meaning is to simply skip a line, so it does not affect any text or any other element. It could be written as `<br></br>`. However, HTML provides a short cut for this type of tag. The tag can be closed between the angle brackets that opened it. The `<br>` tag can be written as `<br/>`.

As the br tag shows, tags can be used to do many things in HTML other than just markup text. For example, tags can be used to tell the HTML processor to include a picture. If a picture exists in the same directory as the web page, the image can be included on the page by adding the following tag into the HTML for the page:

```
<image src=”dog.jpg” />
```

Program 1 = Image Tag

This line of code includes the picture from the file dog.jpg in the web page. The tag is the image tag, but the image needs an *attribute* to indicate where to find the picture. For the image tag the attributed used to find the picture is the *src* tag.

Attributes are data that fill in details needed to implement the desired behavior for the tag. All tags can have some attributes, and these will be looked at in more detail later.

Chapter 2. 2 Standard HTML Tags

There are 4 HTML tags that are considered standard for all web pages. These tags are the <html>, <head >, <title>, and <body> tags. A strict HTML5 web page is required to have these 4 tags, and many IDE’s will automatically insert these 4 tags in a page for you when you start an HTML page. Since these 4 tags are always recommended for every web page, I personally keep a template, shown below, that I copy when I begin all web pages.

```
<html>
    <head>
        <title>Please change this to the title of your page </title>
    </head>

    <body>
    </body>
</html>
```

Program 2 – HTML template

This template code can be represented as a top-down tree, as shown below. In this tree the html tag is used to contain 2 elements, the head and the body. Likewise, the head section contains the title, and as we will see shortly, the head and the body sections will contain many other HTML elements. Thus, we will call these 3 tags *container* tags. The title only contains a block of text, so it is a block tag.

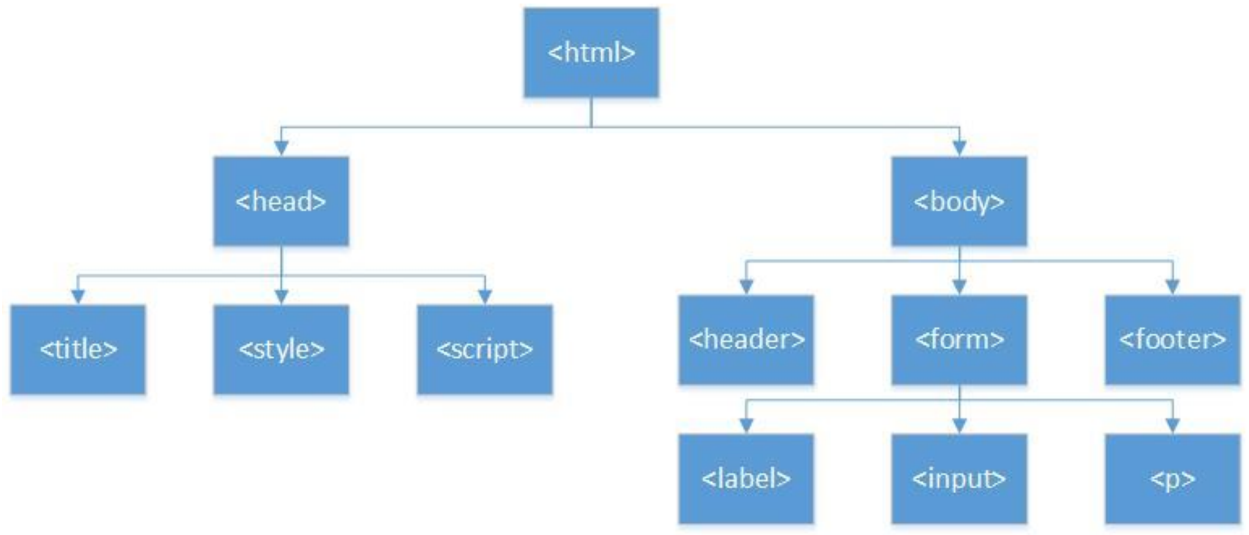


Figure 1 - Tree layout of an html document

These four tags (html, head, title, and body) are special in that they define the structure of an HTML document and are called Document Structure tags. This will be covered more fully in the next section. But first there are some points to be made about how to structure HTML files.

In the file in Program 2, note that each container tag (html, head, and body) is indented to show the hierarchical structure of the document representing the tree in Figure 2.1. This is not required by the HTML processor, as the processor is just looking at strings of instructions and text and ignoring any program format. However, indenting makes it easier for the developer and maintainer of web pages to understand what is going on in the program[5].

The second thing I always recommend writing html code is to end all container tags when the beginning tag is entered. This means when <head> is entered, the </head> is immediately entered. This is the automatic behavior of many IDEs. The reason to enter a close tag when opening a container tag is to enforce boundaries on the ideas and concepts that are being expanded in the container. This does not make sense to many novices, who seem to see ideas as unstructured information that starts at the top of the document and just streams to the end. Novice ideas often appear (to me) to be a jumble of thoughts. They do not see a purpose in creating boundaries or structure to express of idea. This is true in all areas of academia, including unreadable papers and documentation. This is why indenting, and container boundaries are so important to enforce a structured way of presenting the ideas. And why a basic course in CS, which teaches this structuring, can be important for students of any major.

But since this concept of structuring ideas is such an enigma to students, I give a practical reason for entering the enter the closing tag when the opening tag is entered. If the closing tag is not

[5] I make it a point to never help a student with poorly formatted code, as it is frustrating to me to try to understand, and simply insults my sensibilities. The student must format it correctly, then I will help them work on their programs. The most useful outcome of this policy is the vast majority of the time is it results in a comment from the student, “never mind, I figured it out”. Poor formatting nearly always represents confusion about the bounds of specific content and fixing the formatting fixes the confusion.

immediately entered, it is likely to be completely forgotten and lead to other problems. Though the best reason for students seems to be so they don't lose points on a test.

Chapter 2. 3 Document Structure Tags and A Simple Web Page

An HTML document is divided into two main sections, the *head* and the *body*. The reason for this division is that the head is to contain metadata, and the body is to contain the information to be displayed on the web page.

To understand this difference, it is important to understand the meaning of the term *metadata*. According to Dictionary.com, the meta prefix means: "a prefix added to the name of something that consciously references or comments upon its own subject of features"[6]. Hence metaphysics is a physics about physics, a meta-analysis is a study of studies, etc.

Metadata is what its name implies, data about the data on a web page. It defines how the page is to interpret the data which it will process. For example, functions that are used in a web page are defined in the head. How to handle events and interpret the CSS tags are also defined in the head. Anything that is used to define the behavior of the page is in the head of the document.

As important as what is in the head is what is not in the head of the HTML document. The head should not output any information (or data) to be placed on the web page. Functions and other structures defined in the head should return strings to be printed in the body, and not printed to the page in the head. If the statement is defining something to be rendered on the page itself, it does not belong in the head.

This implies (correctly) that the body of an HTML document should contain anything that is rendered and placed on the web page. Any text to be displayed, images to be rendered, or forms to be processed belong in the body of the document. And again, the body should not contain any metadata such as functions, CSS, or code to handle events.

Nothing in HTML enforces this policy, but there are few good reasons to violate it. And when the data in the head and body are mixed, it generally shows that the programmer did not have a clear concept of what the page is to do.

In the Document Structure, the <title> tag is shown as metadata. This is because the title is what appears on the tab in the browser, and not rendered on the page.

Program 3 is a simple HTML web page to illustrate the concepts covered so far. Note that the program uses the large heading (<h1>), and paragraph (<p>) block tags, and the <image> tag, which have not been covered. As has been stated before, this text is not to be a text on learning html, CSS, JavaScript, or any other language or program. It intends to provide enough detail to allow a motivated intermediate programmer, specifically students doing research with myself, enough background to start that research. A complete list of HTML tags can be found at: https://www.w3schools.com/tags/, and many tutorials exist on how to use them in web pages. Readers interested in more functionality of the tags can easily look them up on the WWW[7]. But

[6] https://www.dictionary.com/browse/meta?s=t
[7] In my experience, the best search engine by far for looking up information when programming is Google. Other search engines tend to assume a context for the terms and bring up a lot of noise pages not related to programming.

it is expected that the readers of this text are sufficiently advanced that they can research and learn the implementation details of this type of material.

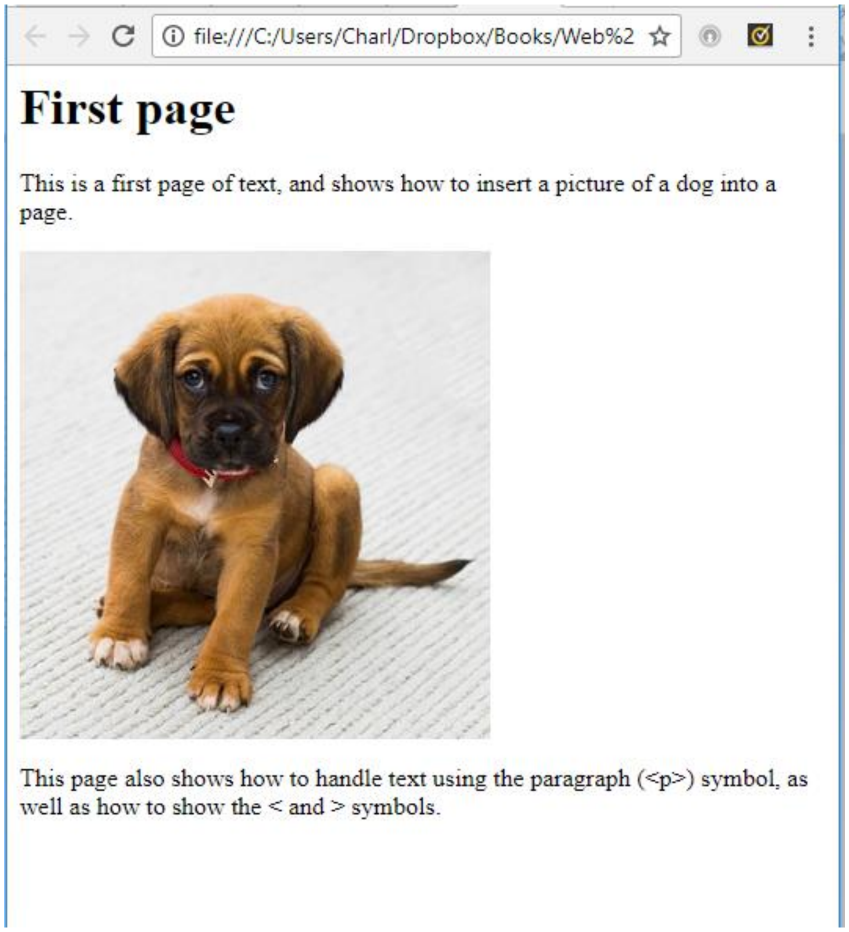
Enter Program 3 is entered into a file with a ".html" extension. Note that the file must have some form of a .html (e.g. .htm, etc.) extension for the browser to recognize it as an html file. Place a jpeg picture (any picture) into a file named `dog.jpg`, and open the file in a browser such as Chrome, Firefox, Safari, IE, or MS Edge[8]. You should get a page similar to Figure 2-2.

```
<html>
      <!--
          Author:  Charles Kann
            Date:        5/17/2017
            Purpose: A first example of an HTML program
      -->
      <head>
          <title>First HTML Web Page </title>
      </head>

      <body>
          <h1>First page</h1>
          <p>
                This is a first page of text, and shows how to
                insert a picture of a dog
                into a page.
          </p>
          <image src="dog.jpg/>
          <p>
                This page also shows how to handle text using the
                paragraph (&lt;p&gt;)symbol, as well as how to show
                the &lt; and &gt; symbols in html text.
          </p>
      </body>
</html>
```

Program 3 – First HTML Web Page

In this program, comments in html begin with a (<!--) tag and continue until a (-->) tag. There is a comment at the start of this document to provide a preamble comment for the file. The need for file preamble comments, and commenting code correctly, is stressed in every introductory programming course I have ever encountered. However, it seems as though students believe such commenting is not useful, only applies to introductory classes and/or the first language they learned. They throw out these lessons as soon as they think they can safely get away with it. That is why at every level student programs need to be graded on commenting, and a poorly commented program by a senior should be given an F, even if it works. Commenting is not something to be avoided. It is always good practice, and it will be good practice in web development also.

[8] For simple HTML, the choice of the browser matters very little, though each browser will likely render the page differently. When beginning JavaScript, the browsers are very different, and some do not implement the JavaScript standard correctly, and will fail on valid JavaScript code. I suggest using the latest Chrome or Firefox browsers, and all of the material in this text has been tested to work with Chrome.


file:///C:/Users/Charl/Dropbox/Books/Web%2

First page

This is a first page of text, and shows how to insert a picture of a dog into a page.

This page also shows how to handle text using the paragraph (<p>) symbol, as well as how to show the < and > symbols.

Figure 2 – Output from the first Web Page

Chapter 2.3. 1 Quick Check

1. What symbol is used to start an HTML tag. What symbol is used to end an HTML tag.
2. What 4 tags should you use in all HTML documents?
3. How do you close an HTML block tag? How do you close an empty tag?
4. What is a tag? What is an attribute?
5. What tags in the web page?
6. What are document structure tags?
7. What tags should be present in all web pages? What are they used for?
8. Give some examples of tags that have attributes. What are the attributes?
9. What happens to text that spans across multiple lines in the HTML source file?
10. What do you think the &lt; and &gt; symbols do? What other symbols do you think are often specified this way in HTML.

Chapter 2. 4 Creating Forms in HTML

The next part of this chapter will show how to implement form elements in HTML. Form elements are things like labels, checkboxes, radio buttons, textboxes, buttons and other elements often use to interact with and gather information from users. Form elements are a major method of creating user interaction with a web page.

All form elements contain a name or id which can be accessed from JavaScript. All form elements have a set of attributes (or properties) that can later be used in JavaScript and CSS to manipulate them.

This section will only implement the form elements. To implement interactivity in a form normally requires JavaScript and will be covered in a subsequent chapter.

Chapter 2.4.1 The <label> tag

The `<label>` is used to put a label on the page and associate it with another HTML element. Since the `<label>` tag simply puts text on the screen, why use a label rather than just putting the text into the HTML? For example, the following two HTML examples will appear the same to the user:[9]

```
<label id="label1" for="title">Title: </label>
<input type="text" id="title" size="20" />
```

Program 4 – Using a label tag

```
Title:
<input type="text" id="title" size="20" />
```

Program 5 – Using no label tag

These two examples will appear the same on a web page, with the string "`Title:`" followed by a textbox. There are two advantages to using the label tag. The first is that when using a `<label>` tag, the label itself can be given an `id`, and that `id` can be used later in JavaScript to manipulate and modify the label.

The second advantage of a `<label>` is it can be used to reference an `<input>` value by setting the *`for`* attribute to the `id` of the field it corresponds to. When the `for` attribute is set in a label, clicking on the label places the cursor in the corresponding field.

Complete documentation for the `<label>` tag can be found at: https://www.w3schools.com/tags/tag_label.asp.

Chapter 2.4.2 The <input> tag

The `<input>` tag tells the HTML interpreter program that the data to follow defines a user interaction that can be queried later for content. The type of user interaction will be specified in the *type* attribute. The different types of input types can be found at https://www.w3schools.com/tags/tag_input.asp.

[9] Note that when using code fragments, the entire html file is not included. The <html>, <head>, <title>, and <body> tags are omitted. This is to save space and to emphasize the concept being introduced. This is no way sanctions or recommends ever dropping these tags in a html file.

There is one word of caution about many of the types that can be used as inputs. All of the types listed at the W3 Schools web site are not implemented in all browsers, and many of the types are implemented differently in different browsers. Using some of these types could (and likely will) lead to a very different user experience for a Chrome user and a Firefox or Safari user. The page designer should be aware of these differences, and plan for and test carefully when implementing a web page.

This section will only document a few input types. The ones that are selected are largely the ones which show how to implement interactions that are used in this textbook. The form implemented is one for adding maps to an application and will be used as an example through much of the text.

The input types to be looked at in the following sections are text, checkbox, radio button, number, date, and button.

Chapter 2.4.3 <input type = ”text”>

The first type of input we will look at is *`type=”text”`*, or the textbox. This is the default type for an input tag. If you expect some other type of field, and get a textbox, check and see if something is misspelled in the type of the input tag.

A textbox is an input box into which a user can type text, as was seen previously in Programs 4 and 5. The line:

```
<input type="text" id="title" size="20" />
```

Program 6 – Input text box

created a text box that contained 20 characters, and with an attribute `id` of name. Note that the `id` is a unique identifier for a field; it must be unique on the page. In a later example the name attribute will be introduced. A field can have a `name`, `id`, or both, and the field can be referenced by its `name` or `id`. A `name` is different from an `id` in that a `name` does not have to be unique on a page; many different elements can have the same name. In fact, in the radio button example, we will rely on the fact that the elements all have the same `name`. An id must be unique on a Web page.

While HTML is not case sensitive, the strings used for the `id` and `name` are case sensitive. Consider the following example.

```
<input type="text" id="title" size="20" />
<input type="text" id="Title" size="20" />
```

Program 7 – Text boxes different by case of name

In this case, two textboxes with different `ids`, one `title` and the other `Title`, are created.

The identifiers for `id` and `name` fields in this text will use standard camel casing for `ids` and `names`, with the first letter lower case, and lower-case letters in the rest of the identifier except for the first letter of new words (e.g. `myTitle`)

There are many attributes other than id and name that can be assigned to an input field, and the overview URL of the `<input>` tag above should be referenced to find them. For example, a Boolean value of readonly can be set on a text box to disallow changes via user input. The string contained in the text box is stored in a field named value and can be set or change in a JavaScript program, but the user cannot enter data into this field.

```
<input type="text" id="title" size="20" readonly value="My Map" />
```

Program 8 = readonly attribute

Chapter 2.4.4 <input type = ”checkbox”>

A checkbox is a box which allows a user to choose a single, discrete option. A checkbox is normally shown on a web page as a square box. To implement a checkbox, use an `input` tag and set the type attribute to `checkbox`. Normally the input tag will also have an `id` attribute, and a Boolean attribute `checked` can be added to indicate if the box is checked by default or not. Two formats for a check box, one checked and the other not, are shown below.

Note the use of the `<br/>` (break) tag in this program. The break tag is used to move the output to the next line.

```
<label id="label1" for="resize">Allow map to be resized: </label>
<input type="checkbox" id="resize"/>

<br/>

<label id="label2" for="recenter">Allow map to be recentered:" </label>
<input type="checkbox" id="recenter" checked />
```

Program 9 – Checkbox Example

Chapter 2.4.5 <input type = ”radiobutton”>

A radio button is similar to a checkbox in that it is a Boolean selection. It is different from a checkbox in that radio buttons form a group, and only one item in any group of radio buttons can be selected. For example, consider two sets of options in Program 10. The first set of options determines the type of map to display: an `XYZ Map` or a `Stamen Map`. The second set of options determines the size of the map (`600x480, 1024x768, or 1280x80`0). As shown in the following example, these form two groups by having the `name` common between the buttons in the group. In the first case, the `name` for both of the buttons is `mapType`, and the choice is between the `XYZ Map` or the `Stamen Map`. In the second case the `screenSize` is the name common between the buttons, and the choices are `600x480`, `1024x768` or `1280x800`.

Radio buttons are implemented in HTML by setting the input type attribute to *radio.* Radio buttons are normally round and are then grouped to show the mutually exclusive options. In HTML, the grouping of options is accomplished using name attribute. All radio buttons with the same name are in one group. The following code implements two groups of radio buttons. Note that even though it has no purpose right now, the value attributes are being set for these radio buttons. The values will be used in processing these radio buttons later in JavaScript.

```
<p>
    Type of Map<br>
      <input type="radio" name="maptype" id="XYZMap" value="XYZ Map"/>
      <label id="label1" for="XYZMap">XYZ map </label>
      <br/>
      <input type="radio" name="maptype" id="StamemMap"  checked />
      <label id="label2" for="StamenMap">Stamen Map </label>
</p>
<p>
    Screen Size<br>
      <input type="radio" name="screenSize" checked id="600x480"
         value="600x480"/>
      <label id="label3" for="XYZMap">600x480 </label>
      <br/>
      <input type="radio" name="screenSize" id="1024x768"
          value="1024x768"/>
      <label id="label4" for="XYZMap">1024x768 </label>
      <br/>
      <input type="radio" name="screenSize" id="1280x800"
          value="1280x800"/>
      <label id="label5" for="XYZMap">1280x800 </label>
</p>
```

Program 10 – Radio Button Example

Chapter 2.4.6 <input type = ”number” >

The input type `number` limits the input for the field to be numeric. The value can be a whole number or a decimal number, but only the numbers 0-9 and the decimal point can be entered (comma is not allowed). If the browser does not support a number type, the type defaults to a text, and a standard textbox is used.

The following example implements number fields for the `latitude` and `longitude` for the center of a map. Note that the entry is to be a fixed point (decimal) value.

```
<p>
    Center of Map<br>
      <label id="label1" for="lat">Latitude </label>
      <input type="number" id="lat"/>
      <label id="label2" for="long">Latitude </label>
      <input type="number" id="long"/>
</p>
```

Program 11 – Number Input Example

The number field can also be used to represent a range of value, as in the following example. Be careful however as a user can type in a number outside of this range. And while this statement implies that the value is an integer value, the user can enter decimal values.

```
<p>
      <label id=label3" for="age">Age</label>
      <input type="number" id="age" min="0" max="115" />
</p>
```

Program 12 – Integer Number Input Example

Chapter 2.4.7 <input type = ” date”>

The last form field covered is a `date`. The reason the date type is covered is to make a point that the browsers can be completely inconsistent in how they implement input types, particularly the input types that were implemented in HTML5, such as number and date. The format of entering the date, the return value from the date, whether or not a date picker is displayed, and the look and feel of the date picker if it is displayed are all browser dependent. The Quick Check questions at the end of this section will ask you to experiment with this field.

Only the most basic format of the date input type is shown here. It is suggested that the reader look at this date field across sever browsers to see how it is different in each one.

```
<p>
        <label id="label1" for="creationDate">Creation Date </label>
        <input type="date" id="creationDate"/>
</p>
```

Program 13 – Date input Example

Chapter 2.4.8 <input type = ”button”>

An input type of `button` creates a button. Buttons are normally placed on a form to trigger processing of the form when they are clicked. How the processing of the form is accomplished will be covered later in the chapter on JavaScript. For now, only the placing of the button on the form will be shown.

The button will have an id attributed so that actions can be assigned to the button in JavaScript. The text that is displayed in the button will be contained in the `value` attribute. A normal definition of a button would be as follows:

```
<input type="button" id=”processForm” value="Process Form" />
```

Program 14 – Button Example

Chapter 2.4.9 Final Example

The following example combines all of the elements seen in this chapter to create a single, working form. This form does not yet have any functionality, which will be introduced in the next chapter on JavaScript. The form is also not styled, which will be covered in the chapter on Cascading Style Sheets (CSS).

```
<html>
    <head>
          <title>Map Example Input Screen</title>
      </head>

      <body>
          <h1>Map Example Input Screen</h1>
            <p>
                <label id="l1" for="title">Title</label>
```

```
                  <input type="text" id="title" size="20">
            </p>
            <p>
                Map Options<br>
                  <label id="l2" for="resize">Allow map to be resized:
                  </label>
                  <input type="checkbox" id="resize"/>
                  <br/>
                  <label id="l3" for="recenter">
                      Allow map to be recentered:
                  </label>
                  <input type="checkbox" id="recenter" checked />
            </p>
            <p>
                  Type of Map<br>
                  <input type="radio" name="maptype" id="XYZMap"
                       value="XYZ Map"/>
                  <label id="label1" for="XYZMap">XYZ map </label>
                  <br/>
                  <input type="radio" name="maptype" id="StamemMap"
                      value="StamemMap" checked />
                  <label id="label2" for="StamenMap">Stamen Map
                 </label>
            </p>
            <p>
                  Screen Size<br>
                  <input type="radio" name="screenSize" checked
                      id="600x480" value="600x480"/>
                  <label id="label3" for="XYZMap">600x480 </label>
                  <br/>
                  <input type="radio" name="screenSize" id="1024x768"
                      value="1024x768"/>
                  <label id="label4" for="XYZMap">1024x768 </label>
                  <br/>
                  <input type="radio" name="screenSize" id="1280x800"
                     value="1280x800"/>
                  <label id="label5" for="XYZMap">1280x800 </label>
            </p>

            <p>
                  Center of Map<br>
                  <label id="label1" for="lat">Latitude </label>
                  <input type="number" id="lat"/>
                  <label id="label2" for="long">Latitude </label>
                  <input type="number" id="long"/>
            </p>

            <p>
                  <label id="label1" for="creationDate">Creation Date
                  </label>
                  <input type="date" id="creationDate"/>
            </p>

            <input type="button" value="Process Form" />
      </body>
</html>
```

Program 15 Complete Form Example

Chapter 2.4.10 Quick Check

1. What are the advantages of using a <label> tag rather than just using text in an HTML document?
2. What tag is used to ask for input from a user?
3. What are the different types attributes for the <input> tag? Which of these are newly defined in HTML5?
4. Explain a radio button group, and how it works.
5. What are the tags and attributes in the example form?
6. Run the final web page in at least 2 browsers, and document as many differences between how the form is rendered in each browser.

Chapter 2. 5 Conclusion

This chapter gives an overview of the HTML that will be necessary to continue using the rest of the text. The reader should now know what a markup language is, and why HTML is structured as it is. The reader should also be familiar with the sections of an HTML document, and what type of data go into the head and body of an HTML document. The differences between a tag and an attribute should also be clear.

Finally, the user should have a sufficient understand of HTML that they can implement a simple form in HTML.

Chapter 2. 6 Problems

1. Create a form for an application of your choice. The application can be for a realtor, a schedule at work, your favorite team's roster, or any topic you choose. Use as many form elements as you can, but at a minimum you must use text boxes, check boxes, and radio buttons.

What you will learn

In this chapter, you will learn:

1. Inserting JavaScript into an HTML file
2. How to output HTML from JavaScript to the Web page
3. How to print debug output to the console.
4. Branching (if) and looping (while and for) program control structures
5. Using Arrays as you would in a more traditional language such as Java or C++
6. Implementing and calling functions
7. Lambda functions
8. JQuery syntax and onload event
9. Assigning functions to handle events in the onload event using Immediately Invoked Function Expressions (IIFE)
10. Functional programming

Chapter 3 JavaScript and processing a simple form processing

This chapter will cover JavaScript with the goal of processing the simple form created in Chapter 2. Programmers from other languages will recognize most of the material in this chapter, as procedural programming in JavaScript is the similar to other languages C derived languages. Even the material on events, lambda functions, IIFE, and functional programming will have analogues to concepts that students will have already studied.

The chapter will also include an introduction to JQuery, mostly to replace functions like `document.getElementById` or to introduce the `$(document).ready()` function. The goal in this chapter is to treat JQuery as a way to short cut examples. The more important reason to introduce JQuery is it is often treated as if it is part of the base JavaScript language. When looking for JavaScript examples online the examples will use JQuery in the examples and to understand these examples some simple JQuery knowledge is useful.

The goal of this chapter is to become familiar with basic JavaScript syntax and to begin learning how to structure programs in JavaScript. Future chapters will continue to cover JavaScript objects and how to abstract applications that will be used in the mapping applications this book will present.

Chapter 3. 1 Starting a JavaScript program

The next 3 subsections will describe how to insert a JavaScript program into an HTML file, how to insert comments into JavaScript, and how to write output to the HTML page. At the end, one example of all 3 sections will be given.

Chapter 3.1. 1 Inserting JavaScript into an HTML file

JavaScript is used to create interactivity in a HTML web page. JavaScript is not part of HTML but is a scripting language that is contained in an HTML document and is interpreted by a

JavaScript engine in the browser. It is inserted into an HTML file by enclosing the JavaScript program between `<script>` and `</script>` tags[10].

The following points should be remembered when implementing JavaScript:

1. JavaScript is not HTML. If you place JavaScript source in a non-scripting portion of an HTML web page, it will likely just print out on the page. If you place HTML source in a JavaScript portion of the page, the JavaScript program will likely fail with completely unpredictable consequences. Only use JavaScript programs inside of script tags.

2. HTML is not case sensitive. The `<i>` and an `<I>` tag are interchangeable. For consistency and readability, one case should be selected and used, but there is no rule that says a body tag cannot be entered as `<bODy>`. However, JavaScript is case sensitive. The variables `Name` and `nam`e are different. Keywords are always lower case (`for, while`, etc), and using their uppercase equivalent will not work.

3. JavaScript statements should end with a ";" (semicolon). However, in most cases the language does not care if you include this semicolon or not. This text will attempt to always use the semicolon to end all statements[11], as it is considered good practice.

Chapter 3.1. 2 JavaScript Comments

Comments in JavaScript are the same as comments in C derivative languages. There are two types of comments. The first type of comment is a line comment. A line comment is signified by a //, and all of the subsequent text on that line is a comment. An example of declaring a variable and commenting is illustrated on the following line.

```
let loopCount; // Counter for string processing loop
```

Program 16 – Line comment in JavaScript

The second type of comment is a block comment. Block comments begin with a `/*` and continue until the program has a `*/`. The following is a block comments.

```
/*
```

[10] HTML was designed to handle scripting languages other than JavaScript. The <script> tag contains an attribute named language, and it is still possible to find HTML code that uses the <script language="javascript"> tag. However, no scripting language except JavaScript was ever seriously used in HTML. The <script> tag now does not need the language attribute, and it is seldom used unless some scripting language other than JavaScript is used.

[11] When teaching introductory class in JavaScript for non-majors, I often drop semicolons as they are confusing to students with no programming background. For example, an if statement that uses a semicolon, such as "`if (a == 1);`", is the source of much confusion to novice programmers, and it really does not serve a good purpose to go through the pain of explaining why the if has a null statement, and what that means.

```
    This function calculates the velocity given the speed and time
*/
```

Program 17 - Block comment in JavaScript

Often novice and even intermediate level programmers will use these two types of comments interchangeably. This can lead to problems, particularly when commenting out lines of code while debugging. For example, consider the following code block where the comment for the variable is done using a block comment.

```
function f() {
    let loopCount; /* Counter for string processing loop */
}
```

Program 18 - Block comment in a function

If something is wrong with the function, a common debugging tactic is to put a line of code in the program to show the program is entered, and then to comment out the rest of the program:

```
function f() {
    console.log("in function f");
    /*
    let loopCount; /* Counter for string processing loop */
    */
}
```

Program 19 - Why you should not use block comments in a function

This comment will not work, as /* and */ tokens are not matched. The /* signals the start of the comment, and the comment ends when a */ is encountered. The highlighted /* and */ tokens are matched above. This leave the `*/` at the end of the function outside of a comment, and this will produce an error.

A good rule for using line and block comments is to use block comments for the documentation of a function that appears before the function. Comments that occur inside of the function should use line comments. This is the commenting strategy used in this book.

Chapter 3.1. 3 Output from a JavaScript program

JavaScript program output is normally written either to the web page that contains it, the console if it is information that is only of use to a programmer, or to a dialog box. These three options are explained below.

- Output can be written it on the web page so that the user can see it. To write HTML code on the document, the `document.write()` function is used. The `document.write()` function can be used to write any HTML formatted string of information to the web page. For instance, to write the string "This is my first web page with JavaScript…" from JavaScript, the following line of JavaScript can be added to the body of the web page.

```
document.write("This is my first web page with JavaScript…");
```

Program 20 - Output to HTML DOM from JavaScript

The string passed to the `document.write()` function can be any HTML formatted string. The document`.write()` function does not simply output text on the page; document.write writes the text to the document, and the document will then correctly parse and render the output in the HTML string on the web page. To write a page heading, the following string can be written to the document:

```
document.write("<h1>First JavaScript Program</h1>")
```

Program 21 - Ouput to HTML DOM with HTML tags

Any valid html tag or statement can be written to a page, as the string which is written is effectively written to and interpreted as HTML by the browser processing the page.

- The second way to produce output is used for debugging and involves printing information which is not intended to be seen by the normal user. This output is written to the web page console. The console is used to write output that is intended to be used by programmers and others who might be supporting this site. The web console can be accessed from all major browsers, but how to access it and even some constraints on how the information is displayed are different for every browser. For example, in Chrome the browser the ctrl-shift-i key will bring up the developer tools, and from there the console can be selected. You should search the internet on how to access the Web Console for your specific browser.

  The Web Console is an invaluable place to write debug output, and other information that a programmer might want the program to produce but not let the end users see. To write to the Web Console, pass a JavaScript element (string, object, etcetera) to the `console.log()` function, as in the following line of code.

```
console.log("The program is running")
```

Program 22 - Writing to the console log

- The final way to output (and input) text from a program is to use a dialog box. Here only the output dialog box, created by using the `alert` function, is shown. Input will generally be handled in forms, so input dialogs, except for specialized dialogs such as file dialogs, are not that useful, and so are not covered. To create an output dialog box, run the `alert` function as follows.

```
alert("Something");
```

Program 23 - alert dialog box

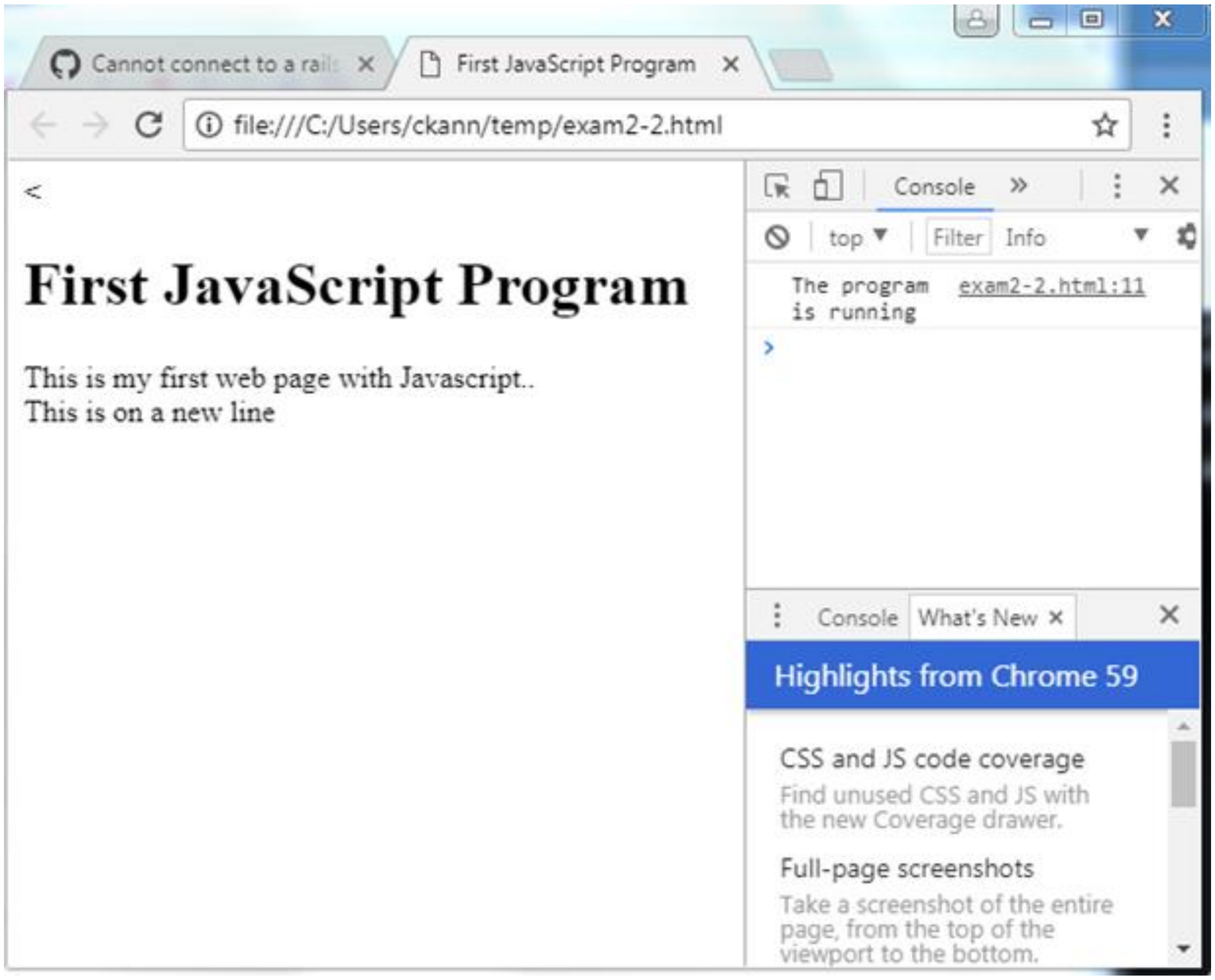
Chapter 3.1. 4 First JavaScript program

The following JavaScript program shows how to combine all of the elements described in the last 3 sections.

```
<html>
  <head>
      <title>First JavaScript Program </title>
  </head>
  <body>
    <script>
      /*
        This is an example of JavaScript
      */
      console.log("The program is running") // Write to the console

      document.write("<h1>First JavaScript Program</h1>") //Print head
      document.write("This is my first web page with JavaScript.");
      document.write("<br/>This is on a new line");

      alert("Here is an alert box");
    </script>
  </body>
</html>
```

Program 24 First JavaScript Program

The result of running this html is the following web page.



<

First JavaScript Program

This is my first web page with Javascript..
This is on a new line

```
The program    exam2-2.html:11
is running
```

Cannot connect to a rail
First JavaScript Program
file:///C:/Users/ckann/temp/exam2-2.html
Console
top
Filter
Info
Console
What's New
Highlights from Chrome 59
CSS and JS code coverage
Find unused CSS and JS with the new Coverage drawer.
Full-page screenshots
Take a screenshot of the entire page, from the top of the viewport to the bottom.

Figure 3 – Final Web Page

Chapter 3.1. 5 Quick Check

1. How is JavaScript code included in an HTML file?
2. How many `<script></script>` tags can be present in an HTML file?
3. What are the different types of comments that can be used in HTML and JavaScript?
4. What happens if you put an HTML comment in JavaScript? What happens if you put a JavaScript comment in HTML?
5. Can you mix HTML and JavaScript code? What happens if you mix them?
6. How can you use HTML tags in JavaScript code?
7. What is a dialog?
8. Give an example of when you would use `console.log, document.write`, and `alert`.

Chapter 3. 2 Primer on JavaScript

This section covers the basic procedural constructs of the JavaScript language. It covers the material that most programmers will be familiar with: variables, procedural constructs (branching and loops), and functions. While some concepts will be different from how most readers are used to thinking about them, all of the concepts here should be similar to concepts the readers already know. It is thus a primer, or elementary introduction, to JavaScript.

Chapter 3.2. 1 Variable Types

Variables in JavaScript are declared using a let[12] statement (see important footnote below). For example, the variable `loopCount` can be declared as follows:

```
let loopCount;
```

Program 25 - Declaring a variable using let in JavaScript

To someone coming from a strongly typed language such as Java, this statement is strange in that it does not declare the type of the variable, only that the variable exists. This is because JavaScript is a dynamic (or loosely) typed language. The type of the variable is the type of the last value that was assigned to it. A variable can, and very often does, change type during the execution of a program. This is illustrated in the following code fragment, where `aVar` is changed from a number to a string.

```
let aVar = 7;       // aVar is a number
aVar = “a string”; // aVar is now a string
```

Program 26 - Dynamic Typing in JavaScript

Dynamic typing is something that a programmer coming from a strongly typed language must become accustom to and requires the programmer to carefully consider their code or strange bugs can be introduced. Consider, for example, the following Java code fragment:

```
int aVar = 7;
```

[12] ECMAScript 6 has deprecated the var keyword in favor of the let keyword. Except for cases such as loop variables, they are generally equivalent. If you see a var statement when reading code on the web, you can mostly set it to a let statement when reading it. The differences are not important at this point.

```
String s = "6";
aVar = aVar + s;
```

Program 27 - Static typing failing when casting in Java

This code fragment will cause a type cast compiler error in Java, as the + sign casts the value of aVar to a string on the right-hand-side(rhs) of the equation, does concatenation to a string that is returned, and then tries to set the string to aVar, an int, which is invalid. The equivalent code fragment in JavaScript simply converts the variable aVar to a string, and performs concatenation, yielding the possible erroneous result of a string containing “76”.

```
<script>
      let aVar = 7;
      let s = "6";
      aVar = aVar + s;
      alert(aVar);
</script>
```

Program 28 - Program works in JavaScript as variable type is changed

If the `let` keyword does not define a type, why use it? The purpose of the `let` keyword is to define the scope of a variable. It can be dropped altogether, in which case the variable defaults to global scope. The scope of a variable has three possibilities in JavaScript (ES6): global, function, and local scope. The rules of scope are as follows. A variable is declared with a `let` in a block, other than a function it is a local variable. A variable declared with a `let` that is contained in a function block defaults to the function where it is scoped. A variable not declared in any block, or a variable that is not declared using the `let` keyword, has global scope.

These scoping rules are not as easy as they might appear. And since we are not yet using functions, variables in this section will have global or local scope. If the variable is declared in a block (between `{}`) it is a local variable, otherwise it is global scope. However, whether it is global or local should not have an impact on the examples in this section.

This text will recommend that the reader use the let keyword to declare all variables. Scope plays a different, and I would argue a larger, role in JavaScript than many other languages. Later in this text it will always be required, as not using it leads to endless confusion. Therefore, I strongly recommend the reader get into the habit of using the let keyword now. But any further elucidation of the word let will have to wait until later in the text.

There are 6 primitive types in JavaScript and one complex variable type called an Object. Information on the Object and Symbol types will be deferred until later in the text, but the other 5 primitive types are[13]:

- Boolean: A Boolean is a binary logical value containing either true or false.
- Number: There is only one number type in JavaScript, and it is a 64-bit IEEE 754 double precision (or floating point) number[14]. The number can also be used as an integer value,

[13] For more information on JavaScript primitive, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures.

[14] For more information on IEEE 754 double format, see https://en.wikipedia.org/wiki/Double-precision_floating-point_format.

and when used as an integer it will have 52 bits binary precision (or 15 digits precision), and allow integer values from -2251799813685248…2251799813685247

- String: A string is a textural representation of data. It consists of indexed 16-bit character values
- Null: A variable having a null value is defined and has a value, but that value is the null.
- Undefined: A variable that has not been assigned a value.

Some novice programmers have difficulties with the difference between null and undefined values. To have a null value, there must be a declared variable. It must be a variable, but the variable has no value. However undefined means the variable is not declared. Understanding this difference is common, and many web pages can be found using Google to find more information on it.

Chapter 3.2. 2 Arrays

Arrays in JavaScript are not like arrays in languages that most students are familiar with. Arrays in JavaScript are in reality map structures (normally called hashes or associative arrays) that use an index as a key. Even when used with an index, JavaScript arrays behave more like a Java ArrayList than a Java array. However, having warned the reader about the complexity of the JavaScript array, that complexity will be deferred until later in the text. For now, arrays will be treated like arrays in most of the languages the readers will be familiar with.

An array can be created in two ways. The first way to create an array is to set a variable equal to an array literal, as in the following example.

```
var weekDays = [“Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”]
```

Program 29 - Creating an array variable by assigning the variable to an array literal

As this example shows, to initialize an array in JavaScript, the variable is set to an *array literal*, or a value that is an array, and uses square brackets (`[]`) to contain the array members. This is a small difference from C derivative languages that use curly braces (`{}`) for array initialization. JavaScript will often look similar to other languages, but programmers should be careful as there are small syntax differences[15].

The second way to create an array is to call the `Array()` constructor function[16], and passing the optional size of the array.

```
var weekDays = new Array(5);
```

Program 30 - Creating an array by calling a Constructor function

[15] This syntax difference in array initialization represents more than just a small syntax difference. It is the first hint that an array and an array variable are semantically different in JavaScript than in other languages.

[16] Note that the Array() function is a JavaScript constructor function. The use of the constructor function requires the new operator and creates a JavaScript object. JavaScript objects, and the prototypes used to create them, will be covered in more detail later. For readers interested in more information, there are many web sites that cover Java objects, such as https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes.

Once again there is more than just a syntactic difference in using parenthesis (`()`) rather than square brackets (`[]`) when creating an array, and this will become apparent later when the semantic meaning of an array is looked at in more detail.

The members of the array are accessed using the square brackets, `[]`, and should be familiar to most reader.

```
weekDays[0] = “Monday”;
weekDays[1] = “Tuesday”; // etc.
```

Program 31 - Assigning array values

Arrays are zero based, as in most programming language, so the first element in the array is `array[0]`, and the second element in the array is `array[1]`, etc.

Arrays in JavaScript automatically allocate space when needed, like a Java ArrayList. So, the following code would result in a `weekDays` array consisting of 5 members, not an `ArrayIndexOutOfBounds` exception as in Java.

```
weekDays = new Array(2);
weekDays[0] = “Monday”;
weekDays[1] = “Tuesday”;
weekDays[2] = “Wednesday”;
weekDays[3] = “Thursday”;
weekDays[4] = “Friday”;
```

Program 32 - Showing how arrays automatically creates new space

Even though the true nature of arrays in JavaScript has been hinted at here, the examples used so far should at least look familiar to the reader and will be sufficient for the rest of this chapter.

Chapter 3.2. 3 Procedural constructs

There is a famous computer science theorem, the “Boehm-Jacopini Structured Programming Theorem”, which holds that all programs can be generated using only three programming constructs:

1. A sequence which is just one statement after another.
2. A branch, such as an *if* statement.
3. A loop, such as a while or for loop.

A sequence is easy to understand, as one statement follows. The other two structures will be covered in more detail in the subsequent sections of this chapter.

Once again, the purpose of this book as a primer for preparing students to work with map applications is emphasized. There are a number of different JavaScript control structures, such as do-while switch, etcetera, that are not covered. This does not mean that they are not useful; however, they do not provide value to the purpose of this text. As was stated at the beginning of this text, this is not a text on JavaScript or any other technology, but a basic overview of technologies and how they fit together.

Chapter 3.2. 4 if statement

`if` statements correlate almost exactly to `if` statements in Java/C/C++/C#/etcetera. The format of an `if` statement follows:

```
if (condition) statement;
```

Program 33 - Format of an if statement

In the if statement, the condition represents a variable or expression that reduces to a single value that is treated like a logic having values true and false.

To use an if statement, the condition is reduced to a logic value and checked if it is true or false. The statement after the condition is run if the value is true.

The statement after the condition is any valid JavaScript statement. The statement could be a single line of code, a block of code between two curly braces, or a null statement. The following code fragment shows these three possibilities.

```
if (x < 100)
    x = x = 1; // statement is a single line of code

if (x < 100){
    x = x + 1; // statement is a block of code
    y = y - 1;
}

if (x < 100); // statement is null
```

Program 34 - Format of if-else statement

Each `if` statement can contain a corresponding `else` condition that is executed when the condition in the `if` statement is false, as show below.

```
if (condition) statement; // Do if true
else statement; // do if false
```

Program 35 - Format of if-else statement

Finally, if and else statements can be combined to create if…else if…else blocks

```
if (condition) statement;
else if (condition) statement;
else statement;
```

Program 36 - Format of if - else if - else statement

All of these branch operations should be familiar to programmers coming from nearly any language. It is assumed that the reader will be familiar with these statements, and they will not be covered in any more detail.

There is one big difference between the if statement in JavaScript and some other languages like Java and C#. This difference has to do with the condition. The condition for if statement in Java/C# must be a boolean value of true/false. If any type other than a boolean is used for a condition, the compiler will produce an error. This is why the following statement produces a compiler error. The condition is set from the return value of the assignment (=) operation, which has a value of a long, not a Boolean.

```
long k = 12;
if (k = 7) alert (“true”);
```

Program 37 - if statement that fails in Java

In this case, the user probably wanted to use the comparison (==) operator. The == operator returns a Boolean value, so the statement with the == operator is valid.

```
k = 12;
if (k == 7) alert (“true”);
```

Program 38 - if statement that works in Java

There are a number of things to cover here. First, the assignment (=) and comparison (==) operators return values. It is hoped that readers are familiar with this idea that an operator is like a function and returns a value. Further it shows that the == statement always returns a logical value (or boolean) value. Finally it shows that Java/C# only allow boolean values for the condition variables.

In JavaScript, the condition variable for an `if` statement can be of any type, which is the same as in C/C++. The condition variable is not a boolean value; instead the condition value is false if it is 0 and otherwise true. It does not matter what the type of the condition variable is, it is not checked in JavaScript.

But because JavaScript allows the condition variable to be of any type, the following examples are all perfectly legitimate in JavaScript. The first 4 result in non-zero values, and would be true. The last two result in values of 0, and return false.

```
<script>
// The following are all true
if (true) alert("true");
if (7) alert ("true");
if("false") alert("true")
var1 = 9
    if (var1 = 7) alert("true")

// The following are false
if (0) ;
else alert("false")

var2 = 9
    if (va2 = 0);
    else alert("false")
</script>
```

Program 39 - Examples of true conditions in JavaScript

This becomes an issue is when using the assignment (=) operator and the comparison (==) for values. Consider the following code fragment, where the programmer incorrectly typed the assignment operator (=) instead of the comparison operator (==)

```
k = 12;
if (k = 7) alert ("true);
```

Program 40 - Incorrect result of using = for condition variable

This statement always returns true since the assignment operator (=) sets the value of `k` to `7`, and then returns a value of `7`, which is `true`. If `k` were set to `0`, this statement would always return `false`. This problem is addressed in most IDE's, but that does not remove it from the language, and it is a problem the programmer must always be aware of.

Other than the assignment (=) and comparison (==) operators, the other logical condition operation will look familiar to most programmers, and correctly return Boolean values. The condition statement supports all of the normal comparison operations, such as ==, >, <, >=, and <=. The logical `&&`, `||`, and `!` also work, and short circuit as expected.

Usefully these logical operations even work on string types, so it is possible to write the following.

```
let a = "name";
if (a == "name"); // true
if (a >= "abcd"); // false
```

Program 41 - Examples of comparison operators with strings

Chapter 3.2. 5 Sentinel control loops - while statement

There are generally three types of looping constructs: A sentinel control loop loops until some condition is met; a counter control loop, that loops a specified number of times; and an iterator, that loops over some data structure such as an array and performs an operation on each member of that data structure. These three looping constructs are often associated with `while` statements, `for` statements, and iterator loop[17].

These three types of loops will be covered in the next three sections. This section will cover the sentinel control loop, implemented with a `while` statement.

The schema[18] for a sentinel control loop is the following:

```
Process the initial input, and set the initial loop condition
Check the loop condition
      Process the data for each iteration of the loop
      Process the new input, and update the loop condition
End of loop
```

Program 42 - Schema for a Sentinel Control loop

[17] An iterator is implemented in Java as an *extended for* loop, but the syntax for these differ widely in different languages.
[18] A schema is "a representation of a plan or theory in the form of an outline or model", https://www.google.com/search?q=schema&ie=utf-8&oe=utf-8. The term schema and plan will generally be used in this text to discuss such structures, where schema is used to talk about the model what is to be done, and a plan is the implementation of a program structure. Experienced programmers have learned and internalized a large number of these schemas and tend to see programs less in terms of how the code progresses, but the schemas needed to solve a problem.

The following JavaScript program shows an implementation of a sentinel control loop. In this plan, the user is asked to input, using the JavaScript `prompt()` function, a numeric value. The program then squares the number, uses the JavaScript `alert()` function to print the result back to the user, and prompts the user for the next number. This is repeated zero or more times until the users enters the sentinel value, `-1`, at which point the loop stops.

```
<script>
    let inputValue = prompt("please enter a number, or -1 to stop")
    while(inputValue != -1) {
          alert(inputValue + " squared is " +
               (inputValue * inputValue))
          inputValue = prompt("please enter a number, or -1 to stop")
    }
</script>
```

Program 43 - Implementation of a Sentinel Control loop in JavaScript

Chapter 3.2. 6 Counter Control loops - for statement

A Counter Control loop is a loop that runs a fixed number of times based on an input variable. The schema for a counter control loop is the following:

```
Initialize the counter
Check the counter for ending value
    Process the data for this value of the counter
    Increment the counter value
End of loop
```

Program 44 - Schema for a Counter Control loop

Because three steps (initialization, check, and incrementing a counter) in the Counter Control Schema always occur, they are included in many languages using a `for` statement. The `for` statement in many languages implements these three steps as follows:

```
for (initialization; end condition check; increment) {
    Process for each element;
}
```

Program 45 - Translating a Counter Control loop into a for statement

The following JavaScript program shows the implementation of a counter control loop[19]. In this plan, the user is asked for an input (`n`). The sum of all odd numbers from `1` to `n` is calculated and output using the `alert()` function.

Note the use of the `let` keyword here to indicate the variable `i` is scoped to a local block and is thus local scoped in the for block. The variables `total` and `inputValue` are outside of any block and are global scoped.

```
<script>
      let inputValue = prompt("please enter a number")
      let total = 0
```

[19] This schema is really a more complicated schema to calculate a sum of numbers, however it contains a counting loop schema.

```
      for (let i = 1; i <= inputValue; i = i + 2 ) {
         total = total + i;
      }
      alert("The sum of odd numbers from 1 to " +
           inputValue + " is " + total);
</script>
```

Program 46 - Implementation of a Counter Control loop

Chapter 3.2. 7 Iterator loops – for/in and for/of for foreach statements

An iterator is a looping structure that iterates over all of the members of an array. To understand an iterator, consider the weekDays array defined earlier.

```
let weekDays = [“Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”]
```

The members of this array could be printed out using a standard `for` loop as follows:

```
<script>
   let weekDays = ["Monday", "Tuesday", "Wednesday", "Thursday",
                   "Friday"]
   for (let i = 0; i < weekDays.length; i++) {
        alert(weekDays[i])
   }
</script>
```

Program 47 - Printing an array using a Counter Control loop

The for/in iterator allows the weekDays array to be processed in a simpler syntax. The for/in equivalent to the Program 47 is shown in Program 48. The value x is set to the index of the first member in the array, and the iterator sets x to the next array index until all members of the array are processed. Note that the implementation of a for/in loop seems strange but will become apparent later when the true nature of arrays in JavaScript is covered.

```
<script>
   let weekDays = ["Monday", "Tuesday", "Wednesday", "Thursday",
                   "Friday"]
   for (let x in weekDays) {
        alert(weekDays[x])
   }
</script>
```

Program 48 – Printing an arraying using a for/in iterator

JavaScript also has a for/of statement, which is an iterator that iterates over the values in the array rather than the array indices. The following is an example of the for/of iterator:

```
<script>
   let weekDays = ["Monday", "Tuesday", "Wednesday", "Thursday",
                   "Friday"]
   for (let day of weekDays) {
       alert(day);
   }
</script>
```

Program 49 - Printing an arraying using a for/of iterator

Chapter 3.2. 8 Functional iterator – forEach

There is one other type of iterator, the forEach iterator. This iterator is executed by providing a function to be called for each array member and illustrates how functional programming works. The forEach iterator will be covered at the end of the chapter after functions have been covered and will be used to introduce the reader to Lambda Functions in Functional Programming.

Chapter 3.2. 9 Functions

Functions in JavaScript are similar to methods in Java or C#, or functions in C. They are used to abstract behavior. Functions are declared using the function statement, followed by the name of the function, an open parenthesis, arguments to the function, a close parenthesis, and a block which contains the body of the function. For example, a function which takes two input value distance and time, and returns the speed would look as follows:

```
<script>
   function speed(distance, time) {
       return distance / time;
   }
</script>
```

Program 50 - Implementatiton of the speed function

Functions represent metadata as they provide some abstracted behavior to be used by data from the web page, and so the function definitions are generally found in the head of a document. Calling functions is part of the information on the web page and occurs in the body of the document. The following is an example of a web page that prompts the user for values of distance and time and prints out the speed. The program exits when a value of "-1" is entered for the distance.

```
<html>
  <head>
    <title>Speed web page.</title>
    <script>
      function speed(distance, time) {
        return distance / time;
      }
    </script>
  </head>

  <body>
    <script>
      let d = prompt("Enter the distance, -1 to end")
      while (d != -1) {
        let t = prompt("Enter the time to travel the distance")
        alert(d / t);
        d = prompt("Enter the distance, or -1 to end")
      }
    </script>
  </body>
</html>
```

Program 51 - Sentinel Control loop program using a function to calculate speed.

Chapter 3.2. 10 Quick Check

1. What is the purpose of the `let` keyword in JavaScript?
2. How is the type of a variable determined?
3. What datatypes exist in JavaScript?
4. What is an undefined variable? How does it differ from a null variable?
5. What types of scoping are available in JavaScript? What is the default scope of a variable?
6. What 3 types of control structures are needed to create programs? Give an example of each in a language you are familiar with.
7. What three looping types exist in most programming languages? Give an example of each type in a language you are familiar with.
8. True or false: All operators (including the assignment (=) operator) return values.
9. What is short-circuiting of a logical operator? Show how this can be used to protect a program from a zero divide.
10. Do operators return values?
11. What is the result of operators returning values for condition variables for if, while, or for loops.
12. The text said an if condition can have a null statement. What is a null statement? Why is it a bane to novice programmers? Why does JavaScript allow null statements?
13. Can a for or while condition have a null statement?
14. Does using the comparison operator (==) work for strings in Java/C#/C++/Python?
15. Functions are said to have "positional parameters" in JavaScript. What is the difference between a positional and a keyword parameter?
16. Do functions always return a value? What do you think is returned if nothing is specified?
17. In your own words, explain what is meant that "functions are data" in JavaScript. This is used to defined Lambda functions. If you are familiar with Lambda functions in Java, are Lambda functions in Java really data? If not, what are they?
18. What happens if you write the following code: specifically, does it work, and if it does, what is the length of the people array? What is the value of `people [0]`? What is the value of `people [5]`?

```
let people = Array(2);
people(1) = "Nick"
people(2) = "Ann"
people(7) = "Carol"
```

Chapter 3. 3 Events, Onload Event and JQuery

One important paradigm that is implemented in JavaScript is Event Based Programming (EBP). Events are asynchronous actions that occur (or are *raised*) while the program is running. These actions (or events) are often generated by the user, such as clicking a button or changing the value in a text field. But events can also occur from asynchronous action of the program, such as loading the web page or the completion of reading of a file. The events which are raised are then associated with a callback function that handles the event.

Chapter 3.3. 1 Associating a call back with an event

Associating a call back with an event is illustrated in the following program.

```
<html>
  <head>
    <title>Error using event </title>
    <script>
      function pressFunction() {
        alert("You pressed me");
            //document.write("You pressed me");
      };
    </script>
  </head>

  <body>
    <input type="button" id="myButton" value="Press me"
           onClick="pressFunction();"/>

  </body>
</html>
```

Program 52 - Using a form element in the head before it is defined.

In this example, the click event (called onClick) is associated with the function pressFunction, which is metadata and defined in the head of the html page. When a user *clicks* on the button that says, "Press me", it calls the function associated with the click even, the pressFunction, and an alert box is shown saying "You pressed me".

This is the complete life cycle of events and call backs. An event (click) is mapped to a callback function (pressFunction). When the event is raised (the user clicks the button), the event call back function is called.

The rest of this section will explain how you should implement these in JavaScript.

Chapter 3.3. 2 Handling an Event – Unobtrusive JavaScript

Because event (or callback) functions are run in response to events from the program, they are metadata. This implies that when writing an HTML page, it is normal for a programmer to define functions to handle events coming from form elements in the head of the html document. A standard know as Unobtrusive JavaScript says that all JavaScript should be maintained in the head of the document, so setting the onClick as part of the button definition is considered bad practice.

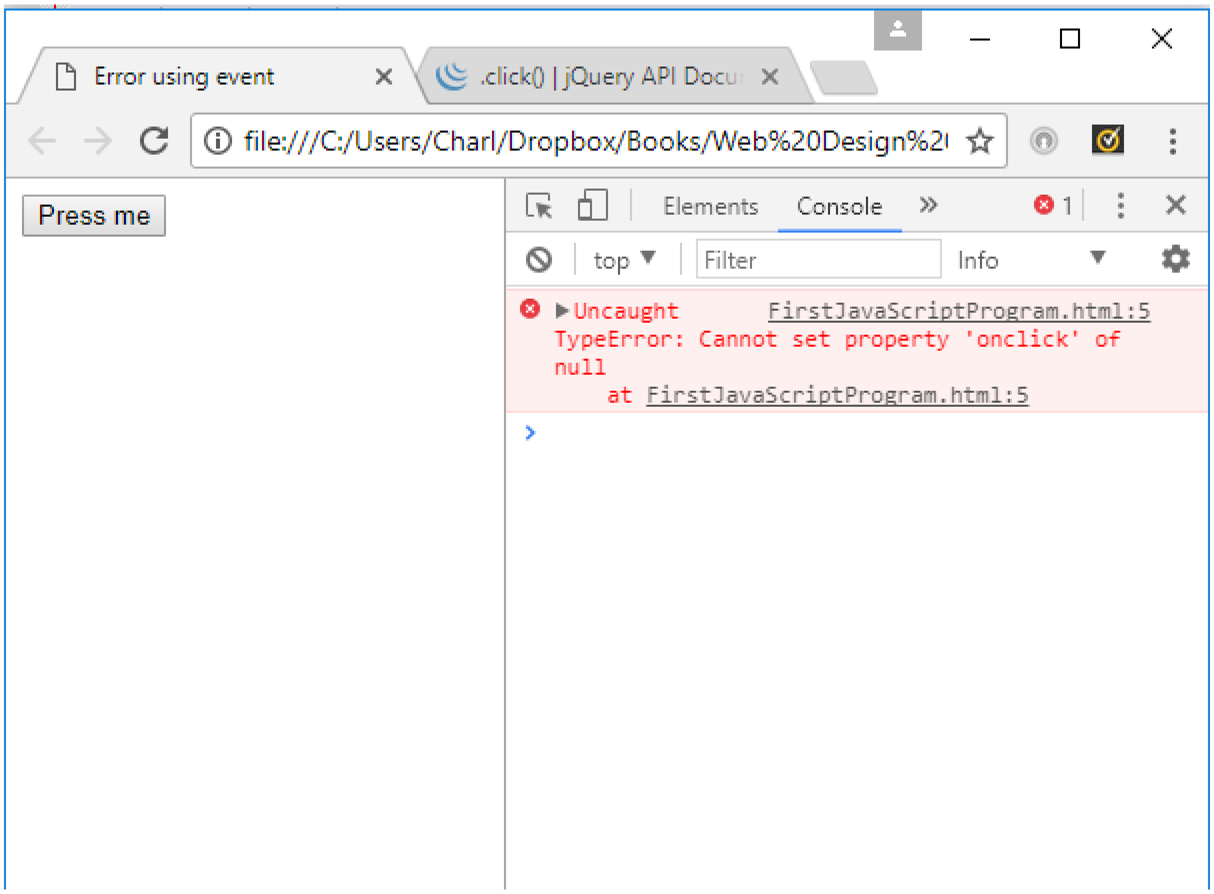
The correct way to assign a callback function is as follows. First it is important to know that all components on a web page are contained in a Document Object Model, or DOM. The DOM is very important when writing JavaScript for a browser, as nearly everything is in the DOM. However, for now you should know that buttons, text fields, checkboxes, etcetera, are all stored as objects in the DOM.

Unobtrusive JavaScript says that the call back should be assigned by retrieving the button, in this case the button named "myButton" from the DOM using the getElementById method. The click event for this button is then assigned to an anonymous function which is then called when the button is pressed. This is shown in Program 53 below.

```
<html>
  <head>
    <title>Error using event </title>
    <script>
      document.getElementById("myButton").click =  function() {
        alert("You pressed me");
      };
    </script>
  </head>

  <body>
    <input type="button" id="myButton" value="Press me" />
  </body>
</html>
```

Program 53 - Using a form element in the head before it is defined.

However, there is a problem with this program. The form elements, such as a button, are defined in the body of the page, and that is **after** the attempt to use it to set its callback occurs in the head. As the following JavaScript program shows, this results in an error when trying to reference the button before it is defined. Note this error occurs in the browser console, which is not directly viewable to the user. If you do not know how to use the browser console, you should Google how to access it for the browser you are using (accessing the console is different for every browser, but is available in every browser).


Error using event
.click() | jQuery API Docu
file:///C:/Users/Charl/Dropbox/Books/Web%20Design%2
Press me
Elements
Console
top
Filter
Info

```
Uncaught            FirstJavaScriptProgram.html:5
TypeError: Cannot set property 'onclick' of
null
    at FirstJavaScriptProgram.html:5
```

Figure 4 - Error message processing a form element before it is defined

How to handle this problem will be the subject of the next section.

Chapter 3.3. 3 Handling an Event – onload event

To handle the problem of needing to access the button in the head, an event is defined that is raised after the form is loaded, and all form elements are defined. This event is called an `onLoad` event. The `onLoad` event occurs after the form is loaded, and thus after all the form elements have been processed and are available to be referenced. A callback function attached to the `onLoad` event can safely reference form elements in the body, knowing that the form is completely loaded, and the body of the HTML file is processed. This is shown in the following example.

```
<html>
  <head>
      <title>Using onLoad event </title>
    <script>
      function myOnLoad() {
         document.getElementById("myButton").onclick = function() {
           alert("You pressed me");
         }
      }
      window. Onload = myOnLoad;
    </script>
  </head>

  <body>
    <input type="button" id="myButton" value="Press me" />
  </body>
</html>
```

Program 54 - Setting a callback function in an onLoad event

This small piece of code is complex, so it will be covered in some detail here. First a function, `myOnLoad` is defined, and it is in this function that the `myButton's` click event is associated with the function to alert the user. The function `myOnLoad` is not called, so it is defined but not yet executed.

The next line of code, `window.onload = myOnLoad,` associates the `myOnLoad` method with the onLoad event. This tells the HTML DOM to run the `myOnLoad` function when the `onLoad` event is raised, after the page is loaded. This means that the button myButton has been defined and can be associated with the listener to alert a message when the button is pressed. This part is fairly simple, at least in a big picture sense, to understand.

What is much more complex is that the function `myOnLoad` is not called in the statement `window.onload = myOnLoad,` but the function is assigned to the variable `window.onload and` called later when the `onLoad` event is raised. To see this, compare the previous program with the next program, which includes the parenthesis in the assignment of the window.onload variable. This is a common mistake by novice programmers, as most novices have been taught that functions are executable statements that are run, and not data that can be passed around using variables.

```
<html>
  <head>
    <title>Window onload event</title>
    <script>
      function myOnload() {
        /* This code causes the JavaScript to fail if
           uncommented. But it would not work anyway as the
           button code has not been loaded…
          document.getElementById("myButton").onclick = function() {
            alert("You pressed me");
          }
          */
          alert("window is not loaded");
          return "you dummy";
      }
      alert(window.onload = myOnload());
    </script>
  </head>

  <body>
  </body>
</html>
```

Program 55 - Program showing function being called rather than being set to a variable.

The result of this second program is that the alert is run once when the head of the HTML file is processed, and the `window.onload` variable is set to the string “you dummy”, which is what is returned from the `myOnLoad` function. Once again, you cannot associate the button with a function as the button would not exist. The code to associate a function with an event must be run in the onLoad event, not when the head of the HTML file is processed.

Before continuing, the reader should note that if a function is referenced without parenthesis (e.g. myOnLoad) the function is treated as data. If a function is referenced with parenthesis (e.g. myOnLoad()), the function is executed. From a syntactic point of view, this is probably the issue that causes the most problems to novice JavaScript programmers.

This example shows an important feature of JavaScript. Functions in JavaScript are treated as *first class object and* can be used like any other data type. Functions that can be used as data are called a Lambda functions. Lambda functions are the basis for a programming paradigm called Functional Programming, which is a very different paradigm from Procedural or Object-Oriented Programming. Even in languages that purport to have included lambda functions, like Java, are actually providing syntactic sugar for other language constructs, and languages like Java do not provide a real basis for Functional Programming.

Note that since functions can be data, they can be passed as data values. This, in this program, the programmer has realized that the myOnload() function is likely never used in the program except in response to an onLoad event. There is really no need to clutter the JavaScript namespace with the name of the function, and the function is defined anonymously. This is the normal way the event callback is defined using an anonymous lambda function.

```
<html>
  <head>
    <title>Using onLoad event </title>
    <script>

      window.onload = function() {
        document.getElementById("myButton").onclick = function() {
          alert("You pressed me");
        };
      }
    </script>
  </head>

  <body>
    <input type="button" id="myButton" value="Press me" />
  </body>
</html>
```

Program 56 - Setting an event callback function using an anonymous function.

Here the variable is set in the same statement as the function defined.

Note that the parentheses here do not mean the function is being executed but are part of the function definition. Once again this is confusing to novices learning JavaScript. To execute this function, the statement windows.onload=function(){…}() would have to be used. This will form the basis for another JavaScript paradigm to be introduced later, the Immediate-Invoked-Function-Expression (IIFE).

Chapter 3.3. 4 JQuery ready function

JQuery is a library that provides many functions which are useful when doing JavaScript programming. This text will use JQuery extensively. This is because JQuery provides so many useful features. Also, many of the examples of JavaScript programming found on the internet use it, and it is hard to read JavaScript examples on the web without knowing the basic concepts of JQuery.

To use JQuery, the following line should be included in the head of your HTML document before any JQuery function is used. This line includes the JQuery library and gives the HTML document access to all JQuery utility. It must be included before any other libraries that might use JQuery.

```
<script src=""https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js""></sc
ript>
```

Once installed, JQuery statements are formatted as `$(`*`identifier`*`)`. In this statement, the `$` is short for the `jQuery` function, and so is just shorthand for saying `jQuery(`*`identifier`*`)`.

The identifier can be one of many different types of HTML elements, such as a CSS selector, JavaScript elements, element arrays, objects, selections, or other types of elements. JQuery will look at the element that is being requested and call a function that will perform the correct operation for that type[20].

In this section, only the JQuery options needed to assign the button `onClick` event will be covered. This will be done in two steps. The first will be how to handle the form loading in JQuery. The second will be how to set the `onClick` event for a button.

The JQuery function that is run after the HTML file has completed processing and all form elements are defined is the `$(document).ready()` function. The `$(document).ready()` function is *similar* to setting a function in the `window.onload` event, and for the purpose here will be used interchangeably.

The following code can be used to print a message to an alert box after the form elements have been processed. Note here that an event is not set to a function to execute, but the `$(document).ready()` function is executed when the HTML file has loaded. This is just a different way to handle the semantics of events. JavaScript will set a function to an event variable, and JQuery will call a method when the event is raised. Do not let this confuse you, as the effect is the same as setting the event variable. The function is run when the event is raised.

```
<html>
  <head>
    <title>Window onload event</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js">
  </script>
  <script>
```

[20] For more information on how JQuery works and the types that are accepted to the jQuery function, see http://api.jquery.com/jquery/.

```
         $(document).ready(function() {
            alert("window is loaded");
         });
  </script>
  </head>

  <body>
  </body>
</html>
```

Program 57 - Using the JQuery $(document).ready() function

As of JQuery 3.0, the `$(document).ready()` syntax has been deprecated, and the current recommended syntax for calling the ready function is to put the function to be run when the page is loaded as a parameter to the jQuery function. Remember that $ is the jQuery function, so

$(function(){…})

is the same as saying

jQuery(function(){…})

The jQuery function will recognize the parameter is a function and set it to respond to the ready event. This is only a syntax change to make the code shorter and maybe easier to understand. Just remember if a function is passed to the `jQuery` function, it is set to the JQuery ready event.

```
<html>
  <head>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js
">
  </script>
    <title>Window onload event</title>
    <script>
         $(function() {
            alert("window is loaded");
         })
    </script>
  </head>

  <body>
  </body>
</html>
```

Program 58 - JQuery 3.0 ready function

Chapter 3.3. 5 Using JQuery to access an DOM variable

One of the big advantages to JQuery is that it has a library to provide shortcuts to accessing JavaScript variables. JQuery shortcuts allow the retrieving any DOM elements such as buttons, textboxes, etc., using a simplified format. The JavaScript function to retrieve a from element by its id, `document.getElementById("id")`, can be written more simply as `$("#id")` in JQuery, as is shown in the program below. From now on, JQuery syntax will be used to retrieve DOM elements.

```
<html>
  <head>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js">
  </script>
    <title>Window onload event</title>
    <script>
         $(function() {
            $("#myButton").click(function(){
                  alert("you pressed me")
              })
         })
    </script>
  </head>

  <body>
    <input type="button" id="myButton" value="Press me" />
  </body>

</html>
```

Program 59 - Using JQuery to access a DOM variable

In this example, the `id` of the button is prefaced by the "`#`" (hash) sign. JQuery can be used to access various types of identifiers (called selectors in CSS), and it uses the CSS conventions to specify the type of selector it is using. For example, to use a class selector, the "." (dot) will be prepended to the id. The "`#`" (hash) means the name is to be used to access the variable. The types of variables that can be accessed correspond to their CSS definitions, and will be seen in more detail in the chapter on CSS.

Also note that the name of the JQuery function to handle an onClick event is the click function and setting the event variable has been changed to a function call. Both renaming the event and changing the event to a function call are conventions followed in JQuery.

Chapter 3.3. 6 Quick Check

1. What is an event?
2. What does it mean to say an "event is asynchronous"?
3. How is EBP different from procedural programming, where all programs being in the main and proceed one statement after another.
4. What is a Lambda Function?
5. What is an onLoad event. How does assigning a function to an onLoad event differ from using the $(document).ready() JQuery function?
6. What is the shorthand method to write the $(document).ready() JQuery function?
7. How would you access an DOM variable named "myVariable" using JQuery?

Chapter 3. 4 Processing Form elements using JQuery and Unobtrusive JavaScript

This section will introduce the reader to how to implement and call functions for process an HTML form in JavaScript. It will use EBP and callbacks, and two JavaScript paradigms, unobtrusive JavaScript and Immediate Invoked Function Expressions (IIFE), currently considered best practice, will be introduced in this chapter.

This section will use EBP to process the data on the HTML form implemented in the shown below. Each topic in this sub chapter will modify the head for the HTML file to show how to process the form element that is introduced in that section. The reader can copy the HTML form and simply replace the head as each new topic is covered to see how they work.

One of the main uses of JavaScript is to provide the interactivity to a web page. This interactivity will be used to show how to process the data on a form and print it back to the user in an alert box. This will be presented in the following manner:

1. A callback function will be added to the button so that when it is clicked the function will be called. The handling of the button will then be processed in the callback function. This section will introduce Unobtrusive JavaScript, and how to use the principals of Unobtrusive JavaScript to set a callback function.
2. An illustration of how to handle each of the different form element types will be shown. These form elements are:
   2.1. A textbox
   2.2. A checkbox
   2.3. A radio button group

The form presented here will be processed, and each of its elements printed out. In the subsequent sections, the entire head of the document will be shown in full as it is developed. To see how each option works, take the head from the subsection and insert it into this program. The final result will be a JavaScript program to process the data on the form and print it to the console.

```
<html>
  <head>
    <title>Map Example Input Screen</title>
    <script>
      // New code will go here
    </script>
  </head>

  <body>
    <h1>Map Example Input Screen</h1>
      <p>
        <label id="l1" for="title">Title</label>
        <input type="text" id="title" size="20">
      </p>

      <p>
        Map Options<br>
        <label id="l2" for="resize">Allow map to be resized:
        </label>
        <input type="checkbox" id="resize"/>
        <br/>

        <label id="l3" for="recenter">
          Allow map to be re-centered:
```

```
        </label>
        <input type="checkbox" id="recenter" checked />
      </p>

      <p>
        Type of Map<br>
        <input type="radio" name="maptype" id="XYZMap"
           value="XYZ Map"/>
        <label id="label1" for="XYZMap">XYZ map </label>
        <br/>

        <input type="radio" name="maptype" id="StamenMap"
          value="StamemMap" checked />
        <label id="label2" for="StamenMap">Stamen Map </label>
      </p>

      <p>
        Screen Size<br>
        <input type="radio" name="screenSize" checked
          id="600x480" value="600x480"/>
        <label id="label3" for="XYZMap">600x480 </label>
        <br/>

        <input type="radio" name="screenSize" id="1024x768"
          value="1024x768"/>
        <label id="label4" for="XYZMap">1024x768 </label>
        <br/>

        <input type="radio" name="screenSize" id="1280x800"
          value="1280x800"/>
        <label id="label5" for="XYZMap">1280x800 </label>
      </p>

      <p>
        Center of Map<br>
        <label id="label1" for="lat">Latitude </label>
        <input type="number" id="lat"/>
        <label id="label2" for="long">Latitude </label>
        <input type="number" id="long"/>
      </p>

      <input type="button" value="Process Form" id="processButton" />
  </body>
</html>
```

Program 60 - HTML form to be processed in Chapter 3.4

Chapter 3.4. 1 Including JQuery

To use the JQuery library, it must be included. This is the first modification to the head of the program made. The head of the program is now:

```
<head>
  <title>Map Example Input Screen</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.Java
Script">
   </script>
   <script>
      // New code will go here…
   </script>
</head>
```

Program 61 – Inserting JQuery into the form

Chapter 3.4. 2 Adding an Event Callback to a Button

To begin processing the form, the *`Process Form`* button at the bottom of the form will be given functionality by setting the callback on the onClick event.

In a button, the `onClick` event is raised when the button is pressed. This event behavior can be specified when the button is created, as in the following example that calls an alert when the button is pressed.

```
<input type="button" value="Process Form"
       onClick="alert('Button is clicked')" />
```

In this code, the JavaScript code to execute (e.g. creating an alert box) is assigned in a string to the onClick event. In the web page we are developing, putting all of the JavaScript code directly in the html button would be difficult and probably unreadable. The `onClick` can execute any JavaScript expression, so it is easy to call a function and put the extended behavior in the function and call that function when the button is pressed.

```
<input type="button" value="Process Form"
       onClick="processForm()" />
```

This method of providing a callback for a function is valid and works but represents an idiom that is out of favor. A style known as Unobtrusive JavaScript[21] is currently considered best practice for developing web pages. One of the principals of Unobtrusive JavaScript is that HTML markup and JavaScript should not be mixed. This was done in the previous example, where a JavaScript statement was placed in the HTML input tag for the `onClick` event.

To separate HTML markup from JavaScript, the JQuery `$(document).ready` function is used to associate the form element (the button) with a JavaScript function. To do this, first the button is given an id so that it can be retrieved from the DOM. This was already done in the HTML form above.

```
<input type="button" value="Process Form" id="processButton" />
```

In the ready function, the DOM element for the button is retrieved, and an anonymous function is attached to its onClick event. This is illustrated in the following example.

```
<head>
  <title>Map Example Input Screen</title>
```

[21] Unobtrusive JavaScript is a set of best practices, one of which not mixing HTML markup and JavaScript. For more information about Unobtrusive JavaScript, see: https://en.wikipedia.org/wiki/Unobtrusive_JavaScript.

```
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.Java
Script">
  </script>
  <script>
    // JQuery $(document.ready()) function
     $(function() {
       document.getElementById("processButton").onclick =
         (function(){
           alert("button is pressed");
         });
       });
  </script>
</head>
```

Program 62 - Head of HTML file to associate a button with a function

Chapter 3.4. 3 Processing a textbox

In the previous example, an alert box was used to test if the program was *wired* correctly, e.g. that the `processButton` function was called when the button was pressed. It is important in any language to make sure parts of the program work before trying to build the entire program. Novice programmers will often attempt to write entire programs and end up with a 20-line program with 100 compiler errors, several logic errors, and no chance to make a working program. The first thing to do when writing a program is to break the program down into simple pieces and make each piece work while building those pieces into the larger program.

This is true in languages like Java, C#, C/C++, etc., but is even more true in a language like JavaScript. In JavaScript, a single bad line of code can cause a program of 100's of lines of working JavaScript to simply produce no output. These programs will often produce confusing errors, or no errors at all. In JavaScript, it is always best to build a program in stages, making sure each change has the desired effect.

Now that the wiring for the button is working, the processing of the form can be implemented. For this program, processing means the element will write a message to the console.log. First, all of the text boxes will be processed. Note that the term *textbox* as used here includes input types other than text. Input types like number and calendar are also treated like text boxes.

To process a textbox, its *value* attribute is retrieved. The value attribute contains the message in the textbox as a string. Note that JQuery is a little strange in that when it retrieves a form element, it retrieves it as an array, and so the first element in the array needs to be dereferenced to get the value. This is shown in the code below to get the value from the title textbox.

```
<script>
   $(function() {
     $("#processButton").click(function(){
       console.log("The title is " + $("#title")[0].value);
     });
   });
</script>
```

Program 63 - Retrieving the text from a textbox using the JQuery array format.

An alternative to using the array notation is to use the JQuery prop() (property) function, which was introduced in JQuery 1.6. Both of these work, but the prop() function is the emerging standard, and so will be used in this textbook. Processing all of the textboxes using prop function is shown below.

```
<head>
  <title>Map Example Input Screen</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.Java
Script">
  </script>
  <script>
    $(function() {
      $("#processButton").click(function(){
        alert("Be sure to check the console");
        // Text Boxes
        console.log("The title is " + $("#title").prop('value'));
        console.log("The center of the map is latitude " +
          $("#lat").prop('value') + " and longitude " +
          $("#long").prop('value'));
      });
    });
  </script>
</head>
```

Program 64 - Processing textboxes with JQuery

Chapter 3.4. 4 Processing a checkbox

The next step to process the form is to process the checkboxes. Just like the textbox had a *value* property that contained the text that was entered, the checkbox has a *checked* property of that indicates whether or not it is checked. If the checkbox is checked, the property is true, else it is false. The checkbox also has a value property, but that is not used in this example.

The processing of a checkbox is analogous to the processing of a textbox above. The checkbox object is retrieved from the DOM, and then the checked property, rather than the value property, is retrieved. This is shown in the code below:

```
console.log("Allow the map to be resized? " +
  $("#resize").prop('checked'));
console.log("Allow the map to be recentered? " +
  $("#recenter").prop('checked'));
```

The processing of the checkboxes is added to the head section of the file above, and the new HTML head is shown below.

```
<head>
  <title>Map Example Input Screen</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.Java
Script">
  </script>
  <script>
    $(function() {
      $("#processButton").click(function(){
        alert("Be sure to check the console");

        // Text Boxes
        console.log("The title is " + $("#title").prop('value'));
        console.log("The center of the map is latitude " +
          $("#lat").prop('value') + " and longitude " +
          $("#long").prop('value'));

        // Check Boxes
        console.log("Allow the map to be resized? " +
          $("#resize").prop('checked'));
        console.log("Allow the map to be recentered? " +
          $("#recenter").prop('checked'));
      });
    });
  </script>
</head>
```

Program 65 - Processing checkboxes with JQuery

Chapter 3.4. 5 Processing radio buttons

Processing radio buttons is more complicated that textboxes and checkboxes. For check boxes, each box was processed separately, and these checkboxes only required retrieving a property (value or checked) from the object. Radio buttons are contained in a group, and so to get the value that is checked requires that the radio button group be processed to see which one radio button has been checked.

To process radio buttons, the first step is retrieved all the buttons in the group by the *name* attribute, not *id* attribute. Unlike the id, which is unique, the name is not. And the name was used to group the radio buttons in the User Interface (UI). To process the radio button, all of the buttons having the same name are retrieved using the getElementsByName[22] method as an array of individual radio buttons.

The most straight forward way to process the radio buttons is to walk the array to see which array element is checked, and then retrieve the properties (id, value, etcetera) associated with that array element. This is shown below in the function below, where the `document.getElementsByName` function is used to get an array of radio buttons, and that array is then processed to find the checked item. If the item is checked (the checked attribute is true), the value attribute is printed.

Note that for the radio button, the value attribute is used in this example. The value is set when the specific radio button was defined in the HTML in order to get a string associated with the radio button actually selected.

```
<head>
  <title>Map Example Input Screen</title>
```

[22] Note that names of the methods. To retrieve a single object such as a checkbox or textbox, the method getElementById is called. Note that the string Element in this name is singular. To retrieve the radio button group, the getElementsByName is called. Not the string Elements is plural.

```
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.Java
Script">
  </script>
  <script>
    $(function() {
      $("#processButton").click(function(){
        alert("Be sure to check the console");

        // Text Boxes
        console.log("The title is " + $("#title").prop('value'));
        console.log("The center of the map is latitude " +
          $("#lat").prop('value') + " and longitude " +
          $("#long").prop('value'));

        // Check Boxes
        console.log("Allow the map to be resized? " +
          $("#resize").prop('checked'));
        console.log("Allow the map to be recentered? " +
          $("#recenter").prop('checked'));

        // Radio Buttons
        maptype = document.getElementsByName("maptype")
        for (let i = 0; i < maptype.length; i++){
          if (maptype[i].checked) {
            console.log("The maptype is " + maptype[i].value);
          }
        }
        });
    });
  </script>
</head>
```

Program 66 - For Loop to process radio buttons

JQuery has abstracted this loop into a single line that can be used to process the radio button.

```
$('input:radio[name=maptype]:checked').prop('value'));
```

This statement says that the elements that have the name *maptype* are an input radio button group, which JQuery is to loop through return the checked radio button. The value property of the selected radio button is then retrieved. The JQuery equivalent to the previous head code would be the following.

```
<head>
  <title>Map Example Input Screen</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.Java
Script">
  </script>
  <script>
    $(function() {
      $("#processButton").click(function(){
        alert("Be sure to check the console");

        // Text Boxes
        console.log("The title is " + $("#title").prop('value'));
        console.log("The center of the map is latitude " +
          $("#lat").prop('value') + " and longitude " +
          $("#long").prop('value'));

        // Check Boxes
        console.log("Allow the map to be resized? " +
          $("#resize").prop('checked'));
        console.log("Allow the map to be recentered? " +
          $("#recenter").prop('checked'));

        // Radio Buttons
        console.log(
          $('input:radio[name=maptype]:checked').prop('value'));
        console.log(
          $('input:radio[name=screenSize]:checked').prop('value'));
      });
    });
  </script>
</head>
```

Chapter 3.4. 6 The final web page to process a form

The following is the final result of the web page to create and process the form. If the reader has correctly implemented all the changes above, they should have this solution, but it is included here in case readers have issues.

```
<html>
<head>
  <title>Map Example Input Screen</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.Java
Script">
  </script>
  <script>
    $(function() {
      $("#processButton").click(function(){
        alert("Be sure to check the console");

        // Text Boxes
        console.log("The title is " + $("#title").prop('value'));
        console.log("The center of the map is latitude " +
          $("#lat").prop('value') + " and longitude " +
          $("#long").prop('value'));

        // Check Boxes
        console.log("Allow the map to be resized? " +
          $("#resize").prop('checked'));
        console.log("Allow the map to be recentered? " +
          $("#recenter").prop('checked'));

        // Radio Buttons
        console.log(
          $('input:radio[name=maptype]:checked').prop('value'));
        console.log(
          $('input:radio[name=screenSize]:checked').prop('value'));
      });
    });
  </script>
</head>


  <body>
    <h1>Map Example Input Screen</h1>
      <p>
        <label id="l1" for="title">Title</label>
        <input type="text" id="title" size="20">
      </p>

      <p>
        Map Options<br>
        <label id="l2" for="resize">Allow map to be resized:
        </label>
        <input type="checkbox" id="resize"/>
        <br/>

        <label id="l3" for="recenter">
          Allow map to be recentered:
        </label>
        <input type="checkbox" id="recenter" checked />
      </p>

      <p>
        Type of Map<br>
        <input type="radio" name="maptype" id="XYZMap"
           value="XYZ Map"/>
        <label id="label1" for="XYZMap">XYZ map </label>
        <br/>

        <input type="radio" name="maptype" id="StamemMap"
          value="StamemMap" checked />
        <label id="label2" for="StamenMap">Stamen Map </label>
      </p>

      <p>
        Screen Size<br>
        <input type="radio" name="screenSize" checked
          id="600x480" value="600x480"/>
        <label id="label3" for="XYZMap">600x480 </label>
        <br/>

        <input type="radio" name="screenSize" id="1024x768"
          value="1024x768"/>
        <label id="label4" for="XYZMap">1024x768 </label>
        <br/>

        <input type="radio" name="screenSize" id="1280x800"
          value="1280x800"/>
        <label id="label5" for="XYZMap">1280x800 </label>
      </p>

      <p>
        Center of Map<br>
        <label id="label1" for="lat">Latitude </label>
        <input type="number" id="lat"/>
        <label id="label2" for="long">Latitude </label>
        <input type="number" id="long"/>
      </p>

      <input type="button" value="Process Form" id="processButton" />
  </body>
</html>
```

Program 67 - Final program to process a form

Chapter 3.4. 1 Quick Check

1. What is a callback function? How is it used to process an event?
2. Give two ways to associate a callback with a button? Which is preferred? Why?
3. Are *id*s unique on a web page? Are *name*s unique on a web page? How would you process each of these?
4. Does callback code have to run a function?
5. What is the difference between a Lambda and Anonymous function? Which (if either or both) are implemented in JavaScript?
6. What is the Web Console? How do you access it? Why would you use it?
7. What determines the radio buttons that make up a group?
8. Retrieve and display using alert boxes, with and without JQuery, the input from a textbox, checkbox, and radiobutton,

Chapter 3. 5 Functional Programming in JavaScript

JavaScript is really a multi-paradigm language. Having the ability to handle more than one paradigm can be a blessing and a curse, as having access to multiple ways to program gives a programmer a lot of power in JavaScript, but it also makes the language hard to use and easy to misuse. As we have seen in this chapter, JavaScript supports Imperative or Procedural Programming. JavaScript supports Object Oriented Programming (OOP) through the use of prototypes, though it is a very different type of OOP than most readers are familiar with in Java/C#/C++ etc. JavaScript supports Event Driven Programming[23].

JavaScript also supports some elements of Functional Programming, in particular lambda functions. The forEach iterator is introduced here to introduce lambda functions, anonymous functions, and the basic concepts of Functional Programming. Note that entire books are written on Functional Programming, and so this meant to give a flavor of what Functional Programming is, and how it is used in JavaScript. The Functional Programming introduction given here is enough to be able to read and understand JavaScript examples, but is not enough to explain how to use Functional Programming.

To understand Functional Programming, the following program a forEach function to iterate over the weekDays array is presented.

[23] For more information, see https://softwareengineering.stackexchange.com/questions/127672/is-javascript-a-functional-programming-language.

```
<script>
    function printDay(day, index) {
           alert("Weekday " + (index+1) + " is " + day)
    }
    let weekDays = ["Monday", "Tuesday", "Wednesday", "Thursday",
                    "Friday"]
    weekDays.forEach(printDay)
</script>
```

Program 68 - Final program to process a form

This program illustrates one aspect of functional programming: that a functional program is built by allowing functions to call other functions that are parameters to the first function. The forEach() function takes another function, the printDay() function. The printDay() function is evaluated in the forEach function for each member of the array[24], and the object and its index are passed as parameters to the function.

The use of functions as data in a language is called a lambda function. A lambda function is a function that is treated as data and can be passed as a parameter to other functions, assigned to variables, or used as any other data type. Another example of using functional programming is given in the Exercises at the end of this chapter.

In the example of Functional Programming in Program 68, the function printDay is only defined to pass to the forEach method. Thus, there is really no reason to give it a name, and the function would be better written as an anonymous function, as in the following example.

```
<script>

    let weekDays = ["Monday", "Tuesday", "Wednesday", "Thursday",
                    "Friday"]
    weekDays.forEach(function(day, index) {
           alert("Weekday " + (index+1) + " is " + day)
    })
</script>
```

Program 69 - Final program to process a form

One Functional Programming pattern, the map reduce pattern, has gained a large following for implementing Parallel Processing and Big Data processing, and so it is useful to be at least somewhat familiar with Functional Programming.

Chapter 3. 6 Exercises

1. The following is Lincoln's Gettysburg Address. Format this using HTML in JavaScript write statements so that each paragraph is separate, and the text is centered. The lines in your program should not be longer than 80 characters.

[24] Note here the printDay function is named, but it is a lambda value since it is passed as data to another function. Do not fall into the trap of believing all anonymous functions are lambda functions, of all lambdas functions are anonymous. For more information about lambda verses anonymous functions, see https://gist.github.com/ericelliott/414be9be82128443f6df.

2. Implement a file dialog in JavaScript, and alert the file name the user has chosen, or a message if no name is chosen.
3. Research modal and non-modal dialog boxes. Discuss the difference between a modal and non-modal dialog? When would you use each?
4. Write a program to read two numbers from textboxes, add them together, and display the answer. Your answer should represent addition (5+6 = 11, not 56).
5. Write a program to read two strings from textboxes, append them together, and display the answer. What does this say operator overloading of the + operator? Can you overload the + operator in JavaScript to produce new semantics? What about other operators in JavaScript?
6. Create an array of 7 elements. Delete the 3rd element (element number 2). What happens to the array?
7. Create an array of 10 elements (e.g. var array = new Array(10)). Try adding 11 elements to the array. What happens?
8. Create an array of 25 numbers. Using the JavaScript array sort with a Lambda function, sort the numbers highest to lowest. You may not use the reverse method, you must use a Lambda function.
9. Process the form you developed at the end of Chapter 2.
10. Agree or disagree with the following statement, and give your reasons. Lambda expressions as defined in Java are not true lambda functions, but syntactic sugar to hide anonymous inner classes.

What you will learn

In this chapter, you will learn:

1. The purpose of CSS
2. How to include CSS into an HTML file
3. What the page header and footer are used for
4. CSS syntax
5. The 4 types of CSS selectors
6. The difference between formatting and semantic tags
7. How to define a header section of a web page
8. How to combine CSS tags to reference specific sections of a document
9. How to include a separate CSS file
10. How to style a simple form

Chapter 4 CSS and styling a web page

When creating a style for a web page, having a consistent look and feel to the application is important. The pages should look similar in format, and the behavior of buttons and menu options should work the same way across an application and suite of applications. Cascading Style Sheets (CSS) is the language used to style web pages, to make the pages look consistent and professional.

The form as it has been presented so far just contains the form elements, with no styling to make it look more presentable. This is in keeping with an age-old programmer prejudice that if something works, who cares what it looks like. But application development is about more than making the page look nice. It is about how to design a User Interface/User eXperience (UI/UX) that makes it easier and more pleasant for the user. A good thing for programmers to remember is a system that is works but does not have users is as useless as a system that does not work.

Because of their implicit bias to simply make a system work, it usually does not fall to the programmer to do the UI/UX. But they must interface with UI/UX designers and need to respect what they do. That is why a basic understanding of CSS is important.

Like JavaScript, CSS is a language that is implemented inside of a web page. CSS uses the HTML `<style>…<style>` tags to contain the CSS attributes. The CSS attributes are mapped to HTML tags, ids, class names, or attributes to define how to style those elements.

This section will cover how to style the form developed in Chapter 2, and thus will present information about CSS as it is needed. At the end of the chapter some extra examples will be given to cover some useful ways to apply CSS for specific formatting.

Chapter 4. 1 Web Page Header and Footer

When designing a user interface, an important part of creating a consistent experience is to design a consistent look and feel for the page. To create a look and feel, the first task is to develop and implement consistent page layout. Here that involves creating a header and footer section for the page.

To start the format for the header should be defined. A typical header presents information such as a title associated with the program or entity, other heading elements (such as the organization), a logo, and author information. Also, in the header is a menu of options for the system, which for now will contain the entries Home, File, and About. The header we will create for this page is as follows.

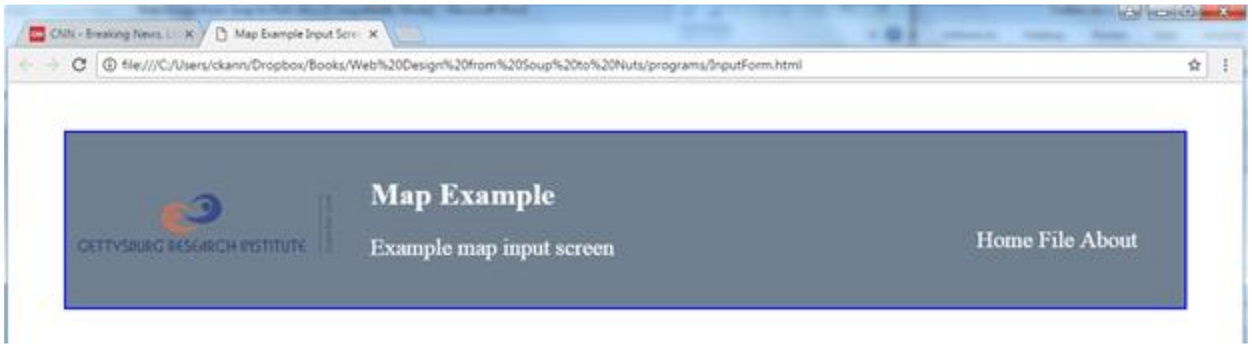

CNN - Breaking News
Map Example Input Scre
file:///C:/Users/ckann/Dropbox/Books/Web%20Design%20from%20Soup%20to%20Nuts/programs/InputForm.html
GETTYSBURG RESEARCH INSTITUTE
Map Example
Example map input screen
Home File About

Figure 5 - Example HTML header

HTML provides two element selectors to make it easy to format the header and the footer of a web page, the `<header>` and `<footer>` tags. This section will format the header element.

Chapter 4.1. 1 CSS Syntax

CSS code is defined in a section of the HTML web page between `<style>` and `</style>` tags[25]. Between these tags, CSS code is embedded into the HTML, just as JavaScript was embedded in the HTML. This CSS code contains display elements, or properties, that inform CSS how to render the page. The format for CSS is the following:

```
<style>
  selector {
    attribute1 : value1;
    attribute2 : value2;
  }
</style>
```

Program 70 - CSS syntax

In CSS, a selector is entered, and then associated with a block of properties to apply to that selector. There are 4 types of selectors that will be covered in this text[26]. They are:

1. Element selectors, which are tags such as the `<header>` and `<p>` tags.
2. Class selectors, which apply to a group of items. Class selectors start with a ".", such as header-`icon` or `.header-desc`, and are them referenced using the "cl`ass=.header-desc`" syntax in the element ag.

[25] CSS styling can also be done directly in an HTML tag using the *style* attribute, but this violates the idea of separating concerns in a web page, and like Unobtrusive JavaScript, best practice says to not mix CSS and HTML.
[26] For a complete list of all selectors and how to use them, see https://www.w3schools.com/cssref/css_selectors.asp.

3. Id selectors, which apply to a single element referred to by its unique id. Id selectors start with a "#", such as "#inputForm". The id attribute in the tag is used to reference the selector.
4. Attribute selectors, which apply to attributes of a tag. For example, to make all textboxes which are readonly have a gray background, the following display attribute tag would be use:

```
<style>
  input:read-only {
    background-color: lightgray;
  }
</style>
```

Program 71 – Attribute Selectors

This attribute informs CSS to take any input tag which has a read-only attribute and make the background color gray.

Each of these selectors will be explained in context in the following sections that style the input form.

Display attributes, or properties, allow the programmer to describe how to display the information that will later be associated with the tag. Properties like font, background-color, border box, indentation, and literally scores of other attributes can be set. A complete list of all attributes that can be set is at https://developer.mozilla.org/en-US/docs/Web/CSS/Reference.

Chapter 4.1. 2 Semantic Tags

The <header> is an element selector. It is also called a semantic tag.

Semantic tags are tags that are intended to simply provide formatting information, but have meaning about the information, or structure of the information, on the page web page. To better understand semantic tags, consider some of the tags in HTML that have been deprecated. For example, the <b> (bold) tag says what to bold the text (what to do with it). The <strong> tag describes the text as important, and to render the text as important, normally by bolding it. Similarly, the <i> (italicize) tag has been deprecated in favor of the <em> (emphasis) tag. In both of these examples, the new tags say something about the text, not something to do to the text.

The <header> tag describes semantic information, that what is contained between the <header> and </header> is a specific part of the page (the header), and not some random division on the page. The header is the block at the top of a page, and the information associated with it is specifically how to format the header of the document.

Chapter 4.1. 3 Setting up the header block

The header of a document is the part of the document that is displayed at the top of the web page. The properties to define how the header should appear are set as display attributes when the header tag is defined. The header tag is defaulted as follows in CSS.

```
<style>
  header {
    display: block;
  }
</style>
```

Program 72 – Header definition

The default `<header>` tag includes just one attribute, `display:bloc`k, which specifies that the header sites in a block by itself at the top of the document. In CSS, a `block`[27] attribute means create a space that spans from the left to right margin of the element or page, allowing no other elements to be displayed to the left or the right. This is the normal behavior of a header, which generally spans the entire length of the top of the page. Like any display attribute can be overridden if needed.

The header for the example page developed in this chapter will be built in stages in the subsections below.

Chapter 4.1. 4 Changing the background and text colors

The first attributes to be change will make the box a dark color (we will use `slategray`) and make the text `white`. This is done by setting `the attributes background-color` to `slategray,` and `color` to `white`.

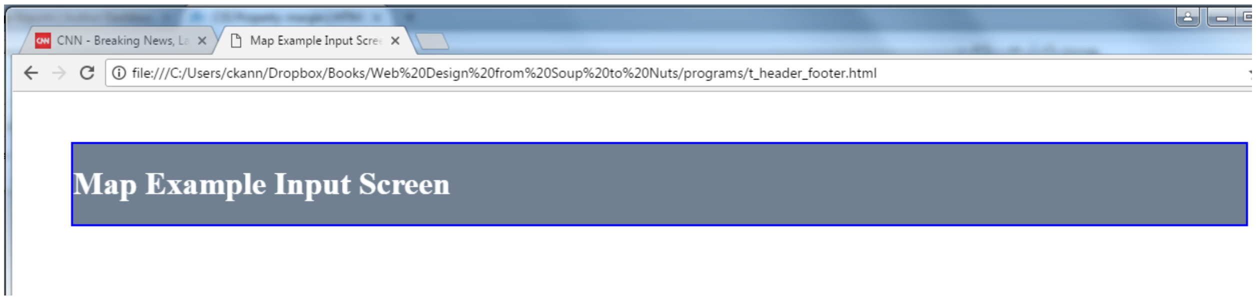
To make the header stand out more, a box will be placed around the header, using a 2-pixel large blue line. The header is also indented 50- pixels from the sides and top and bottom. This results in a first pass for the header in the following code.

```
<html>
  <head>
    <title>Please change this to the title of your page </title>
    <style>
      header {
        margin : 50px;
        border : 2px solid blue;
        background-color : slategray;
        color : white;
      }

    </style>
  </head>

  <body>
    <header>
      <h1>This shows the header style.</h1>
    </header>
  </body>
</html>
```

Program 73 – Header attribute settings

The page resulting from this program looks as follows:

[27] For a complete listing of all display types, see https://www.w3schools.com/cssref/pr_class_display.asp.


CNN - Breaking News, L
Map Example Input Scre
file:///C:/Users/ckann/Dropbox/Books/Web%20Design%20from%20Soup%20to%20Nuts/programs/t_header_footer.html

Map Example Input Screen

Figure 6 - First pass at the web page header

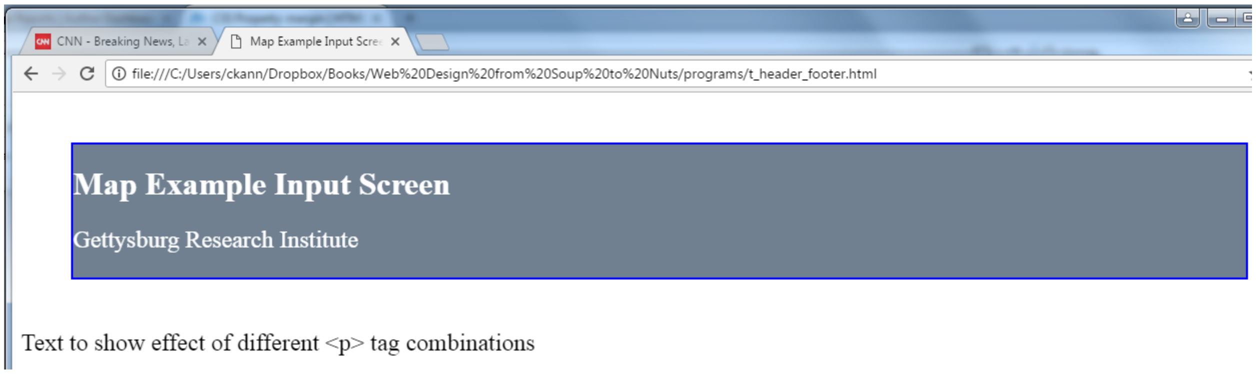
Chapter 4.1. 5 Changing the font size using the <p> tag

The text inside of the header should be larger than the normal text in the document, to make it stand out. Normally text is placed inside of `<p>` (paragraph) tags in a document, and the styling attributes applied to the <p> tag. The following example shows how the font-size can be changed for the `<p>` tag to be 150% of its normal size.

```
<html>
  <head>
    <title>Map Example Input Screen </title>
    <style>
        header {
          margin : 50px;
            border : 2px solid blue;
          background-color : slategray;
          color : white;
        }

        p {
          font-size : 150%;
        }

    </style>
  </head>

  <body>
    <header>
        <h1>Map Example Input Screen</h1>
        <p> Gettysburg Research Institute<p>
        </p>
    </header>
      <p>
          Text to show effect of different &lt;p&gt; tag combinations
      </p>
  </body>
</html>
```

Program 74 – Making the text 150% of the normal size

When running this page, it is obvious that all the text in the document is 150% of the normal size, not just the text in the header. Changing the `<p>` tag caused the text in all of the `<p>` tags in the entire file to have the increased size of text


CNN - Breaking News, L
Map Example Input Scre
file:///C:/Users/ckann/Dropbox/Books/Web%20Design%20from%20Soup%20to%20Nuts/programs/t_header_footer.html

Map Example Input Screen

Gettysburg Research Institute

Text to show effect of different <p> tag combinations

Figure 7 - Second pass at the web page header

To correct this so that the <p> tag will only affect the header, CSS allows tags to be combined so that changes to only have effect when use inside of a specific division or section of the page. The syntax for this is:

```
header p {
    font-size : 150%;
}
```

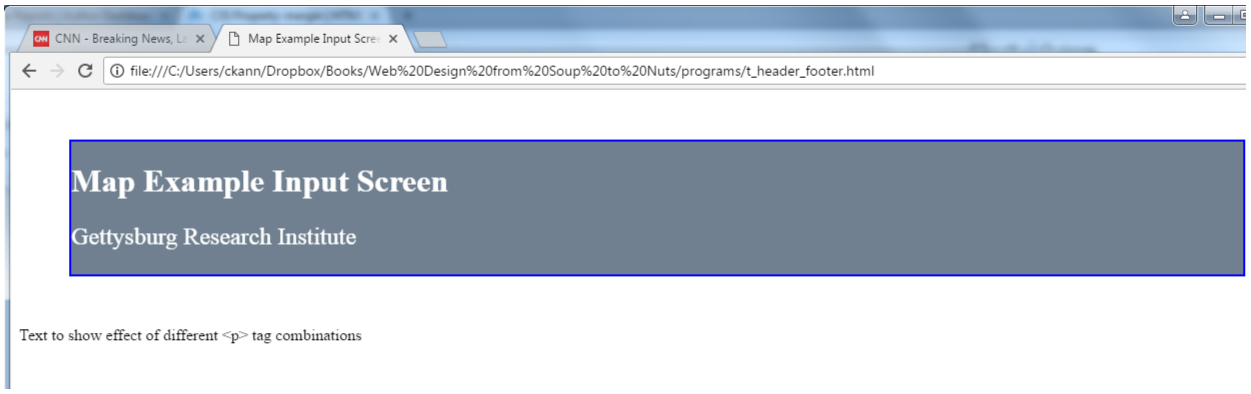
Program 75 - Combining the header and p tags

The code for the previous page has been changed so that only in the header does the <p> tag change the size of the text to 150% of the normal size.

```
<html>
  <head>
    <title>Map Example Input Screen </title>
    <style>
      header {
        margin : 50px;
        border : 2px solid blue;
        background-color : slategray;
        color : white;
      }

      header p {
        font-size : 150%;
      }
    </style>
  </head>

  <body>
    <header>
        <h1>Map Example Input Screen</h1>
        <p> Gettysburg Research Institute<p>
        </p>
    </header>
      <p>
          Text to show effect of different &lt;p&gt; tag combinations
      </p>
  </body>
</html>
```

Program 76 – Paragraph text only affecting the header of the document

The result is the `<p>` tag only effecting the text in the header of the document.


CNN - Breaking News, L
Map Example Input Scre
file:///C:/Users/ckann/Dropbox/Books/Web%20Design%20from%20Soup%20to%20Nuts/programs/t_header_footer.html

Map Example Input Screen

Gettysburg Research Institute

Text to show effect of different <p> tag combinations

Figure 8 - Third pass at the web page header

Chapter 4.1. 6 Dividing up the header block

In Figure 6, the header had 3 separate areas, one for the logo, one for the title and page information, and one area for the options to be implemented. To divide a web page into parts, a `<div>`[28] (or division) tag is used. The `<div>` tag is the most useful of all CSS tags, as it can be used to break up a web page into different areas, and to assign different styling or information types to those areas. For now, it will be used to break the header into 3 pieces.

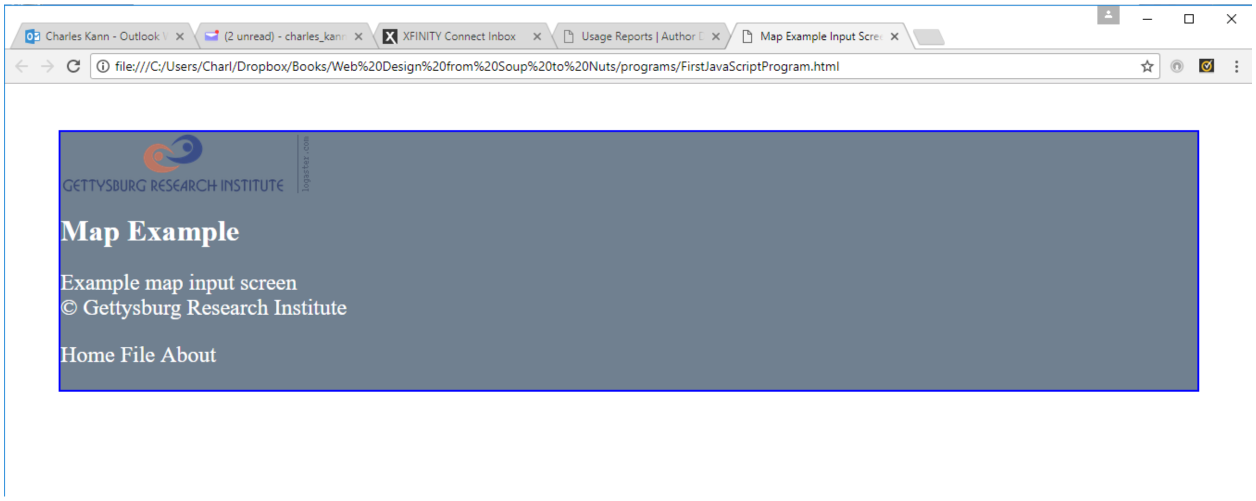
To break the header up into 3 pieces, 3 divisions are created. These are all placed inside of the header section and can be thought of as *parts of* or *children of* the header section. This allows these 3 divisions are to appear inside of the web page header. These 3 division are shown below.

```
<header>
  <div id="header-icon">
    <image src="GRI_logo.png" />
  </div>
  <div id="header-desc">
    <h1>Map Example</h1>
    <p>
      Example map input screen <br>
      &copy; Gettysburg Research Institute
    </p>
  </div>
  <div id="header-menu">
    <p>
      Home    File    About
    </p>
  </div>
```

[28] The current thinking about CSS is that an HTML document should be divided into divisions (using the <div> tag) when the page is being divided for styling purposes, and into sections (using the semantic <section> tag) when it is being divided into parts that would be equivalent to areas in an outline. There is no difference other than the semantic meaning of the section as part of a larger document organization. As this document is concerned with formatting program web pages, it will use only divisions to keep the confusion to a minimum.

Program 77 - Dividing the header into 3 divisions.

This code only divides up the header so that the DOM knows they are 3 separate areas inside of the header. The DOM does not know how to display them, so it just places them in separate blocks, as shown below.


Charles Kann - Outlook
(2 unread) - charles_kann
XFINITY Connect Inbox
Usage Reports | Author
Map Example Input Screen
file:///C:/Users/Charl/Dropbox/Books/Web%20Design%20from%20Soup%20to%20Nuts/programs/FirstJavaScriptProgram.html
GETTYSBURG RESEARCH INSTITUTE

Map Example

Example map input screen
© Gettysburg Research Institute

Home File About

Figure 9 – Un-styled div sections

To make the divisions work as intended, they must be styled.

To style each division, the divisions must be given a way to reference them. If there are a number of divisions to be styled the same, normally a `class` variable is defined in CSS, and referenced using the `class` attribute in the tags. If the division is to be uniquely styled, it will be given an `id` attribute, and CSS will style it using an `id` variable. Since the divisions in the header will be unique for the header, they will use `id` variables to reference and style them. Note in the program above, the individual divisions have been assigned the names `header-icon`, `header-desc`, and `header-menu`. This follows CSS conventions which favor hyphenated names.

When referencing `id` names in CSS, the name is prepended with a hash tag (`#`). For example, to set the `header-desc` to be a display of `inline-block` the following would be used.

```
#header-desc {
  display : inline-block;
  margin : 25px;
}
```

Careful readers will remember that this is how id variables were referenced in JQuery. JQuery uses CSS naming conventions to access DOM elements. This was alluded to earlier, but now is made explicit. Thus, to use JQuery at least a passing knowledge of CSS is necessary to know how to access DOM elements

The main purposed of the CSS for these header divisions is to set the display parameters for them. The choice of inline-block tells the DOM to lay these divisions out as blocks, one after another on the same line. The margin tells the DOM how far to space the elements apart.

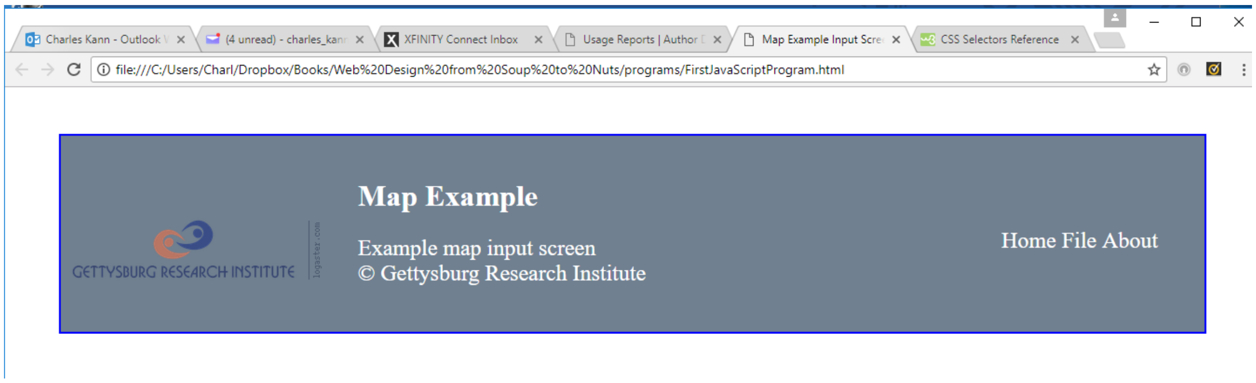
The last attribute set in the header divisions is the `float` attribute. The `float` attribute defines which side of the line to place the attribute on. Normally inline and inline-block elements start at the far left and are placed after each other from left to right. By saying `float: right`, the program is saying place this element starting at the far right, and then place subsequent elements moving from right to left. Menus items generally are place at the right of the header to balance the header.

The final CSS code for this example is shown below.

```
<html>
  <head>
    <title>Map Example Input Screen </title>
    <style>
      header {
        margin : 50px;
        border : 2px solid blue;
        background-color : slategray;
        color : white;
      }

      header p {
       font-size : 150%;
      }


     #header-icon {
       display : inline-block;
       margin: 50px 10px 50px
     }

     #header-desc {
       display : inline-block;
       margin : 25px;
     }

     #header-menu {
       display : inline-block;
       float: right;
       margin : 75px 50px 50px 50px;
     }
    </style>
  </head>

  <body>
    <header>
      <div id="header-icon">
        <image src="GRI_logo.png" />
      </div>
      <div id="header-desc">
        <h1>Map Example</h1>
        <p>
          Example map input screen <br>
          &copy; Gettysburg Research Institute
        </p>
      </div>
      <div id="header-menu">
      <p>
            Home     File     About
      </p>
    </div>
  </header>

  </body>
</html>
```

Program 78 - Completed header

This program gives the final, completed header for the page.


Charles Kann - Outlook
(4 unread) - charles_kann
XFINITY Connect Inbox
Usage Reports | Author
Map Example Input Scre
CSS Selectors Reference
file:///C:/Users/Charl/Dropbox/Books/Web%20Design%20from%20Soup%20to%20Nuts/programs/FirstJavaScriptProgram.html
GETTYSBURG RESEARCH INSTITUTE

Map Example

Example map input screen
© Gettysburg Research Institute

Home File About

Figure 10 – Completed web page header

Chapter 4.1. 7 Managing CSS

To make managing a project and writing and reading source code easier, the CSS source code for a web page is generally kept in a separate file from the html source. This file separation achieves a number of positive benefits: 1) it keeps the styling information separate from the application information, which makes it easier to read and understand the HTML program; 2) By separating the styling and program, UI designers can work on the styling the application without interfering with the programmers developing the application.

For the header developed above, the file containing the CSS could be kept in the file WebMapExample.css, and the HTML could be kept in the file MapExample.html. The CSS file is included in the web page using a <link> tag.

```
<link rel="stylesheet" type="text/css" href="WebMapExample.css">
```

The contents of the WebMapExample.css file is:

```
header {
  margin : 50px;
  border : 2px solid blue;
  background-color : slategray;
  color : white;
}

header p {
  font-size : 150%;
}

#header-icon {
  display : inline-block;
  margin: 50px 10px 50px
}

#header-desc {
  display : inline-block;
  margin : 25px;
}

#header-menu {
  display : inline-block;
  float: right;
  margin : 75px 50px 50px 50px;
}
```

Program 79 - WebMapExample.css file

The application file, MapExample.html, is now much simpler, and easier understand.

```
<html>
  <head>
    <title>Map Example Input Screen </title>
    <link rel="stylesheet" type="text/css" href="WebMapExample.css">
  </head>

  <body>
    <header>
      <div id="header-icon">
        <image src="GRI_logo.png" />
      </div>

      <div id="header-desc">
        <h1>Map Example</h1>
        <p>
          Example map input screen <br>
          &copy; Gettysburg Research Institute
        </p>
      </div>

      <div id="header-menu">
      <p>
           Home    File    About
      </p>
    </div>
  </header>

  </body>
</html>
```

Program 80 - MapExample.html

In the real world, most programmers will deal with the HTML and JavaScript, and UI/UX designers will handle the CSS for a web page. However, both sides, programmers and UI/UX designers, should know enough about the other technologies to be able to interface effectively with their counter parts.

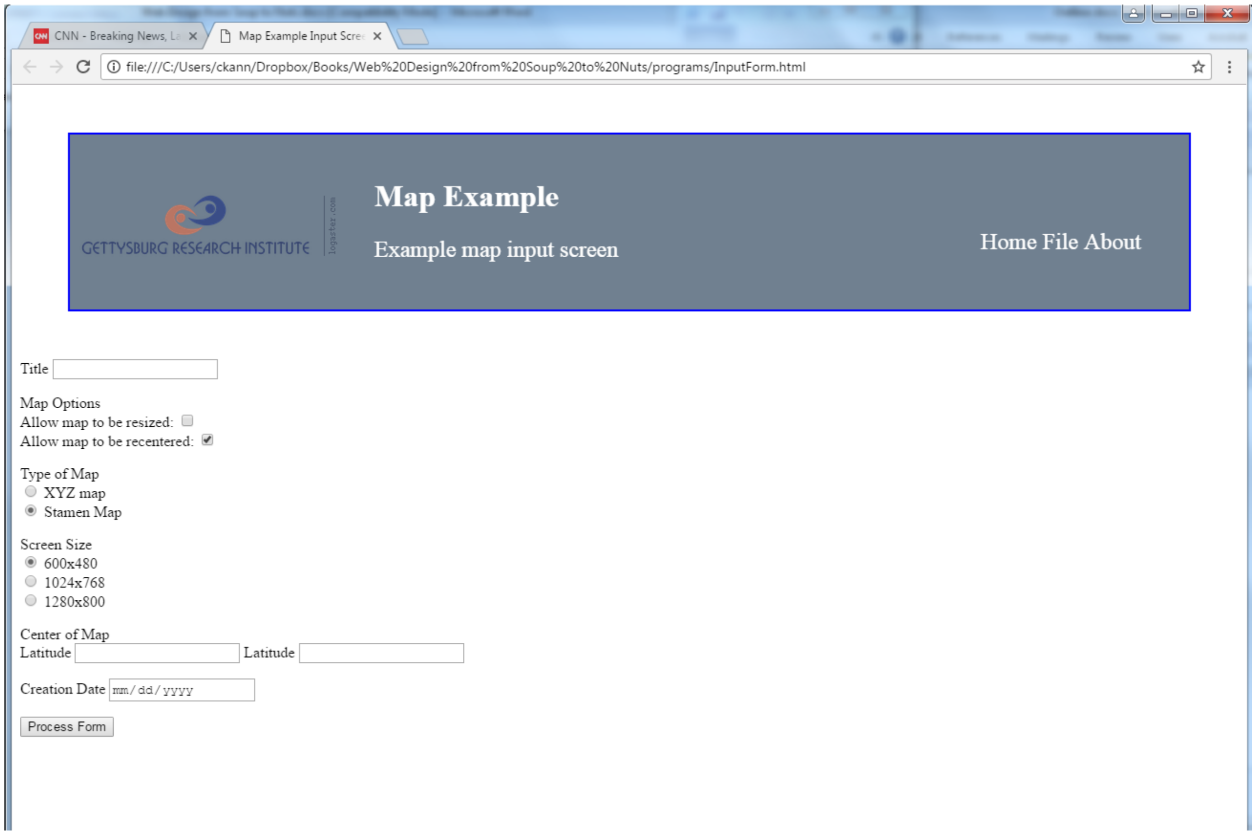
Chapter 4.1. 8 Quick Review

1. What are the four types of CSS tags? Give an example of each and describe how you might use them.
2. In your own words, describe *semantics*? What is an HTML semantic tag, and how does it differ from a formatting tag?
3. What is the difference between a display:block and a display:inline-block attribute? How were they used to set up the header division of the web page?
4. How would you hide a division of an HTML document?
5. Looking at the following web page, https://www.w3schools.com/cssref/pr_class_display.asp, discuss how you might use the display attribute to format a web page?
6. How might you specify a paragraph (<p>) tag so that it only impacts the text in the #header-desc?
7. How would you format the <p> tag so that all of the output in the document associated with an error message would be in red?
8. Create a CSS file and include it in another HTML file. Does the name have to be ".css"?

Chapter 4. 2 Adding the form to the page

After the header has been developed, the form is added to the page. For now, only the display of the form will be presented to make the concepts clearer. Later, this chapter will show how CSS can be used with JavaScript to control display of the form.

If the form is simply added to the page, as shown below, it looks very unprofessional. While it is functional, it is ugly and would not inspire confidence from the users that the web page actually works.


CNN - Breaking News, La
Map Example Input Scre
file:///C:/Users/ckann/Dropbox/Books/Web%20Design%20from%20Soup%20to%20Nuts/programs/InputForm.html
GETTYSBURG RESEARCH INSTITUTE
Map Example
Example map input screen
Home File About

Title

Map Options
Allow map to be resized: [ ]
Allow map to be recentered: [✓]

Type of Map
○ XYZ map
● Stamen Map

Screen Size
● 600x480
○ 1024x768
○ 1280x800

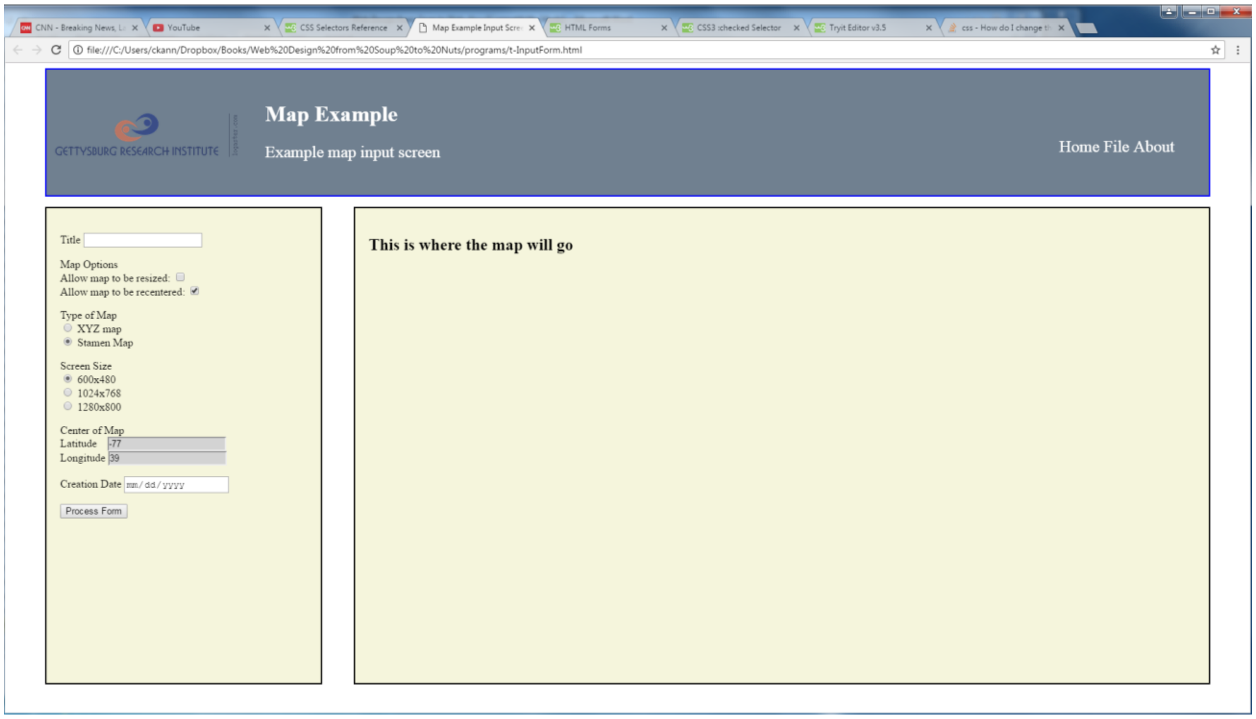
Center of Map
Latitude Latitude

Creation Date mm/dd/yyyy

Process Form

Figure 11 – Completed web page header

A much nicer layout of the form follows. The following is the form display that will be styled using CSS. Following the form, the CSS and HTML are presented here and an explanation of all the elements is given.


Map Example

Example map input screen

Home File About

Title

Map Options
Allow map to be resized: ☐
Allow map to be recentered: ✓

Type of Map
○ XYZ map
● Stamen Map

Screen Size
● 600x480
○ 1024x768
○ 1280x800

Center of Map
Latitude -77
Longitude 39

Creation Date mm/dd/yyyy

Process Form

This is where the map will go

Figure 12 – Completed web page header

CSS

```
header {
    margin : 5px 50px 5px 50px;
    border : 2px solid blue;
    background-color : slategray;
    color : white;
}

header p {
    font-size : 150%;
 }


.header-icon {
    display : inline-block;
    margin: 50px 10px 50px
}

.header-desc {
    display : inline-block;
    margin : 25px;
}

.header-menu {
    display : inline-block;
    float: right;
    margin : 75px 50px 50px 50px;
}
```

```
#inputForm {
    margin : 10px 5px 5px 50px;
    background-color : beige;
    border : 2px solid black;
    display: inline-block;
    float : left;
    padding : 20px;
    width : 20%;
    height : 70%;
}

#map {
    margin : 0px 50px 5px 5px;
    background-color : beige;
    border : 2px solid black;
    display: inline-block;
    float : right;
    padding : 20px;
    width : 67%;
    height : 70%;
}


input:-moz-read-only { /* For Firefox */
    background-color: lightgray;
}


input:read-only {
    background-color: lightgray;
}
```

Program 81 – Form CSS example

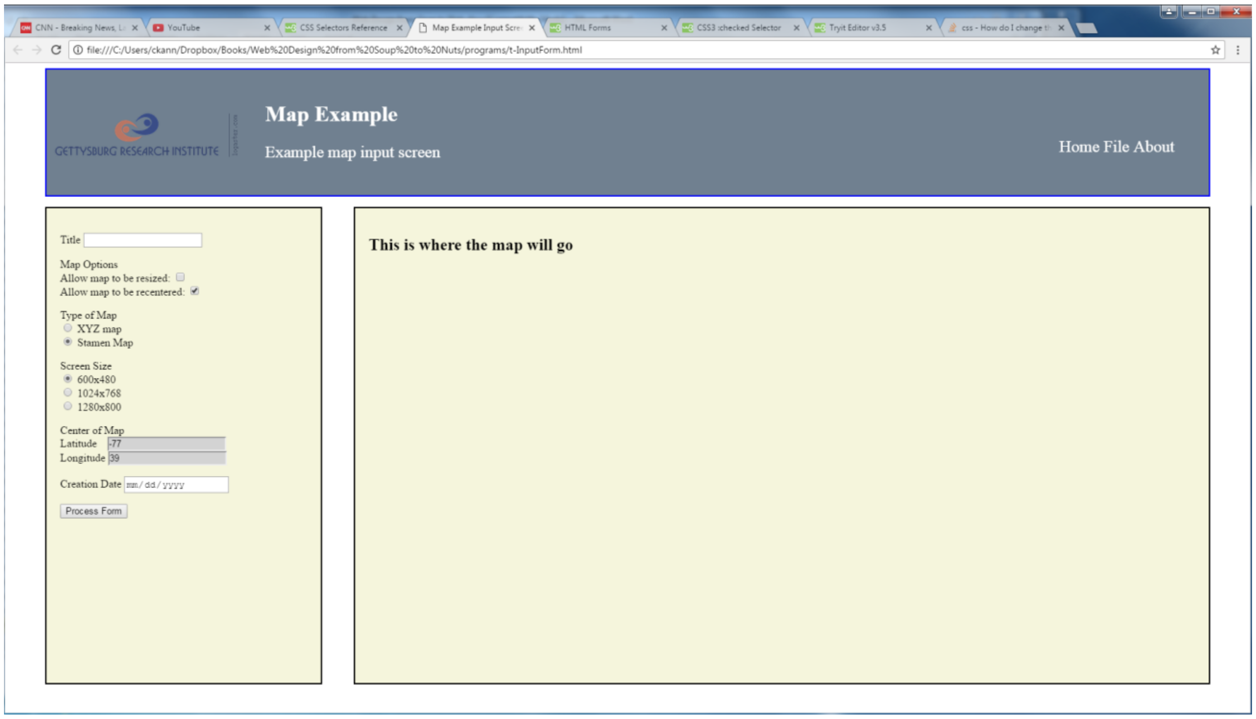
The CSS for the head has been defined, and that leaves just 4 CSS tags to cover here. The first are the `#form` and `#map` id tags. These two tags create boxes displayed on the web page to contain other form elements. The `#form` is 20% of the size of the page, and the `#map` is 67% the size of the page and is anchored to the right of the page using the float:right attribute. The `inline:block` attribute tells CSS to put both on the same line. Since they only take up 87% of the line, 13% of the line will not be included in either division and left blank between the two pages. Both or the div sections have black boarder 2 pixels in size, and both will take up 70% of the height of the page, with the rest used by the header. Finally, the padding says leave some room on the right and left of the divisions before rendering the text.

The final two tags, the input:-moz-read-only and input:read-only tags, are to gray out any input tags which are read only, e.g. the value for these input fields cannot be typed in by the user and must be set programmatically. These are the read only fields on the map for latitude and longitude.

HTML

```
<html>
  <head>
    <title>Map Example Input Screen </title>
    <link rel="stylesheet" type="text/css" href="WebMapExample.css">
  </head>

  <body>
    <header>
      <div class="header-icon">
       <image src="GRI_logo.png" />
      </div>
      <div class="header-desc">
        <h1>Map Example</h1>
        <p class="header-p">
          Example map input screen
        </p>
      </div>
      <div class="header-menu">
        <p class="header-p">
          Home    File    About
        </p>
      </div>
    </header>

    <section id="inputForm" />
      <p>
        <label id="l1" for="title">Title</label>
        <input type="text" id="title" size="20">
      </p>
      <p>
        Map Options<br>
        <label id="l2" for="resize">Allow map to be resized:
        </label>
          <input type="checkbox" id="resize"/>
          <br/>
        <label id="l3" for="recenter">
             Allow map to be recentered:
        </label>
            <input type="checkbox" id="recenter" checked />
      </p>
      <p>
        Type of Map<br>
        <input type="radio" name="maptype" id="XYZMap"
               value="XYZ Map"/>
        <label id="label1" for="XYZMap">XYZ map </label>
        <br/>
        <input type="radio" name="maptype" id="StamemMap"
                    checked />
        <label id="label2" for="StamenMap">Stamen Map
        </label>
      </p>
      <p>
        Screen Size<br>
        <input type="radio" name="screenSize" checked
               id="600x480" value="600x480" />
        <label id="label3" for="XYZMap">600x480 </label>
        <br/>
        <input type="radio" name="screenSize" id="1024x768"
               value="1024x768"/>
        <label id="label4" for="XYZMap">1024x768 </label>
        <br/>
        <input type="radio" name="screenSize" id="1280x800"
               value="1280x800"/>
        <label id="label5" for="XYZMap">1280x800 </label>
      </p>

      <p>
        Center of Map<br>
        <label id="label1" for="lat">Latitude&nbsp;&nbsp;&nbsp;
        </label>
        <input type="number" id="lat" value="-77" readonly /><br />
        <label id="label2" for="long"  >Longitude </label>
        <input type="number" id="long" value="39" readonly />
      </p>

      <p>
        <label id="label1" for="creationDate">Creation Date
        </label>
        <input type="date" id="creationDate"/>
      </p>

      <input type="button" value="Process Form" />
    </section>

    <section id="map">
      <h1> This is where the map will go </h1>
    </section>
  </body>
</html>
```

Program 82 – CSS with form completed example.


file:///C:/Users/ckann/Dropbox/Books/Web%20Design%20from%20Soup%20to%20Nuts/programs/t-InputForm.html
GETTYSBURG RESEARCH INSTITUTE
Home File About
Process Form

Map Example

Example map input screen

Title

Map Options
Allow map to be resized: ☐
Allow map to be recentered: ☑

Type of Map
○ XYZ map
● Stamen Map

Screen Size
● 600x480
○ 1024x768
○ 1280x800

Center of Map
Latitude -77
Longitude 39

Creation Date mm/dd/yyyy

This is where the map will go

Figure 13 – Completed web page header

For the HTML for the page, the only change was to add two section tags to create sections for the form and the map. The section tag is a div tag that has a semantic meaning. A *section* to a for is a completely separate part of the form. A *division* has no semantic meaning.

Chapter 4. 3 Exercises

Take the form you developed in Chapter 3 and style it using CSS.

Part II: JavaScript Objects and CRUD Interfaces

Part II of this book gives an overview of how to create object in JavaScript. This object will then be used as the basis for to create a Create-Read-Update-Delete (CRUD) interface. The interface will initially persist objects LocalStorage for the web site. A server will then be created to persist the objects to a Mongo database, and Asynchronous JavaScript And XML (ajax) protocols will be used to connect the user interface to the server. This will create a complete web application, from soup to nuts.

This application is unreasonably simplistic. However, it opens up and shows how to use all parts of working web application. Once this application is complete, the reader will have a basic idea of all parts of what is called the *full application stack*.

What you will learn

In this chapter, you will learn:

1. What version of JavaScript is used in this text, and why
2. Global (simple) object creation in Java
3. The nature of objects as property maps
4. JSON and how to serialize an object with JSON
5. How to implement and use Constructor Functions
6. What are protocols, and how do protocol chains work
7. How to use the DOM to find program elements such as Constructor Functions
8. How to use Constructor Functions to reconstruct objects
9. JavaScript scoping and closure
10. Models for JavaScript objects

Chapter 5 Objects in JavaScript

The procedural concepts of JavaScript were covered were covered in Chapter 3. These procedural aspects at least appeared to be consistent with other, more common Procedural (or Imperative) and Object-Oriented Programming (OOP) languages.

In some ways, though, JavaScript may already seems strange to the reader. For example, as was previously pointed out, JavaScript has characteristics that are not Procedural; JavaScript has true lambda functions (as opposed to languages like C# and Java that use lambda functions as syntactic sugar to hide other language constructs), and can be used as a Functional language. Functional programming is often a source of confusion to novices just getting used to JavaScript. In addition, arrays are not primitive data structures, where elements are accessed using a base address and index. In JavaScript, arrays are hash maps, and the index numbers are actually keys. This makes accessing of elements in an array seem strange or almost wrong to some programmers. Finally, JavaScript dynamic typing goes against everything most programmers learned in their first programming class, where all types are statically typed.

But perhaps the strangest concept in JavaScript is its Object paradigm, and that is what will be covered in this chapter. It is not at all like the Object paradigm built into more common OOP languages such as Java/C#/C++.

To start, the reader should be aware of the basic definition of an object, which is that objects are defined as collections of data elements and behaviors. This definition is true of more common OOP languages such as Java/C#/C++, and JavaScript.

Languages such as Java/C#/C++ then refine the definition so that a type used for an object definition is statically defined at compile time. The new operation in these languages then means to *instantiate* (or create a completed final instance) of that object in memory. The definition of that object in memory then never changes.

Language constructs such as the Java interface go a long way towards alleviating some of the worst effects of this static model and made a true abstract definition in the language without the

C++ multiple inheritance confusion. This is why interfaces are so successful in making a better object model in Java than C++. But the fact remains that abstract types (interfaces) and classes are always statically defined in Java/C#/C++, and the new operator instantiates a static, final copy of the object.

JavaScript takes the definition of an object as collection of data elements and behaviors, and then uses a completely different view of how data elements and behaviors are defined. This view is not wrong (as some Java/C#/C++ programmers would believe), but it is at odds with the more traditional view of OOP languages. And until this view is understood and accepted for its own merits, it will appear wrong to programmers using an invalid cognitive view of JavaScript objects.

The purpose of this chapter is to explain the JavaScript view of objects. It will present this view with no reference to the view of objects from other languages, as to try to bring in an understanding of how objects are treated in a language like Java is much more a hindrance than a help. A reader with no Object-Oriented-Programming background is likely at an advantage to someone who feels they really grok Java.

The chapter will be divided as follows.

Chapter 5.1 will cover the specifications for the JavaScript, as defined by the European Computer Manufacturers Association (ECMA) and discuss why this book is based on JavaScript 5 (or ECMA 5) instead of JavaScript 6 (ECMA 6+[29]) or higher,

Chapter 5.2 will describe in more detail why JavaScript is hard for programmers coming from other languages to understand.

Chapter 5.3 will look at basic objects in JavaScript. The section will cover associative arrays and give a more correct model of what an array is in JavaScript. It will then show how objects in JavaScript are property maps, and not instances of classes as in a class-based OOP. Finally, it will cover JavaScript Object Notation (JSON) format, which is a way to represent JavaScript objects externally.

Section 5.4 covers the prototype-based object model used by JavaScript. Functions and their prototypes will be explained, and how properties are resolved in JavaScript will be explained. JSON will be revisited to show how a fully functioning object can be serialized to a string and brought back into a program as the object type is was.

Section 5.5 covers JavaScript scope, and introduces JavaScript closures.

Section 5.6 will explain instance-based OOP languages, and how objects are implemented and used in these languages, so that the reader can then compare the properties and behaviors.

At the end of this chapter, the reader should understand the object model that will be used in this text, and that will be used in the next chapter.

[29] I will use the term ECMA6+ to mean versions of JavaScript that are ECMA6 or higher.

Chapter 5. 1 Why Use ECMA 5 as a basis?

There are a number of reasons to choose ECMA5 over ECMA6+ as a basis for this book. The first is that any future implementation of JavaScript[30] will have to have the ability to be *transpiled* into ECMA 5. All of the nifty features built into ECMA6+ are built on a feature in ECMA5. My personal belief is that to understand a concept, the basis on which the concept is built should be known. If you want to program in C, you should know Assembly. If you want to program in Assembly, you should understand Computer Architecture and Computer Organization. While understanding the foundations on which ideas are built might not be necessary, there is almost always cases where not understanding them leads to difficult problems that would easily be fixed if a programmer understood the next lower layer of abstraction.

The second reason for not choosing ECMA6+ is that the language is about making JavaScript *more concise*, e.g. add new features to the language that make it easier to write, probably harder to understand, and which definitely hide details that experienced programs do not need, but which hinder the understanding of novice programmers trying to piece together how things are connected. Once a programmer has an understanding of the parts, conciseness and ease are nice-to-haves. But trying to learn how to implement a function when there are a dozen different syntactic ways to define it (using the function statement, arrow functions with parameters, arrow functions with one parameter, block-scoped functions, and I am sure I missed some) is not my idea of fun.

The third reason for not using ECMA6+ is that it seems to be built with the idea that programs will be written with a framework (React, Angular, Vue, Ember, yada, yada, yada). This is not a book about any of those frameworks or frameworks in general. The intent is to teach CS students CS principals and philosophies. Students can learn a framework, anyone one of which will be relegated to *legacy* status is 5 years, after they graduate.

However, the biggest reason for choosing ECMA5 is that this is a book about understanding CS principals, paradigms, and philosophies, and how to put them together to build a program. This means limiting the complexity of the surrounding environment and concentrating on the underlaying infrastructure that illustrates interesting CS principals. The concept of unstructured data, as is laid out in JavaScript object representation. Object serialization and its realization in JSON data. Storing and retrieving this serialized data. Variable scoping, and specifics like function scoping that result in closures. Property maps and prototypes. These are issues that are of interest in a CS education. Syntactic sugar to make it easier to write programs, or to make a system more accessible to novice CS programmers, is the job of industry. I feel has no place in a CS education.

Chapter 5. 2 Is JavaScript just plain weird?

At some point, nearly every programmer coming to JavaScript from another language will ask themselves the question, “Is JavaScript just plain weird?”. Nothing in the language makes sense based on how they have learned to program, and so JavaScript just seems inconsistent and plain wrong. But let me assure you that there is nothing wrong with JavaScript. It is perfectly

[30] At least until Web Assembly (WASM) becomes the accepted standard, which is still years away. But even WASM will have its roots in JavaScript 5.

consistent and makes very good sense. JavaScript is just different, and at odds with unstated (and probably unrecognized or unknown) cognitive models and metaphors that the programmer accepts as their reality. When I hear someone say how strange JavaScript is, the problem almost always comes from the programmer trying to apply a wrong understanding to the language.

The problem is based in human nature. When approaching a new concept that appears similar to another concept they already know, people seem to be hardwired try to understand the new concept by creating a metaphor of the old concept works and trying to apply it to the new concept. If the metaphor fails, many people will simply give up on understanding a new and perfectly valid concept, saying simply it doesn't make sense. The new concept isn't wrong, the metaphor used to understand it is wrong[31].

I personally am convinced that this is why most people find STEM disciplines so hard. The concepts and ideas in the Liberal Arts and Social Sciences are pretty malleable, and even an incorrect metaphor can be twisted and combined with a twisted version of the concept to create a metaphor that *sort-of* works as explanation of the concept. But in truth the metaphor and the understand are actually invalid.

In STEM, concepts and ideas are much less forgiving, and mean exactly what they mean. Concepts cannot be twisted to fit an incorrect metaphor. STEM forces people to give up incorrect metaphors and adopt new metaphors; a very difficult task.

To better understand what a metaphor is, and why they have to be understood carefully, consider a very common metaphor in CS, that of a stack data structure and a cafeteria tray dispenser, such as the one shown in the figure below.

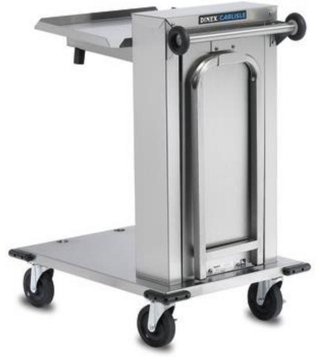


Figure 14 - Tray Dispenser

When learning the stack data structure, the metaphor that is traditionally used is that a tray is placed on the top of the stack of trays, and so the last tray placed is the first one accessed, and a stack is just a Last-In-First-Out (LIFO) queue.

[31] Godel's theorem proves that no metaphor can ever be sufficient, as complete systems have undecidability, and decidable systems are incomplete. A large part of education should be helping people understand this and teaching them how to give up invalid metaphors when they no longer work.

But this metaphor fails quickly. Consider, what is the Big-O of placing a new tray at the top of the dispenser? Since all of the trays must be moved, it is O(n). But as anyone programming a stack data structure knows, placing a new element in a stack is O(1). Something is wrong, and what is wrong is the metaphor[32].

It is only when the student comes to understand a stack as it is implemented in an array that a more correct, and much more useful metaphor, is understood.

The problem with Objects in JavaScript can often be traced to a metaphor problem. Object Oriented Programing (OOP) is a way of abstracting a problem around data which makes up the object, and behaviors that can be applied to the object. There is nothing in the definition that talks about implementation, and there are many ways to implement the basic concept of an object.

In more traditional OOP, a class creates a type that is instantiated to create an Object. This Object is defined by the data (instance variables) and behaviors (methods) that are defined in the class. The class represents a type, or model, of what instance of the class will look like when the program is run, and a variable of the class type is created. The type (class) cannot be changed after compilation, and new behaviors and data cannot be added. In Java, and other traditional OOP languages, the definition of the type is statically defined when the program is compiled. The type of variable can be changed at run time, but only by creating a new variable in memory, and that variable must also correspond to a statically defined class.

JavaScript was specifically implemented using a different way to type and create variables. In JavaScript, the type of the variable is same as the last value it was assigned to. This has been seen previously in the text and was called dynamic typing. But of even greater significance to the discussion hear is that the type itself is dynamically typed and can be modified at runtime. This means that variables of a specific type can be changed to add new properties (what are fields in a class in static OOP), and even the functions associated with the object can change.

In JavaScript the data and type of a variable are dynamic and can be changed while the program is running. A metaphor based on the idea of creating classes to instantiate instances of a type makes no sense to JavaScript.

This view of an object is in keeping with the definition of OOP. Remember that OOP is a way of abstracting a problem around data which makes up the object, and behaviors that can be applied to the object. It is just the abstraction in JavaScript for the object occurs at runtime, not compile time. The unstated assumption of most new JavaScript programmers that the metaphor of a static definition and instantiation is simply not relavent.

To implement this runtime definition of object types, JavaScript implements an Object as a set of properties and functions in a property set, or hash map, and not in static classes. When an object property or function is needed, the hash map is queried to find the property or function, and then that property accessed, or function executed. To implement adding new properties or functions to the object, they properties and functions are added to the hash map.

[32] One of my favorite improper use of metaphors is to ask, "What color is faith?". The obvious answer is yellow, as the Christian Bible describes faith as being "as a mustard seed', and mustard seeds are yellow. This is obvious non-sense, but I personally encounter this type of reasoning many times every day.

The astute reader will suddenly now have an AHA moment when they will understand the importance of Lambda functions. Lambda functions are functions that are treated like data. As such, functions do not exist in statically compiled languages like Java. Functions in JavaScript are simply data and can be stored and changed in an object just like any other data.

Thus, the JavaScript OOP model is built around Lambda functions, and they are what allow this object model to work. Properties in the property map for an object are not simply data items, but also the functions, It is at this point that a Java programmer will have to throw up their hands and admit that a metaphor of a class-based OOP completely fails when understanding JavaScript.

The lack of a types and structured objects in JavaScript flies in the face of everything many programmers have learned about objects, and even seems to defy logic. There are two things a programmer can do. The first is what many programmers do, accept their metaphor of how things work and believe that the JavaScript model of objects must be wrong.

This view is an unreasoned bias, or prejudice, that cannot be logically defended. The JavaScript model works, and that in itself is sufficient reason to accept that what JavaScript is doing makes sense. The only valid response is for the person learning JavaScript to change how they understand OOP[33].

This does not stop programmers from trying to apply Java metaphors to the concepts in JavaScript. But creating these metaphors is much harder, and in the long run much less fruitful, then simply learning how JavaScript works. And learning JavaScript has an added advantage that using unstructured of the data is often a very nice way to easily solve problems that would be very difficult in a structured, type-based approach.

Chapter 5. 3 Basic Objects in JavaScript

When thinking about OOP, remember that at its core OOP is an abstraction mechanism that defines objects by a set of data and behaviors that act on that data. Everything else, like interfaces and classes, are implementation details and not part of the definition.

JavaScript's has a view of an object that stores properties (data) and functions (behaviors) in a hash data structure (which we will call a *property map*).

Property maps consist of object properties are stored as pairs of property names and property values. Since new properties can be added or deleted from the map, the map is does not have a static definition (e.g. it has no class definition). A JavaScript object is just a set of properties and values.

This section will cover how to create, manipulate, and access objects. It will first cover how objects are property maps. It will then show how JavaScript Object Notation (JSON) is a natural way to serialize and store objects externally to the current program.

[33] Not to make too fine a point here, but this is true in all things in life. We all have prejudices and biases, all of them based on some facts and lots of unexamined metaphors of life we have created. Part of education, and of creating an educated populace, is to teach people to examine their assumptions about their metaphors.

Chapter 5.2. 1 Simple Objects

Objects in JavaScript are prototype-based property maps. This is the central point of JavaScript objects, and everything else is just syntax. The syntax for a simple object just lists the property/value pairs separated by a ":" within a block defined by curly braces (`{}`).

```
{
  property1 : "string";          // String value
  property2 : 7;                 // number
  behavior1 : function() { ... };  // function
}
```

The following example shows the creation and printing of a map object in JavaScript.

```
<html>
  <head>
    <title>Object Example </title>
  </head>

  <body>
    <script>
     Let map = {
        title : "MyMap",
        resize: false,
        recenter: true,
        print: function() {
              console.log("title = " + this.title);
              console.log("resize = " + this.resize);
              console.log("recenter = " + this.recenter);
        }
      }

      map.print();
    </script>
  </body>
</html>
```

Program 83 - Implementation of a simple Map object.

In this example, a map object is created that has properties: a title which is a string; resize and recenter Boolean variables; and a print variable that is set to a lambda function.

While this might look similar to a class definition, it would be wrong to think of this map object as an instance of a class. The map object that is defined is not a template, but an instance variable (a value in the program). Remember that an object in JavaScript has no structure or template definition (class). Another variable based on this map definition as a template cannot be created.

Note also the map object can be changed by adding and deleting properties (including functions). Functions and data are associated with the values of the variable. This is shown in the following example, where the variable center is added to the map object after it has been created. When the new property is added, functions needed to be added or changed for the new property are also set.

```
<html>
  <head>
    <title>Object Example </title>
  </head>

  <body>
    <script>
      let map = {
        title : "MyMap",
        resize: false,
        recenter: true,
        print: function() {
          console.log("title = " + this.title);
          console.log("resize = " + this.resize);
          console.log("recenter = " + this.recenter);
        }
      }

      // Add a center point to a map
      map.center = [-77, 39];

      // Change the print function to print the center point.
      map.print = function() {
         console.log("title = " + this.title);
         console.log("resize = " + this.resize);
         console.log("recenter = " + this.recenter);
         console.log("center = " + this.center);
       }

      // Add a new method to get the center of the map
      map.getCenter = function() {
        return center;
      }

      map.print();
      console.log(map.center);
    </script>
  </body>
</html>
```

Program 84 - Adding a center point to the simple map object

Chapter 5.2. 2 Objects are Property Maps

The following example is given to emphasize the point that objects are property maps, an example is provided to show that in JavaScript dereferencing an object and looking up a property in a property map are the same thing. The following program shows this equivalence. Here the dereference operator *object.propertyname* is used interchangeably with *array* processing using the *object[propertyname]*, which accesses the variable from the property map using a key.

Note that it is code like this that is impossible to understand if the reader insists on using their understanding of statically typed OOP to understand JavaScript.

```
<html>
  <head>
    <title>Object Example </title>
  </head>

  <body>
    <script>
      let map = {
        title : "MyMap",
        resize: false,
        recenter: true,
        print: function() {
          console.log("title = " + this.title);
          console.log("resize = " + this.resize);
          console.log("recenter = " + this.recenter);
        }
      }

      // Add a center point to a map
      map.center = [-77, 39];

      // Print the variable via a dereference operator
      console.log(map.title);
      console.log(map.resize);
      console.log(map.recenter);
      console.log(map.center);

      for (let i in map) {
        console.log("property " + i + " is " + map[i]);
      }

    </script>
  </body>
</html>
```

Program 85 - Printing an object out as a hash

Chapter 5.2. 3 JavaScript Object Notation (JSON)

Serialization of an object is just transforming an object into a format that allows it to be represented in a format external to the program. In JavaScript this external format is a collection of named primitives (or variable: value pairs), stored in such a way as it looks like a JavaScript Object definition. This serialized format can then be used to reconstruct a semantically equivalent object spatially (to be used in another program, for example by sending the object over the network) or temporally (at a different time, for example by saving the object to a file).

JSON is the most common notation for serializing JavaScript objects to be used externally from the JavaScript program. Over time JSON has gain in popularity and is now also a popular format for serializing objects in languages other than JavaScript.

JSON syntax is very similar to how an object is defined in JavaScript. In fact, Douglas Crockford, who discovered JSON in 2001, wrote:

I discovered JSON. I do not claim to have invented JSON, because it already existed in nature. What I did was I found it, I named it, I described how it was useful.

To serialize a JavaScript object in JSON, objects that contain only variables that are primitives will have syntax with named variables and their primitive values. These objects will look very similar to the objects that have been presented in this text so far. There will be two differences: 1- The property names will be in quotes, and 2 – All functions will be removed from the objects. As an example, the map object that was seen earlier:

```
var map = {
  title : "MyMap",
  resize: false,
  recenter: true,
  print: function() {
    console.log("title = " + this.title);
    console.log("resize = " + this.resize);
    console.log("recenter = " + this.recenter);
  }
)
```

Program 86 - Map object to be written to JSON format

Will be represented in JSON format as:

```
{

  "title":"MyMap",
  "resize":false,
  "recenter":true

}
```

Program 87 - JSON output of Map object

JSON formatted objects are easily created using the `JSON.stringfy` function. The JSON formatted strings can then be used to reconstruct a semantically equivalent object using the `JSON.parse` function. The JSON object above was created and written to the console log using the following program, and then reconstructed as an example of using JSON.

```
<html>
  <head>
    <title>Object Example </title>
  </head>

  <body>
    <script>
      let map = {
        title : "MyMap",
        resize: false,
        recenter: true,
        print: function() {
              console.log("title = " + this.title);
              console.log("resize = " + this.resize);
              console.log("recenter = " + this.recenter);
        }
      }

      console.log(JSON.stringify(map));

    </script>
  </body>
</html>
```

Program 88 - Program to stringify and parse a JSON object

Chapter 5.2. 4 JSON Serialization and Object Composition

The example given above illustrates a fundamental problem with serialization of objects, and that is that most objects are not collections of primitives but can contain other objects. In fact, it is more common for objects to contain other objects or arrays than to contain only primitive values. Object containing other objects is called object composition[34], implies that each object contained in the object graph must be serialized recursively until only primitives exist. The need to serialize all objects until only primitive values are referenced are not unique to JSON and exist in all object serialization mechanisms.

The simplest case of serializing object composition is a tree graph structure[35]. To do serialization of an object graph represented by a tree, each node is walked recursively. If the node has primitives, output them to the JSON file. If the node references another object or array, create a new object or array, and continue to recursively process the nodes.

To illustrate serialization of complex composed objects, consider the following example object tree.

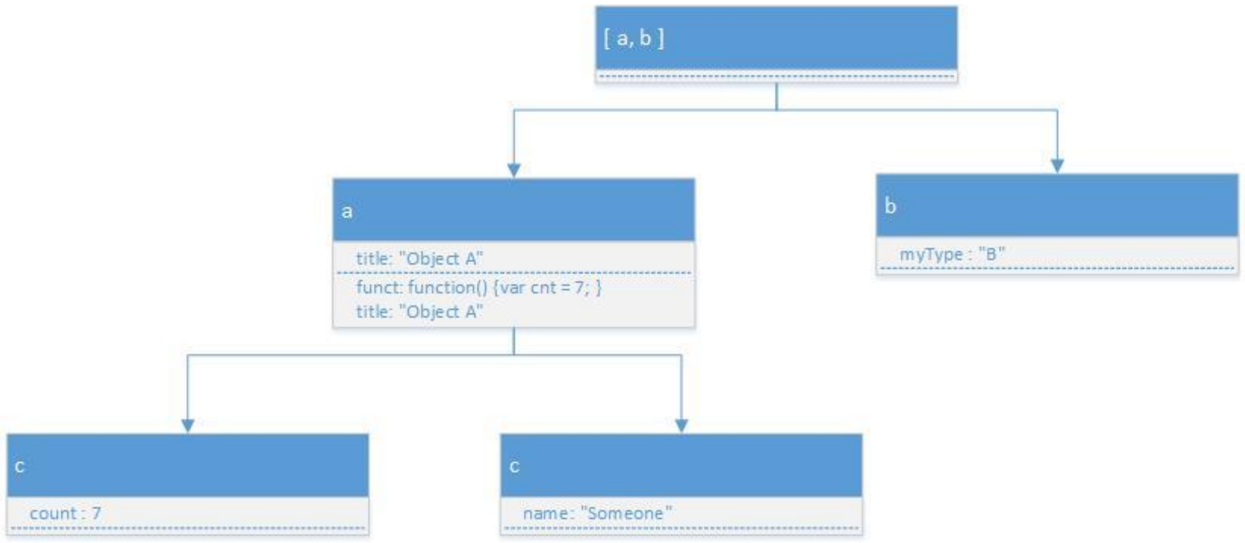


Figure 15 - Composite object to be written to JSON

[34] UML aficionados will argue as to whether to call this aggregation or composition. The distinction is moot here, but in fact this object is a UML composite object, and the term composition will be used in this text.
[35] Object composition means that serialization of objects can result in complex graphs with cycles, multiple references to common objects, and other complex graph structures. There are serialization mechanisms to deal with these problems in JSON, but they are complex and not needed for the material covered in this book.

This program contains an array, named Arr, that consists of two objects, a and b. The first object, a, is composed of a variable title, which is the primate string “Object A”, and a variable func that is a function. Object a also contains two objects, c and d. Object c consists of a primate variable, count, set to the number 7, and object d contains a primitive variable, name, set to the string “Someone”. Object b is composed of a variable “myType”, which contains the string “B”. This would be written in JavaScript as the following:

```
<html>
  <head>
    <title>Serialize </title>
  </head>

  <body>
    <script>
      let c = {
        count : 7,
      }

      let d = {
        name : "Someone"
      }

      let a = {
        title : "Object A",
        ptr1 : c,
        ptr2 : d,
        func : function() {
         letcnt = 7;
        }
      }

      let b = {
        myType: "B"
      }

      let Arr = [a,b];
      console.log(JSON.stringify(Arr));
    </script>
  </body>
</html>
```

Program 89 - Complex object to be serialized to JSON

To serialize an object composed of other objects, each internal object node must be serialized, and that process is repeated until only primitive elements exist. The following is the serialized version of this object, showing how the individual objects are decomposed until the entire object can be represented only as primitives. The tree graph of the object can clearly be seen in this representation of the object.

```
[
  {
    "title":"Object A",
    "ptr1":
    {
      "count":7
    },
    "ptr2":
    {
      "name":"Someone"
    }
  },
  {
    "myType":"B"
  }
]
```

Program 90 - JSON serialization of a complex object.

This JSON definition is sufficient to reconstruct the original array. There are a few points that should be noted here:

1. The variable names (a, b, c, and d) are dropped. Variables are program entities, and not part of the data that make up the object definition. These variables are not needed to rebuild the original object.
2. Arrays and objects must both be serialized, as both are not primitive values. Objects are serialized by enumerating their properties. Arrays are serialized using the JavaScript array syntax, with the array bounded by [], and each member of the array being comma separated.
3. Serialization can result in cycles within the object definitions. These can be quite complex, and will be avoided in this book.
4. As was pointed out earlier, functions are never serialized in JSON format. Unlike method definitions in Java/C#, there is no technical reason that a lambda function (which is data) could not be stored externally. The reason functions are not serialized is that it is inherently unsafe to store program code in an external format since the code could easily be changed, and the program could be made unsafe.

Chapter 5. 4 Constructor Functions and Prototype Objects

The purpose of this section is to continue to build the JavaScript Object Model. It will explain what a Constructor Function is, and how it can be used to create objects. It will show to define Constructor Functions to define how to build generic templates for creating objects, and how to use prototypes and prototype chains to add behavior easily to these object definitions.

Chapter 5.4. 1 Constructor Function and Object Creation

Property maps are a nice way to represent object properties and have many advantages over static class-based definitions of objects. But the way we have defined objects up to this point does not allow for default values, or for behaviors (functions) to be associated with the object without redefining the function for each object. JavaScript addresses these problems by allowing functions (called Constructor Functions) to build the object property maps. The JavaScript `new` operator is then used to set a variable to the property map created in the function.

To understand how this works, consider the following program fragment that creates two Map objects, `mapA` and `mapB`. This is accomplished by setting both variables to different object definitions. The entire definition for this object, including the functions, must be copied.

```
<script>
  let mapA = {
    title    : "MyMap",
    resize   : false,
    recenter : true,
    center   : [-77, 39],

    print: function() {
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
    }
  }

  let mapB = {
    title    : "MyMap1",
    resize   : false,
    recenter : true,
    center   : [-100, 45],

    print: function() {
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
    }
  }

  mapA.print();
  mapB.print();
</script>
```

Program 91 - Creating two objects by setting them to different object literal values

In the code in Program 91 above, the function print has access to a variable `this`, which is just a property map (or object) associate with the object. The use of the this variable will be used to define objects from a common function.

The need to create two object definitions for the same object is inefficient and error prone. In JavaScript the ability to create a *template for an object* and use it to create objects is accomplished using a Constructor Function. A Constructor function is a function that constructs objects. It uses the `this` property map when the function is invoked, and builds the object that is to be used on the `this` object. The JavaScript `new` operator then makes the `this` property map accessible to assign to a variable.

The following program shows the use of a Constructor Function `Map` and the `new` operator to create the equivalent `mapA` and `mapB` variables from Program 86 above. The objects `mapA` and `mapB` are exactly equivalent to the ones from Program 82, including the fact that they each contain their own copy of the `print` function.

```
<script>
  function Map() {
    this.title    = "MyMap",
    this.resize   = false,
    this.recenter = true,
    this.center   = [-77, 39],

    this.print = function() {
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
    }
  }

  mapA = new Map();
  mapB = new Map();
  mapB.title = "MyMap1";
  mapB.center=[-100, 45];

  mapA.print();
  mapB.print();
</script>
```

Program 92 - Using a Map Constructor Function

Note that all of the properties for both maps, including the functions, are stored separately. There are two property maps created, one for each variable reference. These property maps include two separate print functions, as each property map will have a variable referencing a different print function. How to make a single print function will be covered later in this chapter.

At this point, there are probably readers who want to equate a JavaScript Constructor Function with a Java `class`, and to equate the Java and JavaScript `new` operators. This is a completely wrong understanding of the `new` operator in the two languages. In Java a class is a template which includes functions and data, and the `new` operator creates an instance of that template. The Java `new` operator then calls a Java Constructor to initializes the variables in the constructed instance.

To re-emphasize, a JavaScript Constructor Function is not a template for a class that is instantiated. A Constructor Function is a function that is invoked, and when executed assigns properties, including functions, to a property map. The property map is then made available to a variable using the `new` operator. The `new` operator plays no part in constructing the object or how the function properties are called. It is as wrong to equate how a Java and JavaScript object are created as it is to equate how Java and JavaScript define and use objects. Going down that path will in the end only lead to confusion.

Chapter 5.4. 2 Passing parameters to a Constructor Function

The Constructor Function provided in the previous example is not really useful because only the default values for properties can be set when the function is called. To be useful the function needs to have parameters which can be used to set the properties. The simple answer to this problem would be to have a parameter to each value to be set, as in the following example.

```
<script>
  function Map(title, resize, recenter, center) {
    if (title == null)
        this.title    = "MyMap";
    else
      this.title    = title;

    if (resize == null)
      this.resize   = false;
    else
      this.resize   = resize;

    if (recenter == null)
      this.recenter   = true;
    else
     this.recenter    = recenter;

      if (center == null)
      this.center   = [-77, 39];
    else
      this.center = center;

    this.print = function() {
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
    }
  }

  mapA = new Map("Map1", null, null, [55, 20]);
  mapB = new Map(null, true, null, null);

  mapA.print();
  mapB.print();
</script>
```

Program 93 - Passing parameters to a Constructor Function

This solution of adding a parameter for each default value is not a good solution. What happens when a value for a parameter is not defined, implying that the default value should be used? The value is passed as null in the program above, but what if null is a valid value to set to the variable? JavaScript also allows objects to contain values that would not have a default. How are these set?

The solution used here is to pass an object (property map) to the JavaScript Constructor Function, with only the non-default properties in that parameter object. The following illustrates how to implement this solution with the Map Constructor. Note that not only is this code much shorter, to properly handles default values, and allows the object to have values that are not predefined with default values.

```
<script>
  function Map(options) {

    // Set default values
    this.title    = "MyMap";
    this.resize   = false;
    this.recenter   = true;
    this.center   = [-77, 39];

      // Load values from options
    for (let prop in options) {
        // The property has no default value
      if (!this.hasOwnProperty(prop)) {
                        console.log("Property " + prop +
                        " not recognized in Map");
      }
        this[prop] = options[prop];
    }


    this.print = function() {
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
    }
  }

  mapA = new Map({title:"Map1", center: [55,20]});
  mapB = new Map({resize:true});

  mapA.print();
  mapB.print();
</script>
```

Program 94 - Using a Constructor Function to reconstruct an object

This example is the first where the true value and power of JavaScript objects and its object model as shown. But it is just the start of the amazing power of objects in JavaScript.

Chapter 5.4. 3 Constructor Functions and JSON

One nice feature of constructor functions is how nicely they work with JSON. Remember that when creating a JSON object, the functions are not serialized. Only the primitive data items are set in the JSON object. A map object serialized to JSON and then read back into the program will be missing the print function. The issue is how the print function can be added back to the object.

To add the print function back to the object, the JSON object can be passed to the Constructor Function. The object that will be returned from the constructor function add back the methods which were originally part of the object. This is shown below in the Program 95. In this example, a Map object, mapA, is created and serialized into a JSON object. This JSON object is then set to objA, which has all the fields of the mapA object, but is not a Map object and it does not have a print function defined. The objA object is then passed to the Map Constructor function, where the properties of objA are copied into a new mapB object. The print method can then be called on the mapB object.

```
<script>
```

```
  function Map(options) {

    // Set default values
    this.title    = "MyMap";
    this.resize   = false;
    this.recenter   = true;
    this.center   = [-77, 39];

      // Load values from options
    for (let prop in options) {
        // The property has no default value
      if (!this.hasOwnProperty(prop)) {
                        console.log("Property " + prop +
                        " not recognized in Map");
      }
        this[prop] = options[prop];
    }


    this.print = function() {
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
    }
  }

  mapA = new Map({title:"Map1", center: [55,20]});
  mapA.print();    // works

  objA = JSON.parse(JSON.stringify(mapA));
  // objA.print(); // This line fails,here is no print method for ObjA

  objB = new Map(objA)
  objB.print();    // print works, as objB is a Map
</script>
```

Program 95 - Using a Constructor Function to reconstruct a JSON object

This ability to use JSON to store data, and to later pass those JSON objects to the Constructor Functions and reconstruct the original object, will be extensively used later.

Chapter 5.4. 4 Abstracting behavior and prototypes

This chapter so far has emphasized the fact that the functions that are part of an object are copied into the property map of every object. While storing these functions as data with each object allows for exceptions in which specific objects can have behavior that is different for special instances of the object, it is obviously inefficient. What is needed is a way to store the functions once and have them be accessed by all the objects constructed by a single Constructor Function.

This functionality should be implemented in a way consistent with the way the JavaScript language works, and this means using property maps (not class-based templates). A simple solution is to have Constructor Functions have a special property map associated with the function, and then have a link to the Constructor Function's property map stored with each object to share properties across multiple objects. Then when a property, such as a function, is requested for an object, the object's property map is searched for this property. If the property or function is not found in the object's property map, the shared property map for the Constructor Function can then searched. Multiple shared property Maps can exist, and they can each be recursively searched in turn until either the property is found, or the end of the shared property maps is reached. This is shown in the Figure 16.

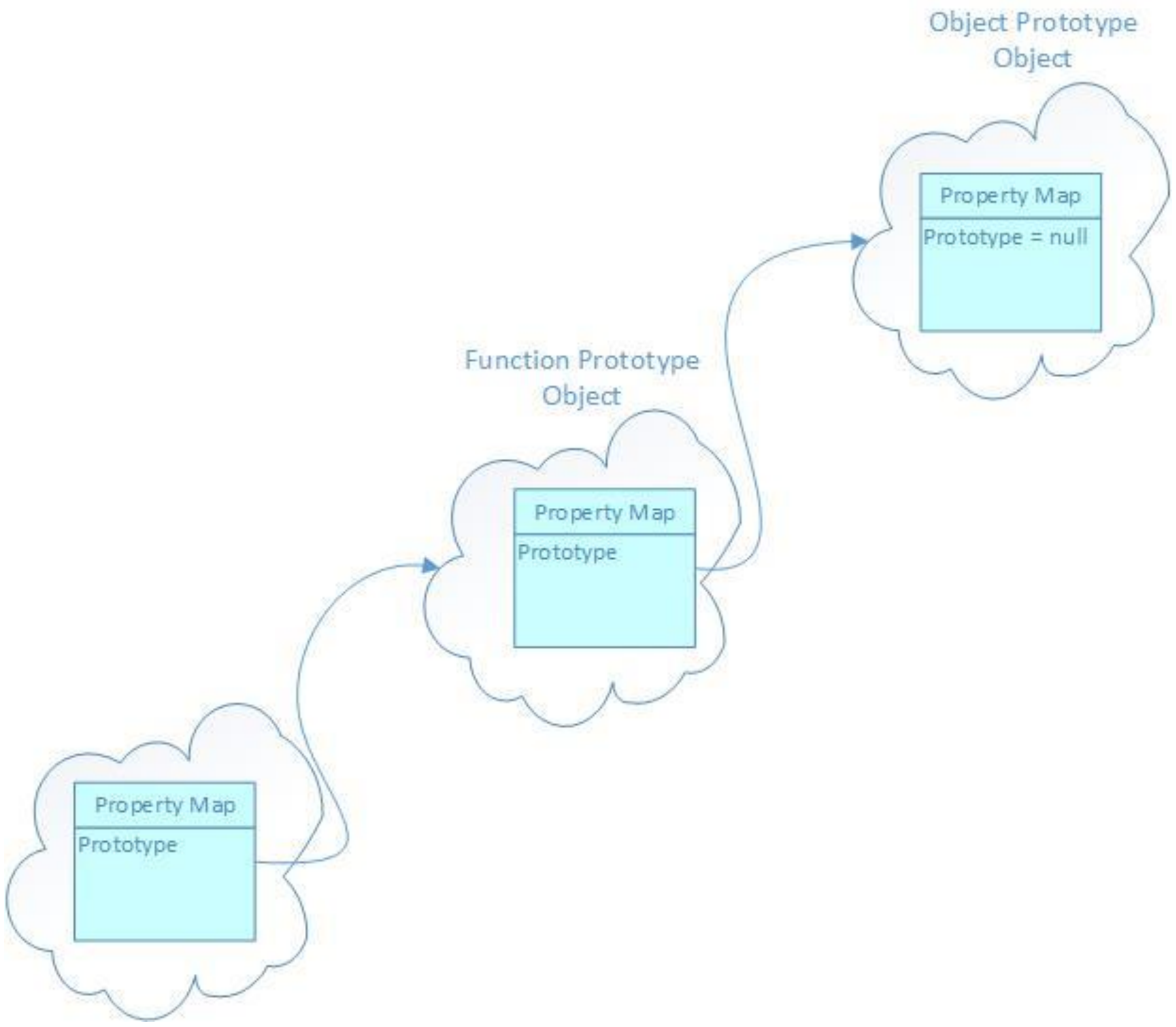


Figure 16 - Recursive search for property in linked property maps

This recursive shared property map concept is implemented in JavaScript and is called *prototypes*.

Every function, not just Constructor functions, has associated with it a prototype object (property map).[36] To see how this prototype object is used, the construction of the JavaScript object with the new operator needs to be understood.

When the JavaScript `new` operator is invoked, it does two things.

1. First it sets the variable on the left-hand-side (lhs) of the equal sign to the `this` variable that was used in the Constructor Function.
2. Second a prototype variable (a link) in the object's property map is set to point to the prototype object property map corresponding to the Constructor Function. The value of this link cannot be directly changed by the object but is accessed when a property is requested for the object that cannot be found in the objects property map.

The behavior of recursively searching property maps, described earlier, can now be implemented. When a function is requested, the property map for the individual object is searched. If the function is not found, the search continues in the prototype map of the Constructor Function. If it is still not found, the search continues until either the function is found, or the root of the tree, the Object prototype, is reached.

In the following example, the print method is moved from the individual objects, and stored in the Map protocol object. Now the two objects, `MapA` and `MapB`, both share the same print function.

```
<script>
  function Map(options) {

    // Set default values
    this.title    = "MyMap";
    this.resize   = false;
    this.recenter   = true;
    this.center   = [-77, 39];

      // Load values from options
    for (let prop in options) {
        // The property has no default value
      if (!this.hasOwnProperty(prop)) {
                        console.log("Property " + prop +
                        " not recognized in Map");
      }
        this[prop] = options[prop];
    }
  }

  Map.prototype.print = function() {
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
  }
```

[36] Unless a function is a Constructor Function, this prototype object is not used, but the function will still have prototype objects associated with them.

```
  mapA = new Map({title:"Map1", center: [55,20]});
  mapB = new Map({title:"Map2"})

  // These two objects share the print method from the
  // Map prototype object.
  mapA.print();
  mapB.print();
</script>
```

Program 96 - Accessing the print function in the Map prototype object

Chapter 5.4. 5 Inheritance and Polymorphism

As I wrote the title to this section, I could just see the eyes rolling of a number of readers. Inheritance and polymorphism are always a problem in a class in Java/C#/C++/etc. Let me put everyone's mind at ease. In JavaScript, if you understood prototypes from the last section, you already know these concepts in JavaScript.

To see how polymorphism can be used in JavaScript, consider the following problem. You want the object mapB from Program 96 to have a different print method than the standard one that is defined for the Map Constructor Function. This is a trivial change, as all that is needed is to add a different print function to the mapB property map, and that function will be encountered first when looking for the print function for the mapB object. This is shown in the example Program 97.

```
<script>
  function Map(options) {

    // Set default values
    this.title    = "MyMap";
    this.resize   = false;
    this.recenter   = true;
    this.center   = [-77, 39];

      // Load values from options
    for (let prop in options) {
        // The property has no default value
      if (!this.hasOwnProperty(prop)) {
                          console.log("Property " + prop +
                          " not recognized in Map");
      }
        this[prop] = options[prop];
    }
  }

  Map.prototype.print = function() {
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
  }

  mapA = new Map({title:"Map1", center: [55,20]});
  mapB = new Map({title:"Map2"});
  mapB.print = function() {
      console.log("Same print function, but polymorphic")
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
  }

  // These two objects share the print method from the
  // Map prototype object.
  mapA.print();
  mapB.print();
</script>
```

Program 97 – Polymorphism in JavaScript

This changes the meaning of *inherit from* as used in Java and other class-based OOP languages. When discussing prototypes and JavaScript, the term *inherit from* means that the property map for the current object links to (inherits from) the prototype object's property map, and the property maps are searched recursively for a property. This is, in a sense, how polymorphism is implemented in class-based languages, but it is so abstracted away that any sense of what is going on is lost to most programmers.

To summarize, polymorphism, or calling different methods which have the same name, in JavaScript means that the property maps are searched until the first occurrence of that property is found.

What is missing in this section is how to implement inheritance and composition delegation in JavaScript. There is a good reason for that. First, while the concept of inheritance is simple in JavaScript, changing the property map links is non-trivial; and until someone shows me a valid design using inheritance that cannot be implemented more easily and correctly using some other design mechanism, my advice is to not use inheritance. Thus, I never cover inheritance (other than interface inheritance, which is not the same as class inheritance) in any language.

As for compositional delegation, this is normally solved in JavaScript using Functional Programming. While I might be more inclined to describe how to implement delegation, as it is not that difficult, until a see a real-life case where it is an advantage over Functional Programming in JavaScript, I see no need to cover something that is at best useful in one-off situations.

Chapter 5.2. 1 JSON and prototype properties

What happens to the protocol chain when an object is serialized to JSON? When serializing an object, only the properties that can be externally represented are written to the JSON object. For the prototype variable (link to a Constructor Function prototype object) there is no way to know if the corresponding Constructor Function will exist when this JSON object is loaded into the new environment, so it must be dropped from the JSON definition.

But dropping the prototype link variable from the JSON object does not represent a problem. If the Constructor Function for the object is known, the appropriate Constructor Function can be called passing in the JSON object to reconstruct the original object, just as was done in Program 98. This is shown in the following program to reconstruct the prototype chain for Map objects[37].

```
<script>
  function Map(options) {

    // Set default values
    this.title    = "MyMap";
    this.resize   = false;
    this.recenter   = true;
    this.center   = [-77, 39];

    // Load values from options
    for (let prop in options) {
        // The property has no default value
        if (!this.hasOwnProperty(prop)) {
           console.log("Property " + prop +  " not recognized in Map");
        }
        this[prop] = options[prop];
    }
  }

  Map.prototype.print = function() {
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
  }

  mapA = new Map({title:"Map1", center: [55,20]});

  jmapA = JSON.stringify(mapA);
  mapA1 = new Map(JSON.parse(jmapA));
  mapA1.print();
</script>
```

Program 98 - Reconstructing the protocol chain for a JavaScript object

Chapter 5.2. 2 Finding the JavaScript Constructor Function in the DOM

Calling the correct Constructor Function to reconstruct the object works fine so long as the correct Constructor Function is known for the JSON object. But what if a number of different functions, created with different Constructor Functions, are stored externally, and the program does not know what Constructor Function corresponds to each JSON object?

[37] The protocol chain for a JSON object can be updated more simply by using the JavaScript setPrototypeOf function. Use of this function is, however, strongly discouraged. The setPrototypeOf function must change many structures in the program to optimize how prototype chains are optimized, and is a very expensive function to call. Also, any default code that is normally executed when constructing an object is not run. For these reasons, it is recommended that a new variable with the proper prototype constructor be created. This text will recommend that to create a new object, a correct Constructor Function be written that sets all default property values and saves all property values of the original object be created, and that Constructor Function be called using the JSON object.

Fortunately, there is an answer for this in JavaScript. All Constructor Functions are stored as lambda values in the DOM using the `window` property map. To retrieve and execute the `Map` Constructor Function using the object `JSONObject` can be done in the following line of code:

```
var myMap = new window[“Map”](JSONObject);
```

All that is needed to recreate the JSON object is the name of the Constructor Function, and that can be stored as a property in the JSON object itself. A strange looking name should be used so that it will not overlap with names the programmers might choose, so we will use the name “`__cfName`”. The new definition of the Map Constructor Function will now be the following:

```
<script>
 function Map(options) {

    // Set default values
    this.__cfName = "Map";
    this.title    = "MyMap";
    this.resize   = false;
    this.recenter   = true;
    this.center   = [-77, 39];

      // Load values from options
    for (let prop in options) {
        // The property has no default value
      if (!this.hasOwnProperty(prop)) {
                        console.log("Property " + prop +
                        " not recognized in Map");
      }
        this[prop] = options[prop];
    }
  }

  Map.prototype.print = function() {
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
  }

</script>
```

Program 99 - Map Constructor Function setting the __cfName variable

Now that the name of the Constructor Function to call is known, the following function, getObjectFromJ can reconstruct any JSON object using the correct Constructor Function.

```
// The input parameter is the JSON object
function getObjectFromJSON(string) {
  let parsedObject = JSON.parse(string);
  let cf = parsedObject["__cfName"];
  return new window[cf](parsedObject);
}
```

Program 100 - getObjectFromJSON function

The following example shows how to use the getObject function on a JSON object that represents a Map.

```
<script>
  function Map(options) {

    // Set default values
    this.__cfName = "Map";
    this.title    = "MyMap";
    this.resize   = false;
    this.recenter   = true;
    this.center   = [-77, 39];

      // Load values from options
    for (let prop in options) {
        // The property has no default value
      if (!this.hasOwnProperty(prop)) {
                        console.log("Property " + prop +
                        " not recognized in Map");
      }
        this[prop] = options[prop];
    }
  }

  Map.prototype.print = function() {
      console.log("title = " + this.title);
      console.log("resize = " + this.resize);
      console.log("recenter = " + this.recenter);
      console.log("center = " + this.center);
  }

  // The input parameter is the JSON object
  function getObjectFromJSON(string) {
    let parsedObject = JSON.parse(string);
    let cf = parsedObject["__cfName"];
    return new window[cf](parsedObject);
  }

  mapA = new Map({title:"Map1", center: [55,20]});

  // Create the JSON object, and then pass it to getObjectFromJSON
  // to show that the function does indeed reconstruct the object.
  jmapA = JSON.stringify(mapA);
  mapA1 = getObjectFromJSON(jmapA);
  mapA1.print();
</script>
```

Program 101 - Using the getObjectFromJSON function to reconstruct the object

Chapter 5. 5 Scoping in JavaScript

Variables in JavaScript can have 3 types of scoping; block, function (or local), and global. These 3 types of scoping will be explained briefly here, mainly in preparation for the next section on closure. While it seems like having only 3 types of scope would make the concept of scope easy to understand, it always seems to be one of the most difficult concepts to get across to programmers.

There are some parts of ECMA6+ that are very useful, and one of them is the inclusion of the let keyword. The let keyword has allowed block scoping in JavaScript but has muddied the waters and caused additional difficulties in programs that combine the use of the `let` and the older `var` keyword. This text will avoid those problems by always using the let keyword.
While a JavaScript programmer should probably understand the var keyword, the only reason is to understand how it impacts older programs that use it. When it is used, it can cause some variables that would be block scoped using the let keyword to be function scoped. If a reader gets into a situation where they are supporting code that uses the var keyword, there are any number of good sites that explain what it means. This will not be a problem for readers who have understood the 3 types of scoping described here.

Chapter 5.4. 1 Undeclared variables

First, JavaScript does not require that variables be declared. Variables that are not declared are created and put in the global scope. This allows for all sorts of problems, such as misspelling of variables, or variables meant to be in other scopes being put in global scope.

To avoid accidently spelling variables wrong or to prevent programmers from using undeclared variables, the JavaScript program can use the "`use strict`" directive in their programs. This will cause the program to fail with error if an undeclared variable is encountered. For this book, the `use strict` directive will be used, and all variables must be declared. Undeclared variables simply should not exist. The issue of undeclared variables is therefore dispensed with, and not covered further.

Chapter 5.4. 2 The let keyword

The `let` keyword declares a variable, and scopes that variable to the next larger program block, where the next larger program block is a unit of a program that is contained between two curly braces (`{}`). A variable declared inside a block will be scoped to the block represented by those curly braces. If the variable is outside of any curly braces, it will have global scope.

The first type of scoping that will be covered is block scoping. Block scoping allows a variable to exist only in the block in which it is declared. This is often useful for variables that should only exist in a block, for example a for loop. The following for loop shows a good use of the let variable for block scoping:

```
function myFunc() {
    for (let count = 0; count < 10; count++) {
        Do something
    }
}
```

Here the next larger program block is the for loop, so the variable count will only exist in the for loop and cannot be used outside of that loop. The variable will come into existence when the loop block is entered and will be destroyed when the loop is exited. This use of scoping is what most programmers are used to seeing.

The next type of scoping is function scope, which is also called local scope in JavaScript. Function scope occurs when a variable is declared inside of a program block that is a function. The following program shows the loop above, but now the variable count has function scope.

```
function func() {
    let count = 0;
    for (count = 0; count < 10; count++) {
        Do something
    }
}
```

The difference between the block scope and function scope code would be basically meaningless in most programming languages such as C#/Java/C++ because function variables only exist while the function exists (e.g. the function is on the program stack). In Java/C#/C++, the function is pushed on the stack when the function is entered and popped when it is exited.

Function scope in JavaScript is very different, and function variables are not released when a function exits. This will be covered in more detail in the section on Closure.

The final type of scoping in JavaScript is global scoping. A variable declared outside of any block of code defined by curly braces (`{}`) is globally scoped. There is one copy of a global variable in the entire program, and for the purposes of this discussion, that copy comes into existence when the program starts running and exists the entire time the program runs. This is similar to a Java/C# static variable. The following shows how the loop above would look if a global variable was used. Note that in this case, the variable must be reset each time the function is entered, as the previous value from the loop is still stored in the variable.

```
let count = 0;
function func() {
    count = 0;
    for (count = 0; count < 10; count++) {
        Do something
    }
}
```

The take away from this section is that scoping controls where a variable is accessible and the time the variable definition is maintained. Block variables and function variables are only visible in the block/function they are declared in. Block variables only exist so long as the program is executing in the block they are declared in. Function variables exist for a much longer period, but that will be covered in the section on closures. Global variables are always accessible, and always exist.

The only strange idea coming from this section is how long function variables live, as unlike other languages, they do not go away when the function exits. This is the subject of the next section on closures.

Chapter 5.3. 1 Closures

One interesting aspect of JavaScript that is not found in most languages the reader will be familiar with (e.g. Java, C++, C#, etcetera) is that a function (an outer function) can contain other functions (inner functions). This program structure gives rise to an interesting problem with variables known as *closure*. The problem starts with the fact that variables scoped in the outer function are also scoped in the inner functions, as shown in Program 96 below.

```
<script>
  function f1() {
    let var1 = "function scope"

      function f2() {
          console.log(var1);
      }

      f2();
  }

  f1();
</script>
```

Program 102 - Calling an inner function from an outer function.

This example by itself seems very natural and does not seem to be a problem. The issue arises when the inner function is used as a lambda value. The lambda function can now be called after the outer function has completed, causing the variable scoped in the outer function to be referenced after the outer function has completed running. This is shown in the following program.

```
<script>
  let obj1 = new function f1() {
    console.log("Entering f1");
    let var1 = "function scope"

      this.f2 = function() {
          console.log(var1);
      }

      console.log("Leaving f1");
  }

  obj1.f2();
</script>
```

Program 103 - Running an inner function after an outer function has completed.

This program illustrates the closure problem. The inner function, `f2`, is associated as a lambda value property in the obj1 object. This lambda function exists even after the function f1 has completed and exited. In Java/C#/C++/etcetera, the variables of the outer function are allocated on the program stack and cease to exist after the outer function has completed running and so in Java the variable `var1` would not be available when `f2` is called using the reference in `obj1`. The nature of JavaScript requires, however, that the function local variables outlive the execution of the function, just for this type of situation where inner functions can exist (as lambda values) after the function has executed. This is a JavaScript closure. Closure is not hard because it is a difficult concept, instead it is hard because programmers generally have not seen it before, and it is often explained in terms of its impacts, and how they are different from Procedural and class-based OOP languages.

The concept of a closure is simple, but the implications are huge, and the concepts are not easily understood in terms of class-based OOP or procedural languages[38]. The need to make function variables outlive the function execution is the crux of the reason why closures are included in JavaScript. JavaScript Closures are really not that complicated, especially when used for their purpose. Once again, problems arise when trying to use an invalid metaphor to understand a concept, resulting in the myriad of *helpful* web pages that attempt to ferret out how to understand a closure using the wrong metaphor.

This concept of closure will be looked at in more detail later in the chapter when an encapsulated object model in JavaScript is describe.

Chapter 5. 6 A simple JavaScript OOP Model

This section will combine all the material covered in this chapter up to this point to create an OOP model to be used throughout the rest of this text. Two object models will be given. The first is the one normally used in JavaScript, so it will be the one used through the rest of the book. It does not enforce encapsulation or information hiding, but languages like TypeScript have been introduced since ECMA6 to provide encapsulation and information hiding, and they generally compile to JavaScript that follows the basic pattern of an object presented in this first object model.

A second model of OOP will be presented to show how JavaScript closure can be used to enforce encapsulation and information hiding in basic JavaScript. This would be a good implementation of OOP in JavaScript if encapsulation and information hiding are needed, but it has been overtaken by events, and most programmers using these features will be using languages such as TypeScript. It is presented because it highlights and explains a lot of features of JavaScript programming.

Chapter 5.6.1 A first JavaScript Object Model

In Chapter 5.3.4 a method to combine prototypes and constructor functions into an object was shown, though it was not given as an object model at the time. The idea is that objects are constructed in a Constructor Function, and the functions that operate on that object are stored in the Constructor Function prototype object. The strategy for creating this type of object is as follows:

[38] There are analogs to this behavior of requiring a method local variable to outlive the method it is declared C# and Java. Closure analogs provide much of the justification for anonymous inner classes in Java, final local variables when using Java Events, and the new inclusion of lambda functions. The basic problem is the same in all of these languages, however Java used anonymous inner classes, and the syntactic sugar of Java lambda functions, to handle these problems. Understanding why the problems occur in JavaScript helps understand why they occur in Java, but the solutions are completely different.

1. The Constructor Function is defined, and it will take one argument, which is an options object of properties to be included and set in this object.
2. The __`cfName` variable is assigned to the name of the Constructor Function in the `this` object. This is so that object can later be reconstructed if it loses its constructor (e.g. it is written and then read from a JSON data source).
3. The property default values are defined in the Constructor Function for the `this` object.
4. All properties passed into the Constructor Function will be moved to properties in the newly constructed `this` object. The properties will include all properties in the options object, including those that do not have default values. The reason for this will be explained in the section on Unstructured Data later in this chapter.
5. All methods that will act on this object will be included in the prototype object for this Constructor Function. These prototype functions will be defined after the Constructor Function has finished executing. The prototype function will be called with an object, so the `this` variable can be used to access the appropriate object.

An example of implementing and using this definition for a Map object is presented below.

```
<script>
function Map(options) {
  // Set Defaults
    __cfName = “Map”;
    this.title = "Please change the title";
    this.recenter = true;
    this.center = [-77, 32];

  // Get properties from parameter
  outerloop:
  for (i in options) {
    for (j in this)     {
      if (i == j) {
          this[i]= options[j];
          continue outerloop;
      }
    }
    console.log("Property " + i + " is not a default Map property");
    this[i] = options[i];
  }

// Set accessor function
Map.prototype.setTitle = function (title) {
  this.title = title;
}
Map.prototype.getTitle = function () {
  return this.title;
}

Map.prototype.setRecenter = function (recenter) {
  this.recenter = recenter;
}
Map.prototype.getRecenter = function () {
  return this.recenter;
}
```

```
Map.prototype.setCenter = function (center) {
  this.center = center;
}
Map.prototype.getCenter = function () {
  return this.center;
}

// Set toString function
Map.prototype.toString = function() {
  return ("title: "     + this.title
        + " recenter: " + this.recenter
            + " center: "   + this.center);
}

var m1 = new Map({title: "newMap"})
console.log(m1.getTitle());

m1.title = "This breaks encapsulation";
console.log(m1.getTitle());

</script>
```

Program 104 - Map object definition

The only issue with this definition of an object is that encapsulation and information hiding is broken. As is shown in the example above, where the programmer directly sets the value of `m1` by saying `m1.title = "This breaks encapsulation";`, the programmer can directly access the title property without using the `setTitle` and `getTitle` methods.

The following object model adds encapsulation using function variables and closures. It is included to show how closures could be used in JavaScript, but is not a standard object model, and should not be used for production programs.

Chapter 5. 7 A JavaScript object model that includes encapsulation and data hiding

The problem with the object model in the previous section is that the constructed object was returned to the main program where all of its properties were visible. This is why the principal of encapsulation is broken. The easiest way to prevent the leaking of the object is to make the object be defined not by this, but by a function variable. Functions variables are only scoped as visible in the function itself, and this in effect makes the object itself private.

To implement this new object model, the strategy from the last section needs to be tweaked a little. First, the Constructor Function will not construct a `this` object, but will create a function local variable and construct the object inside of this local variable. Now that the object is a local variable that cannot be accessed outside of the Constructor Function (which is the outer function), inner functions need to be defined that can access the properties of the object.

Note that the object will still want to use prototype functions so that only one instance of the function needs to be defined, but these prototype functions must now be moved so they are inner functions of the Constructor function. This gives the prototype functions access to the variables defined in the outer function.

The strategy for creating this type of object is as follows:

1. The Constructor Function is defined, and it will take one argument, which is an options object of properties to be included and set in this object.
2. A function local variable, named `__properties`, is defined of type object.
3. The `__cfName` variable is assigned to the name of the Constructor Function in the __properties object. This is so that object can later be reconstructed if it loses its constructor (e.g. it is written and then read from a JSON data source).
4. The property default values will be defined in the `__properties` object.
5. All properties passed into the Constructor Function will be moved to properties in the newly constructed `__properties` object. The properties will include all properties in the options object, including those that do not have default values. The reason for this will be explained in the section on Unstructured Data later in this chapter.
6. All methods that will act on this object will be included in the prototype object for this Constructor Function. These prototype functions will be defined inside of the Constructor Function, so they have access to the function local variable.
7. A *stringify* method needs to be defined to allow the internal object (the __properties) object can be accessed and converted into a string.

An example of implementing and using this definition for a Map object is presented below.

```
<script>
function Map(options) {
  // Set Defaults
    let __properties = new Object;
    __properties.title = "Please change the title"
    __properties.recenter = true;
    __properties.center = [-77, 32];
      __properties.__cfName = "Map"

 // Get properties from parameter
   outerloop:
  for (i in options) {
    for (j in __properties)   {
      if (i == j) {
          __properties[i]= options[j];
          continue outerloop;
      }
    }
    console.log("Property " + i + " is not a default Map property");
    __properties[i] = options[i];
  }

 // Set accessor function
  Map.prototype.setTitle = function (title) {
    __properties.title = title;
  }
  Map.prototype.getTitle = function () {
    return __properties.title;
  }
  Map.prototype.setRecenter = function (recenter) {
    __properties.recenter = recenter;
  }
  Map.prototype.getRecenter = function () {
    return __properties.recenter;
  }

  Map.prototype.setCenter = function (center) {
    __properties.center = center;
  }
  Map.prototype.getCenter = function () {
    return __properties.center;
  }

  Map.prototype.stringify = function() {
      return JSON.stringify(__properties);
  }

  Map.prototype.toString = function() {
    return ("title: "     + __properties.title
          + " recenter: " + __properties.recenter
          + " center: "   + __properties.center);
  }
} //End of Constructor Function

function getObjectFromJSON(string) {
  let parsedObject = JSON.parse(string);
  let cf = parsedObject["__cfName"];
  return new window[cf](parsedObject);
}

let m1 = new Map({title: "newMap"})
let m2 = getObjectFromJSON(m1.stringify());
console.log(m2.toString());

</script>
```

Program 105 - Object example with encapsulation and data hiding

Chapter 5. 8 Unstructured Data

Before leaving the topic of objects, there is a concept that just doesn't seem to make sense to most programmers, and that is unstructured data. This is probably because most of the CS industry is built around structure data[39]. Data in a relational database is structured into tuples and relations. Data in OOP is built into classes and instances. When unstructured data is encountered, programmers go to great lengths to build work arounds, none of which seem to work well[40]. The author has seen many of these problems and has yet to see a good solution in a relational database.

[39] For more information about unstructured data, see information about NoSQL database, such as https://www.youtube.com/watch?v=qI_g07C_Q5I
[40] http://blogs.tedneward.com/post/the-vietnam-of-computer-science/

To understand unstructured data, an example will be given that occurs when working with maps. On a map there can be placed various features, which can be thought of as markers on a google map. These features have standard properties, such as position, title, marker style, and description, and these properties can be stored in a structured format as a tuple (record) in a relational database.

Now consider a real application, for example a map of the monuments on the Gettysburg Battlefield (http://chuckkann.com/MonumentsMap/Monuments.html). In this map various features are placed on the map. These features represent many different types of monuments. Some features are regimental monuments and may/may not be associated with states. Some features are monuments to people who fought at Gettysburg, and could are associated with a person, but could also be associated with a state or regiment. Some monuments are buildings, and could be associated with person, or a regiment, or a part of the battlefield.

To make the situation even harder, the application should not define what are valid associations when the application is written. If a new association type, such as casualty percentage, is to be added to some of the features in the future, this data should be stored and easily searched.

While this type of design might not be a common business problem, it is not uncommon. Probably the reason it is not more common is because programmers are always inventing work arounds to handle the problems, and most work arounds using relational databases are poor hacks.

However, using unstructured data (e.g. a JSON type format), this problem of unstructured information and relationships does not exist. Because all objects are just property maps, it is possible to add attributes of any type to any object.

One important aspect of learning JavaScript and its object definition is that it often requires the program to look at object-oriented design in a completely new and different manner, increasing the tools they have to solve problems.

Chapter 5. 9 Conclusion

I hope that after having read this chapter, the reader is convinced that JavaScript is not Java/C# or any other class-based OOP language they have used. I hope that they have either decided it is too weird to ever make sense using the metaphors and models they have used in the past with these languages, and started to understand JavaScript for what it is, not what it is not.

Chapter 5. 10 Exercises

1. Explain why it is a very bad idea to serialize functions in JSON and store them to an external file or send them to another computer.
2. What is the eval() function in JavaScript? What do you think is the original of the JavaScript saying "eval is evil".
3. Explain why an object built with encapsulation but the ability to change the methods accessing the closure variable is safe.
4. What is the difference between using the new operator and operator.create?

What you will learn

In this chapter, you will learn:

1. What a CRUD interface is
2. How to create a user interface for an application
3. How to create the object(s) needed for an application from a UI
4. Mapping the UI into program behavior

Chapter 6 CRUD, Objects, and Events

This chapter will bring together all of the components needed to design and develop a web based Create-Read-Update-Delete (CRUD) application for the web. This application will use local storage to store the data from the application, but the application will be developed in a way that will allow it to later be easily adapted to take advantage of storing the data on a server using web services.

Chapter 6.1 CRUD Interface

CRUD is an acronym for the most basic type of user interface. Most interfaces use this pattern, though it might be hard to see sometimes. Consider an online store that has an interface for purchasing an item. The interface allows users to:

1. create a record, for example to purchase an item from an online store,
2. read the record, or get the details of the purchase
3. update, or change the details of the purchase
4. delete the purchase completely.

The store then continues the pattern, allowing a user to create a record for a purchase order, consisting of many items to be purchased. This purchase order interface again has a CRUD interface, though purchase order CRUD interface manipulates purchases of individual items.

The example in this chapter is much simpler than the example application for the store. It will simply create a set of map records that are stored in an array. The CRUD application will add/update/delete the records from the array. The application will use a version of the map application that the books has been developing.

This map application will use OOP to collect and stored is as an object. The object will be implemented as shown in Chapter 5.6.1.

Each of the CRUD operations will be implemented using events, which in this book will be called Event Based Programming (EBP). EBP will seem very different and strange to readers familiar with only Procedural Programming, where all program actions emanate from a main method. In EBP, all functions are independent and are run as a result of an event occurring.

To help in understanding the system design and implementation, this chapter will be structured as follows:

1. An initial design of the system (a mockup of how the system should generally look and act) will be created.
2. The HTML and CSS programs for this mockup will be implemented, separately from the functionality that will be included in the system. The design look-and-feel of the system will be kept separate from the functionality throughout the implementation, and in the real world represent two completely different skill sets.
3. The forms in the application will be mined for the data needed for the application. The arrays and objects needed to implement this design will be created.
4. The events needed to run the application will be defined, and a brief description of how to implement each event will be given.
5. The code to handle each event will be written.

At the end of these steps, a fully functioning CRUD interface will have been developed.

Chapter 6.1. 1 Overall Application Design

The first step in designing a system is to create some sort of mockup of the system to see how it will work. This type of mock up is called a *wire frame.* The purpose of a wire frame is to set up what the system will look like to define the options that will be available in the system, as well as the data and any data dependencies. The following is a wire frame design of the application which was generated in Pencil.

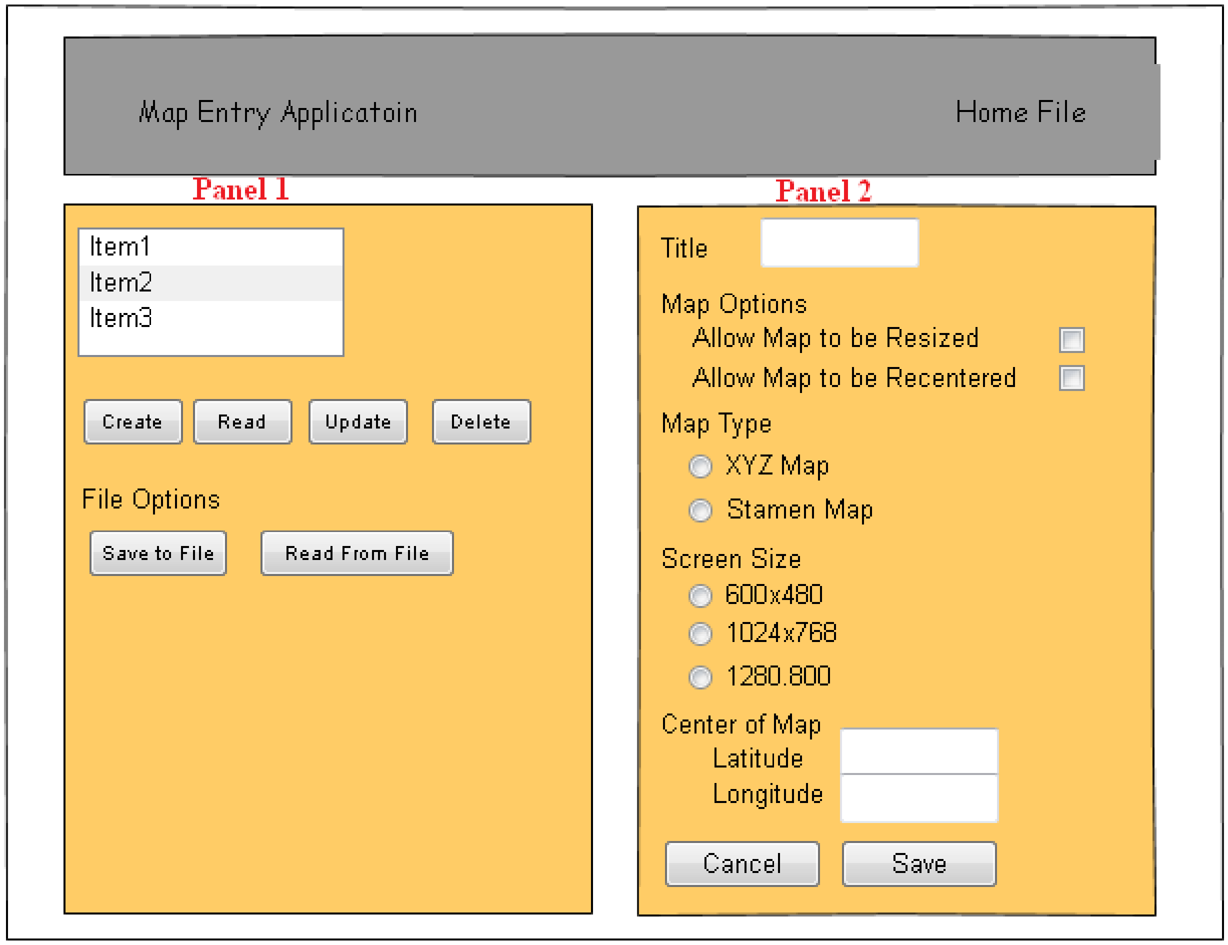


Figure 17 – Wire Frame design for CRUD interface in Pencil

The application will consist of two panels. Panel 1 will always be displayed while the application is running. Panel 1 contains a list box which shows the map features in an array of map records managed in this application. The map records in this application will be a global variable named *records* and should be the only global variable in the program.

Also in Panel 1 are several buttons to define user operations. These operations will generate events which will need to be handled. Details on how to handle the events will be given in a later section, but the following is a general overview of the functionality on Panel 1.

Associated with the list box of map records are the buttons *Create*, *Read*, *Update*, and *Delete*. These buttons will read the current record highlighted in the list box and run the appropriate operation on that map object.

There are a second set of buttons which are program options, they are *Save to File*, and *Read from File*. The save to file button will write the array to some persistent store using a JSON format. The read from file will read a persistent store previously written and parse the JSON to restore a semantically identical copy of the data previously stored in the program.

Panel 2 is a form to manipulate data about the features. Panel 2 should be hidden unless the user has selected the create, read, or update options. If the user is not actively interacting with a specific feature, Panel 2 should not be displayed. When Panel 2 is shown, fields that cannot be written to should be read only. Proper editing of the fields for correct values should be done. The values of the edits and read only setting will be specified in the data section of this chapter.

On Panel 2 there are two buttons. The first is a cancel button, which throws away any edits that have been performed on the data on this panel. The second is the save button, which saves changes made to this form. Note that the save button will be displayed for both the create and update options, though the behavior will be different between those two options.

Chapter 6.1. 2 Creating the CSS and HTML definition

The CSS and HTML files for this application are largely defined in previous chapters of this text, and so are presented here without comment.

```
/*
 *     File:      CRUD.css
 *     Author:    Charles W. Kann
 *     Date:      July 8, 2017
 *
 *     Purpose:   To define the styling for
 *                the CRUD application
 */

/*
     CSS for the header of the page
 */
header {
  margin : 5px 50px 5px 50px;
  border : 2px solid blue;
  background-color : slategray;
  color : white;
}

header p {
  font-size : 150%;
}

.header-icon {
  display : inline-block;
  margin: 50px 10px 50px
}

.header-desc {
  display : inline-block;
  margin : 25px;
}

.header-menu {
  display : inline-block;
  float: right;
  margin : 50px 50px 50px 50px;
 }
```

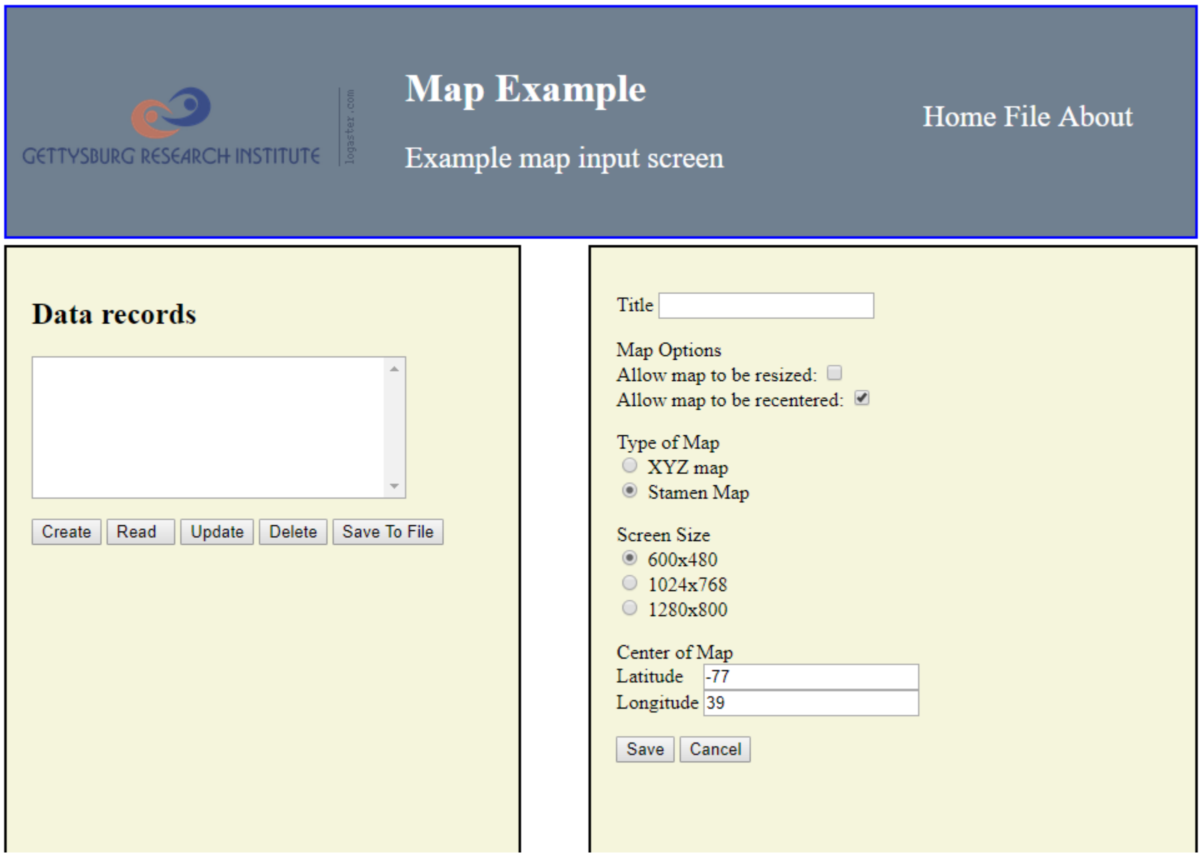
```
/*
     CSS for the input form (Panel 2)
 */
#inputForm {
  margin : 0px 50px 5px 5px;
  background-color : beige;
  border : 2px solid black;
  display: none;
  float : right;
  padding : 20px;
  width : 42%;
  height : 70%;
}

/*
     CSS for the records (Panel 1)
 */
#recordDivision {
  margin : 0px 5px 5px 50px;
  background-color : beige;
  border : 2px solid black;
  display: inline-block;
  float : left;
  padding : 20px;
  width : 35%;
  height : 70%;
}

/*
    Select list for records
 */
#dataRecords {
  width : 300px;
}

/*
    Gray out read-only input
 */
input:-moz-read-only { /* For Firefox */
  background-color: lightgray;
}


input:read-only {
  background-color: lightgray;
}
```

Program 106 – CSS for CRUD interface

```
<!--
   File Name: CRUD.html
   Author:    Charles Kann
   Date:      7/8/2017
   Purpose:   To define the Map Example CRUD application.
   Modification History:
     7/8/2017 - Initial Release
-->

<html>
  <head>
    <meta charset="UTF-8">
    <title>Map Example </title>
    <link rel="stylesheet" type="text/css" href="CRUD.css">
      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.Java
Script">
      </script>
      <script src="Map.JavaScript"> </script>
      <script src="CRUDLibrary.JavaScript"> </script>
      <script src="CRUDOnload.JavaScript"> </script>

      <script>
        // Global records array declaration
        var records = new Array();
      </script>
  </head>

  <body>
    <header>
      <div class="header-icon">
       <image src="GRI_logo.png" />
      </div>
      <div class="header-desc">
        <h1>Map Example</h1>
        <p class="header-p">
          Example map input screen
        </p>
      </div>
      <div class="header-menu">
        <p class="header-p">
          Home    File    About
        </p>
      </div>
    </header>

      <section id="recordDivision" >
          <h1> Data records </h1>

            <select id="dataRecords" size="7" >
            </select>
          <p>
            <input type="button" value="Create" id="create" />
            <input type="button" value="Read  " id="read" />
            <input type="button" value="Update" id="update" />
            <input type="button" value="Delete" id="delete" />
          </p>

          <p>
            <input type="button" value="Save To File" id="saveFile" />
            <input type="button" value="Read From File" id="readFile"/>
          </p>
          <p id="userMessageParams">
            <input type="text" size="50" id="userMessage" />
          </p>
      </section>

    <section id="inputForm" />
      <p>
        <label id="l1" for="title">Title</label>
        <input type="text" id="title" size="20">
      </p>
      <p>
        Map Options<br>
        <label id="l2" for="resize">Allow map to be resized:
        </label>
          <input type="checkbox" id="resize"/>
          <br/>
        <label id="l3" for="recenter">
             Allow map to be recentered:
        </label>
            <input type="checkbox" id="recenter" checked />
      </p>
      <p>
        Type of Map<br>
        <input type="radio" name="maptype" id="XYZMap"
               value="XYZMap"/>
        <label id="label1" for="XYZMap">XYZ map </label>
        <br/>
        <input type="radio" name="maptype" id="StamenMap"
               value="StamenMap"
                    checked />
        <label id="label2" for="StamenMap">Stamen Map
        </label>
      </p>
      <p>
        Screen Size<br>
        <input type="radio" name="screenSize" checked
               id="600x480" value="600x480" />
        <label id="label3" for="XYZMap">600x480 </label>
        <br/>
        <input type="radio" name="screenSize" id="1024x768"
               value="1024x768"/>
        <label id="label4" for="1024x768">1024x768 </label>
        <br/>
        <input type="radio" name="screenSize" id="1280x800"
               value="1280x800"/>
        <label id="label5" for="XYZMap">1280x800 </label>
      </p>

      <p>
        Center of Map<br>
        <label id="label1" for="lat">Latitude&nbsp;&nbsp;&nbsp;
        </label>
        <input type="number" id="lat" value="-77" /><br />
        <label id="label2" for="long"  >Longitude </label>
        <input type="number" id="long" value="39"  />
      </p>
      <input type="button" value="Save" id="saveNew" />
      <input type="button" value="Save" id="saveUpdate" />
      <input type="button" value="Cancel" id="cancelEdit" />
    </section>

  </body>
</html>
```

Program 107 – HTML CRUD interface

This html results in the following screen.



Map Example

Example map input screen

Data records

Title

Map Options
Allow map to be resized: ☐
Allow map to be recentered: ☑

Type of Map
○ XYZ map
● Stamen Map

Screen Size
● 600x480
○ 1024x768
○ 1280x800

Center of Map
Latitude -77
Longitude 39

Figure 18 – HTML implementation of the wire frame design

Chapter 6.1. 3 Application Data

The following diagram shows the data, from the forms, which will be needed for the application.

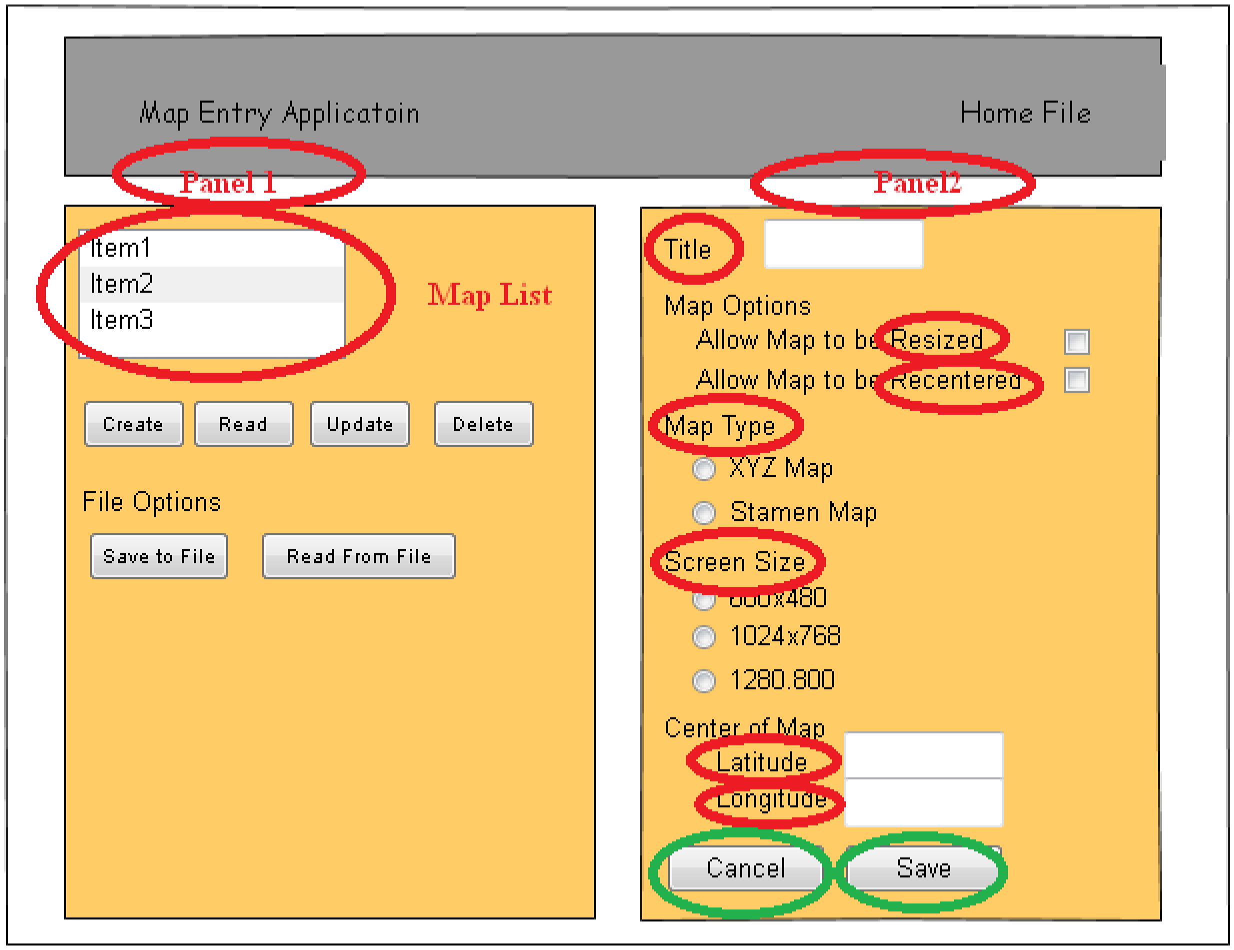


Figure 19 – Data items found in the design

The data in this application consists of the items that make up a map object (all the fields circled in red in Panel 2) and an array of all the map objects (the list box circled in red in Panel 1). The JavaScript definition of the map data fields can be represented as a Map constructor function as shown below.

This JavaScript data represents a single entity, and so is stored in a separate file.

```
 /*
    File name:    MyMap.js
    Type:         Object definitoin
    Purpose:      To create and manipulate Map objects
    Author:       Charles Kann
    Date:         July 7, 2017

    Modification History:
        7/7/2017 - Initial Release
 */
function MyMap(options) {
  // Set the Constructor Function Type
  this.__cfName = "MyMap";
  // All object must be created with a title...
  if (options.title == null || options.title == undefined)
   throw "Title must be specified";

  // Set default properties for a map
  // Note that title is set in case a user
  // creates this object using Object.create;
  this.id = null;
  this.title = "Please change the title"
  this.recenter = true;
  this.resize = false;
  this.mapType = "Stamen";
  this.screenSize = "640x480";
  this.lat = -77;
  this.long = 39;

  // Get properties from parameter
  // Note that this acts like operator overloading.  If
  // a property is not set in the parameter, the default
  // will be used.  Also this allows for other data fields
  // that might have been added.
 outerloop:
   for (i in options) {
      for (j in this)   {
      if (i == j) {
          this[i]= options[j];
          continue outerloop;
      }
    }
    console.log("Property " + i + " is not a default Map property");
      this[i] = options[i];
  }

  // Return the current object.  This is not needed, but
  // but doesn't hurt and makes the intent clear.
  return this;
}

  // Function:  update
  // Purpose:   to update all of the properties in the object.
  //            Note that this function allows unstructured
  //            properties
  // Input:     objectbject with property values to update
  // Output:    none
  // Side Effects: object properties are updated.
MyMap.prototype.update = function(options) {

     // Title property cannot change on update.  It can be
       // thought of as the immutable key for the object.
     var t1 = options.title;
     var t2 = this.title;
     if (t1 != t2)
       throw "Title changed not allowed in update";

    // Get properties from parameter
    // Note that this acts like operator overloading.  If
    // a property is not set in the parameter, the default
    // will be used.  Also this allows for other data fields
    // that might have been added.
  outerloop_1:
    for (i in options) {
      for (j in this)   {
      if (i == j) {
          this[j]= options[i];
          continue outerloop_1;
      }
    }
    console.log("Property " + i + " is not a default MyMap property");
    this[i] = options[i];
  }
}

// Function:  toString
// Purpose:   to create a string representation of the object
// Input:     none
// Output:    string representation of the object
// Side Effects: none
MyMap.prototype.toString = function() {
  return (JSON.stringify(this));
}
```

Program 108 – Map object definition

The map objects are stored in a global array variable in the head of the CRUD.html file. This is the only global variable in this program. Global variables should be used sparingly, and only if the intent is known and can be explained. In this case, the records will be used in both the constructor function and the library functions, and thus needs to be global.

Chapter 6.1. 4 Mapping the object to data fields

Now that the properties for the object has been defined, the next step is to map it to the form which was generated earlier. This is done in the table below. This table contains 3 columns. The first column is the name of the item in the object. The second column is the corresponding field name on the form.

The last column requires some explanation. Panel 2 can be brought up in 3 modes that are create, read, or update. For each of these fields the field has 2 possible attributed, if it is read-only, and if it is visible. This column defines the value of these attributes for each. Note that the two buttons on Panel 2 (circled in green) will also have values for each mode of the form, and so these two buttons are included in the table.

Object Name	Field Name	Notes	Default value	Attributes
title	title		“Enter Title”	Create: Shown, read_only = false Read: Shown, read_only = true Update: Shown, read_only = true
recenter	resize		false	Create: Shown, read_only = false Read: Shown, read_only = true

				Update: Shown, read_only = false
resize	recenter		true	Create: Shown, read_only = false Read: Shown, read_only = true Update: Shown, read_only = false
mapType	mapType	Values are from radio buttons are: Stamen, XYZMap	Stamen	Create: Shown, read_only = false Read: Shown, read_only = true Update: Shown, read_only = false
screenSize		Values are from radio buttons are: 600x480, 1024x768, 1280x800	600x480	Create: Shown, read_only = false Read: Shown, read_only = true Update: Shown, read_only = false
lat		lat	-77	Create: Shown, read_only = false Read: Shown, read_only = true Update: Shown, read_only = false
long		long	39	Create: Shown, read_only = false Read: Shown, read_only = true Update: Shown, read_only = false
Cancel Button				Create: Shown, behavior: hide form Read: Shown, behavior: hide form Update: Shown, behavior: hide form
Save Button				Create: Shown, function: SaveNew Read: Hidden Update: Shown, function: SaveUpdate

This table allows functions to be written that reset the form (sets the values to defaults), reads the values from the form into an object, set the values on the form to values from an object, and to set the attributes for the fields on the form to read-only or read-write. It shows that 2 Save buttons are needed (one for update, and one for create), and how to set the visibility of those two Save buttons.

The functions defined in this table are shown below and are stored in a file CRUDLibrary.JavaScript. The attributes for the buttons will be set later when the behavior for the events are defined.

```
/*
    File:     CRUDLibrary.js
      Author:   Charles Kann
      Date:     July 8, 2017

      Purpose:  To define library functions needed
                for the Map Example CRUD application

      Methods:   resetForm
                 writeDataToForm
                     readDataFromForm
                     setFormReadWrite
                     setFormReadOnly

      Modification History:
          7/8/2017 - Initial Release

 */

/*
    Function:     resetForm
      Author:       Charles Kann
      Date;         7/8/2017
      Purpose:      Set form to default values
      Input:        None
      Output:       None
      Side Effects: Fields on form have default values
 */
function resetForm() {
  $("#title").val("");
  $("#resize").prop("checked", false);
  $("#recenter").prop("checked", true);
  $("#StamenMap").prop("checked", true);
  $("#600x480").prop("checked", true);
  $("#lat").val("-77");
  $("#long").val("39");
}

/*
    Function:     writeDataToForm
      Author:       Charles Kann
      Date;         7/8/2017
      Purpose:      Set form to values form obj
      Input:        obj - an array containing data values.
                          obj must have values for all fields.
      Output:       None
      Side Effects: Fields on form have values from obj
 */
function writeDataToForm(obj1) {
  $("#title").val(obj1.title);
  $("#resize").prop("checked", obj1.resize);
  $("#recenter").prop("checked", obj1.recenter);
  $("#"+obj1.mapType).prop("checked", true);
  $("#"+obj1.screenSize).prop("checked", true);
  $("#lat").val(obj1.lat);
  $("#long").val(obj1.long);
}

/*
    Function:     readDataFromForm
      Author:       Charles Kann
      Date;         7/8/2017
      Purpose:      create an object with data values for
                         all fields on form
      Input:        None
      Output:       obj - an object with the data values
      Side Effects: None
 */
function readDataFromForm() {
   obj = new Object();
   obj.title = $("#title").val();
   obj.resize = $("#resize").is(":checked");
   obj.recenter = $("#recenter").is(":checked");
   obj.mapType = $("input[name='maptype']:checked").val();
   obj.screenSize = $("input[name='screenSize']:checked").val();
   obj.lat = $("#lat").val();
   obj.long = $("#long").val();

   return obj;
}

/*
    Function:     setFormReadWrite
      Author:       Charles Kann
      Date;         7/8/2017
      Purpose:      Set all form fields to allow
                         reading and writing.
      Input:        None
      Output:       None
      Side Effects: Fields on form are read/write
 */
function setFormReadWrite() {
  $("#title").attr('readonly', false);
  $("#resize").attr('disabled', false);
  $("#recenter").attr('disabled', false);
  $("#StamenMap").attr('disabled', false);
  $("#XYZMap").attr('disabled', false);
  $("#600x480").attr('disabled', false);
  $("#1024x768").attr('disabled', false);
  $("#1280x800").attr('disabled', false);
  $("#lat").attr('readonly', false);
  $("#long").attr('readonly', false);
}

/*
    Function:     setFormReadOnly
      Author:       Charles Kann
      Date;         7/8/2017
      Purpose:      Set all form fields to read-only
      Input:        None
      Output:       None
      Side Effects: Fields on form are readonly
 */
function setFormReadOnly() {
  $("#title").attr('readonly', true);
  $("#resize").attr('disabled', true);
  $("#recenter").attr('disabled', true);
  $("#StamenMap").attr('disabled', true);
  $("#XYZMap").attr('disabled', true);
  $("#600x480").attr('disabled', true);
  $("#1024x768").attr('disabled', true);
  $("#1280x800").attr('disabled', true);
  $("#lat").attr('readonly', true);
  $("#long").attr('readonly', true);
}
```

Program 109 – Library functions for the CRUD application

Chapter 6.1. 5 Application behavior – events

Many readers will have come to this book having had a minimum of programming experience, and often that experience is with a program that begins with a main method from which all actions in the program emanate. The program works in a top-down manner, where all actions are part of tree rooted in the main method.

The programming model presented here uses a very different model of programming that we will call Event Based Programming (EBP). EBO is very different from procedural. First, there is no main in the program that is the parent of all the actions in the program. Functions are triggered (or execution called) via asynchronous events, in our case as actions from the user. The system then runs code to respond to that event, returning the program to some known, safe state from which it can respond to other events.

In the application in this chapter, events are generated by the user when a button is pressed. The pressing of the button creates an event that calls a function associated with that button.

Unlike the design of a procedural program, which proceeds in a top down manner, EBP sets up the framework for the program, and then defines functions for each of the behaviors (or buttons presses) in this application.

To design the program a table is created to explain the behavior to respond to each button press. The following figure shows the buttons that need to be created for this application. This is translated into a table where each button is assigned an id, and a column is created to outline the behavior for each button.

Note that one of the buttons in the diagram, the save button in Panel 2, is special in that it will be implemented as two separate buttons. The reason is that the save button is context sensitive; it will call one function when the user is doing a create, and another function when the user is doing an update. This will be implemented as two separate buttons having different ids but the same values. It will appear to the user that the button is a Save button but depending on the context the button will be different.

There is one other issue. A message box is provided to give feedback to the user. Each button will have zero, one, or more messages that they can provide.

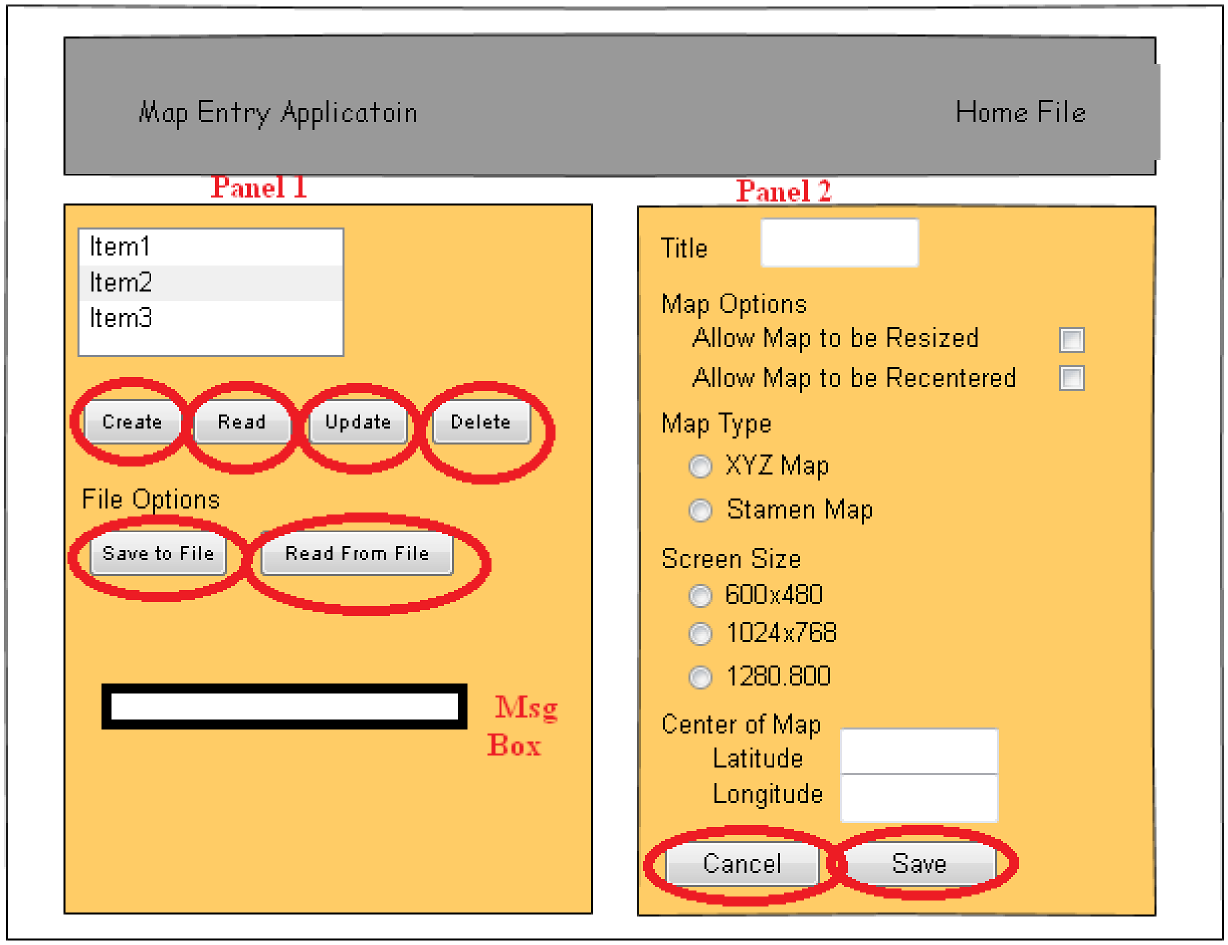


Figure 20 – buttons with functionality to be defined

Button	ID	Notes	Behavior
Create	create		1. Set form to default values (resetForm()) 2. Clear and hide message box 3. Set form so all fields can be edited (setFormReadWrite()) 4. Set buttons: a. saveNew show b. saveUpdate hide c. cancel show 5. Display input form
Read	read		1. Get the title (key) from list 2. Read record from records array a. Print error and throw exception if not defined. 3. Clear and hid message box 4. Write data to forrm (writeDataToForm(obj)) 5. Set form so all fields are read-only (setFromReadOnly()) 6. Set buttons a. saveNew hide b. saveUpdate hide

			c. cancel show 7. Display input form
Update	update		1. Get the title (key) from list 2. Read record from records array a. Print error and throw exception if not defined. 3. Clear and hid message box 4. Write data to forrm (writeDataToForm(obj)) 5. Set form so all fields are readWrite(setFromReadWrite()) 6. Set the title to read-only 7. Set buttons a. saveNew hide b. saveUpdate show c. cancel show d. 8. Display input form
Delete	delete		1. Get the title (key) from list 2. Read record from records array a. Print error and throw exception if not defined. 3. Prompt to confirm delete a. If no – message that record not deleted b. If yes i. Remove record from array ii. Remove item from list iii. Message that item was deleted
Save to File	saveFile		1. Use JSON.strinigfy to make a JSON formated file for the records array, and LocalStorage.setItem to put it in local storage.
Read from File	readFile		1. Empty the list 2. Empty the records array 3. Read and parse the array from local storage 4. For each member of the arrray a. Parse data b. Store to list c. Store to records array
Cancel	cancel		1. Hide input form
Save	saveUpdate		1. Read data from form (readDataFromForm()) 2. Get object from records array 3. If record not found, throw exception 4. Update record (obj.update()) 5. Hide input form.
Save	saveNew		1. Read data from form (readDataFromForm()) 2. Create new map object 3. Push object on records 4. Update the list 5. Hide input form.

These behaviors are attached to the buttons when the form is loaded, which is done in the JQuery onload function in the file CRUDOnload.JavaScript. This file is shown below.

```
/*
    File Name:   CRUDOnload.js
    Author:      Charles Kann
    Date:        July 9, 2019
    Purpose:     To initialize the application, and set update
                 the event function call backs.

      Events in this file:
        create - create a new Map record
        read   - read a Map record (display only)
        update - update a Map record
        delete - delete a  Map record
        save to file - save the record array to local storage
        read from file - read the record array from local storage
        cancel - cancel editing a Map record
        save(1)  - save a map record on create
        save(2) - save a map record on update

    Modification History:
          7/9/2017 - Initial release
*/
$(function() {
  // Set form to default values
    resetForm();
    $("#userMessage").hide();
    $("#userMessage").val("");

    /*
        Function:  Create a new record
        Purpose:   To respond to the create
                   button
    */
    $("#create").click( () => {

      // Initialize form
      $("#userMessage").hide();
      $("#userMessage").val("");
      resetForm();
      setFormReadWrite();

      // Set buttons
      $("#saveNew").show();
      $("#saveUpdate").hide();
      $("#cancelEdit").show();
      $("#inputForm").show();
    });

    /*
        Function:  Read a Map record
        Purpose:   To respond to the read
                   button
   */
    $("#read").click( () => {
      // get record to read from list
      var myTitle = $("#dataRecords").val();

      //Find the record to read
      var recordToUpdate = records.find( (currentObject) => {
        return (currentObject.title == myTitle)
      });

      // Record is not found, throw exception
      if (recordToUpdate == undefined) {
        $("#userMessage").show();
        $("#userMessage").val("Record does not exist - Make sure  a
record is selected" );
          throw "The title does not exists";
      }

      // Initialize form
      $("#userMessage").hide();
      $("#userMessage").val("");
      writeDataToForm(recordToUpdate);
      setFormReadOnly();

      // Set buttons
      $("#saveNew").hide();
      $("#saveUpdate").hide();
      $("#cancelEdit").show();
      $("#inputForm").show();
    });

    /*
        Function:  Update a Map record
        Purpose:   To respond to the update
        button
   */
    $("#update").click( () => {
      // get record to read from list
      var myTitle = $("#dataRecords").val();

      //Find the record to read
      var recordToUpdate = records.find( (currentObject) => {
        return (currentObject.title == myTitle)
      });

      // Record is not found, throw exception
      if (recordToUpdate == undefined) {
        $("#userMessage").show();
        $("#userMessage").val("Record does not exist - Make sure  a
record is selected" );
          throw "The title does not exist";
      }

      // Initialize form
      $("#userMessage").hide();
      $("#userMessage").val("");
      writeDataToForm(recordToUpdate);
      setFormReadWrite();
      $("#title").attr('readonly', true);
      // Set buttons
      $("#saveNew").hide();
      $("#saveUpdate").show();
      $("#cancelEdit").show();
      $("#inputForm").show();
    });

  /*
      Function:  Delete a Map record
        Purpose:   To respond to the delete
                   button
   */
    $("#delete").click( () => {
      // get record to read from list
      let myTitle = $("#dataRecords").val();

      // Record is not found, throw exception
      if (myTitle == null)
      {
        $("#userMessage").show();
        $("#userMessage").val("Record does not exist - Make sure  a
record is selected" );
          throw "The title does not exist";
      }

      // Confirm Delete
      let retVal = confirm("Do you want to delete " + myTitle + " ?");
      if( retVal == true ){
        //Find the record to read
        let index = records.findIndex ( (currentObject) => {
          return (currentObject.title == myTitle)
        });

        // Record is not found
        if (index == -1) {
          $("#userMessage").show();
          $("#userMessage").val("Record " + myTitle + " not found");
        }
        // Record is found, remove it from records and list
        else {
          records.splice(index, 1);
          $("#dataRecords option[value='" + myTitle + "']").remove();
          $("#userMessage").show();
          $("#userMessage").val("Record " + myTitle + " deleted");

        }
      }

      // Inform the user the record is not deleted, as
        // was indicated by return from dialog
      else{
        $("#userMessage").show();
        $("#userMessage").val("Record " + myTitle + " not deleted");

      }
      });
    /*
        Function:  Save records to local storage
        Purpose:   To respond to the Save To File
        button
   */
    $("#saveFile").click( () =>  {
      localStorage.setItem("MapData", JSON.stringify(records));
    });

    /*
        Function:  Read records from local storage
        Purpose:   To respond to the Read From File
        button
   */
    $("#readFile").click( () => {

      // clear out the list and array
        $("#dataRecords").empty();
      records = new Array();

      // get the records array from local storage
      let arr = JSON.parse(localStorage.getItem("MapData"));

      // Each record was set to JSON independently.  Get
        // each record, parse it, and put it in the records
        // array and list.
      for (var i = 0; i < arr.length; i++) {
        records[i] = new MyMap(arr[i]);
        $('#dataRecords').append($("<option></option>")
            .attr("value", records[i].title)
            .text(records[i].title));
      }
    });

    /*
        Function:  Save record data from create
        Purpose:   To respond to the Save Button
   */
    $("#saveNew").click( ()=> {

      // See if record exists.  Do not allow duplicates.
      var saveObj = readDataFromForm();
      var recordExists = records.find( (currentObject, idx) => {
        return (currentObject.title == saveObj.title)
      });
      if (recordExists != undefined)
        throw "The title already exists";

        // Put new record in records array, and update list.
      records.push(new MyMap(saveObj));
      $('#dataRecords').append($("<option></option>")
                        .attr("value",saveObj.title)
                        .text(saveObj.title));

      // hide the input form
      $("#inputForm").hide();
    });
    /*
        Function:  Save record data from update
        Purpose:   To respond to the Save Button
   */
    $("#saveUpdate").click( () => {

      // Check if record exists.  It must exist to
        // update it.
      var saveObj = readDataFromForm();
      var recordToUpdate = records.find(function(currentObject) {
        return (currentObject.title == saveObj.title)
      });

      if (recordToUpdate == undefined)
        throw "The title does not exists";

        // Update record.
      recordToUpdate.update(saveObj);

      // hide the input form.
      $("#inputForm").hide();
    });

    /*
        Function:  Cancel Edit
        Purpose:   To respond Cancel Button
   */
    $("#cancelEdit").click( () => {
      $("#inputForm").hide();
    });
});
```

Program 110 – Applications events set in the onLoad function

What you will learn

In this chapter, you will learn:

1. The basic function of the Node.js server environment, the npm package manager, and the Sails framework.
2. Installing and accessing Node.js and Sails
3. Creating an application with a REST Interface for the CRUD program developed in the last chapter.
4. What a REST interface is, and how to access it.
5. Accessing the REST interface from Postman
6. Accessing the REST interface form the CRUD program using AJAX to persist the data on server

Chapter 7 Creating a server for the persistent storage of our CRUD application

In Chapter 6 a fully working, simple CRUD application was built on the information presented in the first 5 chapters. The biggest problem with the application from Chapter 6 is that the data was persisted on the local computer using the LocalStorage. This means that the application can be used by only one user on one computer.

In this chapter the CRUD application from Chapter 6 is changed so that it can be accessed from remote client browser environments. This will allow multiple users to have shared access to the data.

Chapter 7.1 Creating and Accessing a Server Using Node.js, Sails

In this section, an HTTP web CRUD server will be created using Node.js with Sails Blueprints. The HTTP server will be used with the Postman application to explain REST interfaces, and the server will be accessed and tested using the Postman application.

Chapter 7.1.1 Node.js

For the purposes of this text, Node.js can be thought of as a server runtime that is used to run JavaScript. In this way, the *node* command can be thought of like the *java* command. Just as the *java* command runs an interpreter to run Java Byte Code (JBC), so the *node* command runs an interpreter to run JavaScript. Any JavaScript program can be run with Node.js, not just servers, including interfacing to single board computers like the Raspberry Pi or Arduino[41].

This book will be using Node.js from within Sails, so there will not be a need to ever directly run the *node* command for the server that will be created.

[41] See https://www.w3schools.com/nodejs/nodejs_intro.asp for more information about running programs using Node.js including creating servers, accessing databases, and programming a Raspberry Pi. For information about using Node.js and Arduino, a google search of “node Arduino” will bring up multiple examples.

To install Node.js, go to the site https://nodejs.org/en/, and select the option for the latest Long-Term-Release (LTR) version. Then follow the instructions for your operating system. When writing this text, Node.js was installed on a Windows computer, and the LTR downloaded a ".msi" file, and the whole installation completed without any problems.

To verify the installation of node, open a DOS command window and type "node -v". You should get back a string describing the version of node you have installed.

Chapter 7.1.2 npm

Most modern languages can access tools, frameworks, and other useful libraries of functionality that are built on top of the language. For example, frameworks have been built for most modern languages that standardize and abstract how interface with databases. For each language, there are multiple number of these frameworks, and no single project would use all of them. Because there are so many, and some conflict with each other, these tools are not built into the base language.

Not having the tools built into the languages leads to the issue of how to make them available to developers. The old method (and the one still used today in Java) is to require the programmer to manage their own environments. But this is difficult, and except for large environments that can afford to have personal dedicated to this effort, unworkable.

The modern answer to the tools build problem is to create *packages* that contain all the files and information needed to install the tool or framework. These packages are then managed by a package manager. Most modern languages have a package manger. For example, in C# there is NuGet, in Ruby these is Ruby-Version-Manager (rvm) and bundler, and in Python there is pip and PyPM.

Node Package Manager (npm) is the Node.js tool used to manage packages. It is installed by default when you install Node.js. It will be used in the next section to install the Sails package.

To verify the installation of npm, open a DOS command and type "npm -v". You should get back a string describing the version of npm you have installed.

Chapter 7.1.3 Installing Sails

When implementing a web server application, most programmers choose a framework to start from. These frameworks provide a standard view of the system, as well as many of the tools that make developing a basic implementation of the application. Later we will see just how simple the basic application development can be.

Web Sever frameworks generally fall into three large categories: Model-View-Controller (MVC), REpresentational State Transfer (REST), and full-stack. Within MVC, there are two types of frameworks, both of which came from Ruby. They are Sinatra-like and Rails-like. Each of these frameworks has many implementations of the basic scheme, and to see the implementations of these frameworks just in Node.js, see http://nodeframework.com/.

To attempt to explain in detail even one of these frameworks, let alone all of them, is a task for an entire book, so the details of the frameworks, advantages and disadvantages, etcetera will not be covered. This text will choose one framework, Sails, and will simply use it to create persistent storage for out CRUD application.

Sails is a MVC Rails-like framework designed to allow a standard implementation of a server application. Sails was chosen simply because the author has used Rails in the past, and at the time this book was written, Sails was one of the most downloaded Rails like frameworks.

To install Sails, npm will be used. At the DOS command prompt, type

```
npm install sails -g
```

This command tells npm to install the package sails, and to make it globally available for all users. When it completes, you can test that sails has been installed by typing "sails -v". You should get back a string describing the version of Sails you have installed.

Chapter 7.1.4 Implementing your Sails application

Now that Sails has been installed, an application can be written to store the data from the CRUD application on the server. To start, I suggest that a directory named Sails be created in an appropriate place on your computer, and that you create all of your Sails applications in this directory[42].

Change Directory (cd) to your Sails application directory and type the following command:

```
sails new MapData
```

Choose option 2, an empty sails application. Option 1 provides a complete basic framework, with authentication, user management, and even credit card processing, and will be the normal option you will choose in the future. For this project though, all of this infrastructure just gets in the way of implementing a simple application, and so do not select it.

Now generate the specific application we need to store our map data. cd to the MapData directory and type the following command:

```
Sails generate api MapData
```

Doing this has actually generated a Sails Blueprint application that can actually be run, but the application does not know what data it needs to deal with. To fix this, edit the file in your MapData directory named api/models/MapData.js, and change the Primitives section so the file is as follows[43]:

```
/**
 * MapData.js
 *
```

[42] The site https://devdactic.com/rapid-development-with-sailsjs/ provides a good resource for implementing a server in Sails.

[43] For more information about models, see https://sailsjs.com/documentation/concepts/models-and-orm/attributes

```
 * @description :: A model definition.  Represents a database
table/collection/etc.
 * @docs        :: https://sailsjs.com/docs/concepts/models-and-
orm/models
 */

module.exports = {

  attributes: {

    //  ╔═╗╦═╗╦╔╦╗╦╔╦╗╦╦  ╦╔═╗╔═╗
    //  ╠═╝╠╦╝║║║║║ ║ ║╚╗╔╝║╣ ╚═╗
    //  ╩  ╩╚═╩╩ ╩╩ ╩ ╩ ╚╝ ╚═╝╚═╝
    title : 'string',
    resize : 'boolean',
    recenter : 'boolean',
    mapType : 'string',
    screenSize : 'string',
    latitude : 'number',
    longitude : 'number',

    //  ╔═╗╔╦╗╔╗ ╔═╗╔╦╗╔═╗
    //  ║╣ ║║║╠╩╗║╣  ║║╚═╗
    //  ╚═╝╩ ╩╚═╝╚═╝═╩╝╚═╝


    //  ╔═╗╔═╗╔═╗╔═╗╔═╗╦╔═╗╔╦╗╦╔═╗╔╗╔╔═╗
    //  ╠═╣╚═╗╚═╗║ ║║  ║╠═╣ ║ ║║ ║║║║╚═╗
    //  ╩ ╩╚═╝╚═╝╚═╝╚═╝╩╩ ╩ ╩ ╩╚═╝╝╚╝╚═╝

  },

};
sails.config.models.migrate='alter';
```

Program 111 – api/models/MapData.js file

You also need to add the line `sails.config.models.migrate='alter'` at the end of the file. This allows the data to be kept between starting instances of the server. The other options for the migrate option are drop (drop the data in the server each time sails is started), and safe, which should be used once the program is in production.

Now you will start the server by typing the following command:

```
sails lift
```

The server should now start running, and you should get a screen similar to the following.

```
info: Starting app...

 info: ·• Auto-migrating...  (alter)
 info:    Hold tight, this could take a moment.
 info:  ✓ Auto-migration complete.
 info:
 info:                     .-..-.
 info:
 info:    Sails                <|    .-..-.
 info:    v1.0.2                |\
 info:                         /|.\
 info:                        / || \
 info:                      ,'  |'  \
 info:                   .-'.-==|/_--'
 info:                   `--'-------'
 info:    __---___--___---___--___---___--___
 info:  ____---___--___---___--___---___--___-__
 info:
 info: Server lifted in `C:\Users\Charl\sails\MapCrud`
 info: To shut down Sails, press <CTRL> + C at any time.
 info: Read more at https://sailsjs.com/support.

debug: -------------------------------------------------------
debug: :: Sun Aug 19 2018 11:45:10 GMT-0400 (Eastern Daylight Time)

debug: Environment : development
debug: Port        : 1337
debug: -------------------------------------------------------
```

Program 112 – display after sails have started correctly

If you have any errors, check to make sure you are in the MapData directory, and that you spelled all the text in your models file correctly.

Once you have no errors, a CRUD server with create, read, update, and delete capabilities, is up and running on your local computer (named localhost) and on port 1337. This server is ready to be used by the CRUD application. The rest of this chapter will cover accessing the server.

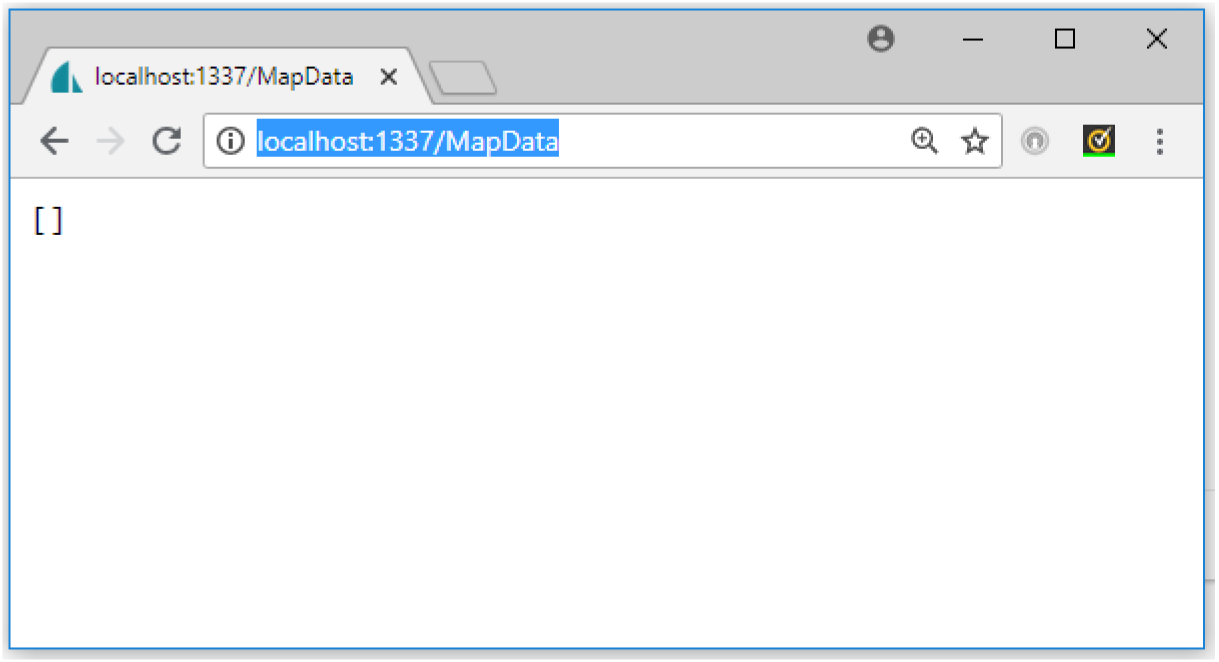
Chapter 7.2 Accessing a Server Using Node.js, Sails

Now that the application server is running, it can be accessed through any browser. To see this type the following Uniform Resource Locator into any web browser:

```
http://localhost:1337/MapData
```

The statement tells the browser to look on the local host for a program attached to port 1337. At that port a resource, MapData, will be found, and data from that application will be shown on the web page.

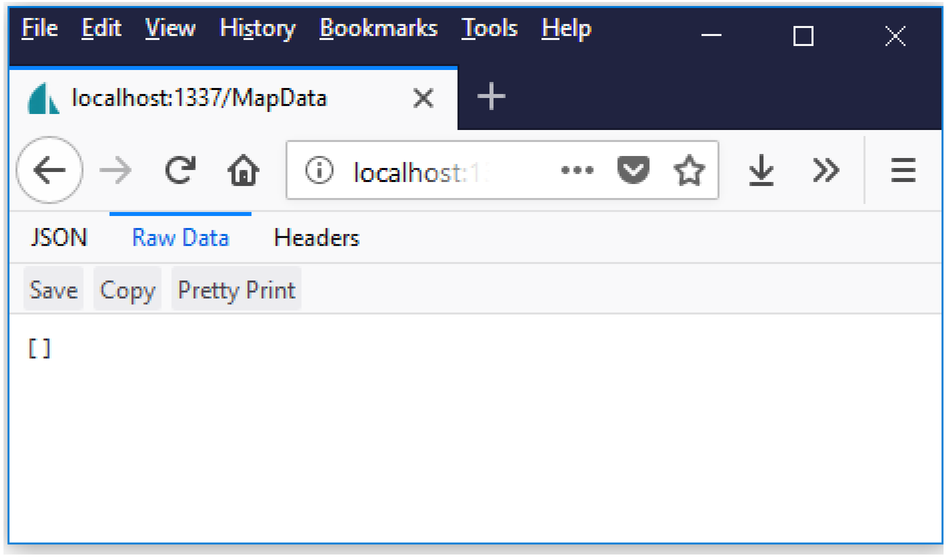
Depending on the browser, you will get different outputs. Chrome will display an open and close square bracket, indicating that there is no MapData in the application.



```
[]
```

Figure 21 – Response from server when URL is called from Chrome

Firefox recognizes that the data is JSON data, and it consists of an empty array. It gives you move options to query this data, but looking at the raw data gives the same open and close square bracket.



```
[]
```

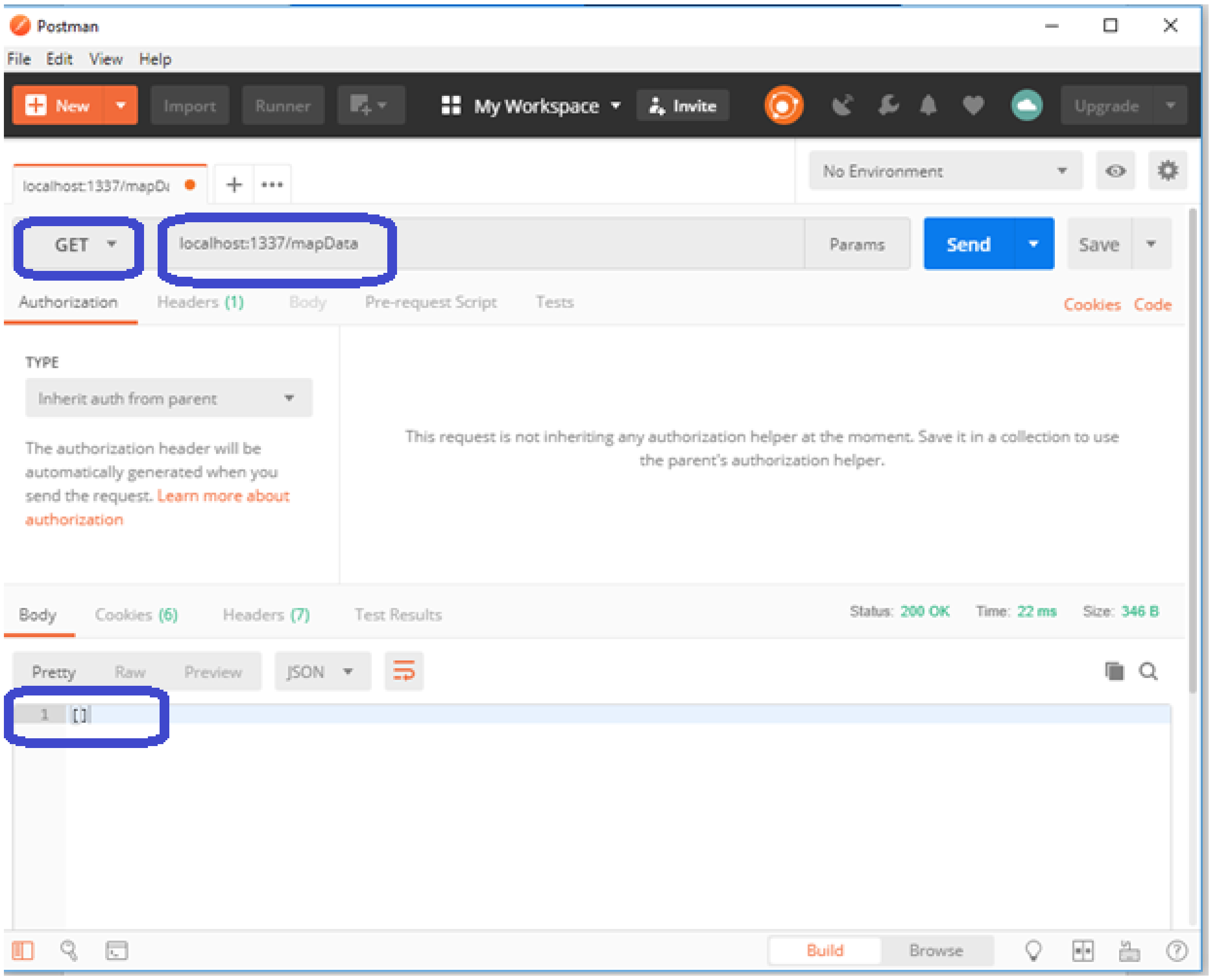
Figure 22 – Response from server when URL is called from Firefox

This is the concept of a REST server. All commands are accessed via HyperText Transport Protocol (HTTP) using HTML requests. These requests can all be generated from the browser, and later in this chapter this will be accomplished using AJAX requests from JavaScript. But to have to write an entire program just to test that the REST services work is inefficient.

Fortunately, there are a number of apps that can be used for interacting with REST APIs. The one used in this text is Postman.

Chapter 7.2.1 Getting started with Postman

To install Postman, go to the website https://www.getpostman.com/, and download the free app. Follow the installation instructions, and the Postman application should come up with a window that appears similar to the one below.



GET localhost:1337/mapData

TYPE
Inherit auth from parent

The authorization header will be automatically generated when you send the request. Learn more about authorization

This request is not inheriting any authorization helper at the moment. Save it in a collection to use the parent's authorization helper.

Status: 200 OK Time: 22 ms Size: 346 B

```
[]
```

Figure 23 – Postman GET request that is the same as Figure 21

From this window, choose the GET method and type in the URL for the server. When you hit the Send button, the window should come back with the two square brackets, as it did with the web page. Postman is now ready to test your system.

Chapter 7.2.2 Creating data

If all Postman does is emulate a very poorly setup browser, it is not that useful. But it does much more. As we said at the end of Chapter 7.1.4, the server has implemented a full set of CRUD operations. However, to access them we need a way to format the requests. Formatting these requests is really not practical (or possible) from a web browser without using JavaScript. And for testing the application, implementing the test cases in JavaScript would be a poor choice. The only reason the web browser was able to easily access the application earlier is because the default request, get with no parameters, is the default request.

Postman allows us to easily develop and test the REST server application. This is probably best seen in the context of using Postman, so the following is a life cycle of an application presented using Postman.

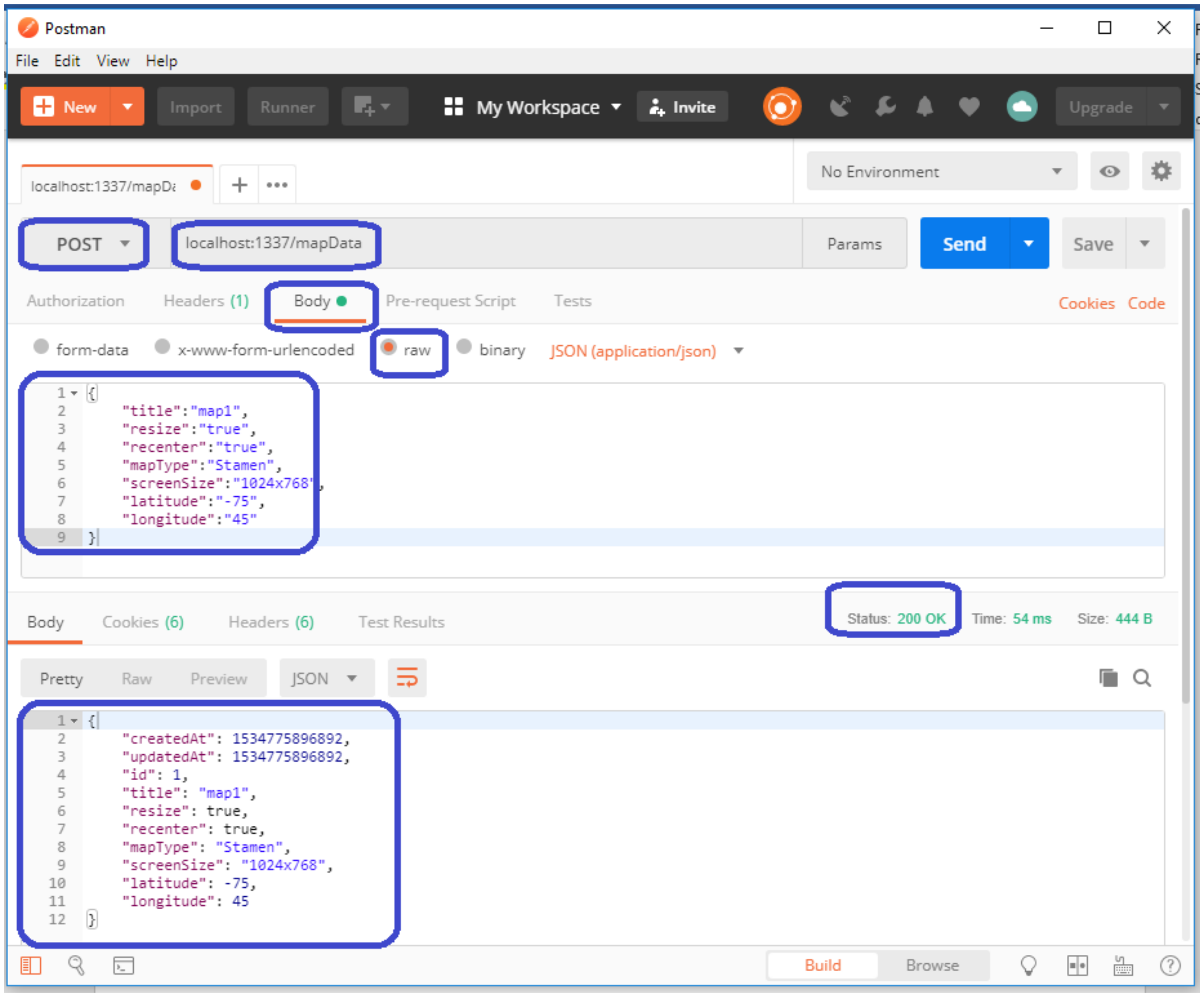
The first step, making sure Postman can reach the server, is already done in the last section. Next data should be added to the server. This is accomplished using the POST method with the same URL. When sending a POST request, note that a body of the post request can be included by clicking the *Body* button. Click the button and include the following JSON object in the body of the request, as shown in the figure below.

```
{
    "title":"map1",
    "resize":"true",
    "recenter":"true",
    "mapType":"Stamen",
    "screenSize":"1024x768",
    "latitude":"-75",
    "longitude":"45"
}
```

Program 113 – Input object for Postman POST request

When this request is sent, the server sends back a status code of 200 indicating the request was successful. It also sends back a copy of the object as it was stored in the database. Note that 3 fields are included that were not part of the original request but are added by Sails. These fields are createdAt, updatedAt, and id. The important filed is the id field, which will be the primary unique key for this record in the persistent storage. This key will be used for subsequent queries to read/update/delete this record.

Add 2 more records, changing at least the title in the Postman body so that you know 3 different records have been saved.



POST localhost:1337/mapData

Body: raw JSON (application/json)

```
{
    "title":"map1",
    "resize":"true",
    "recenter":"true",
    "mapType":"Stamen",
    "screenSize":"1024x768",
    "latitude":"-75",
    "longitude":"45"
}
```

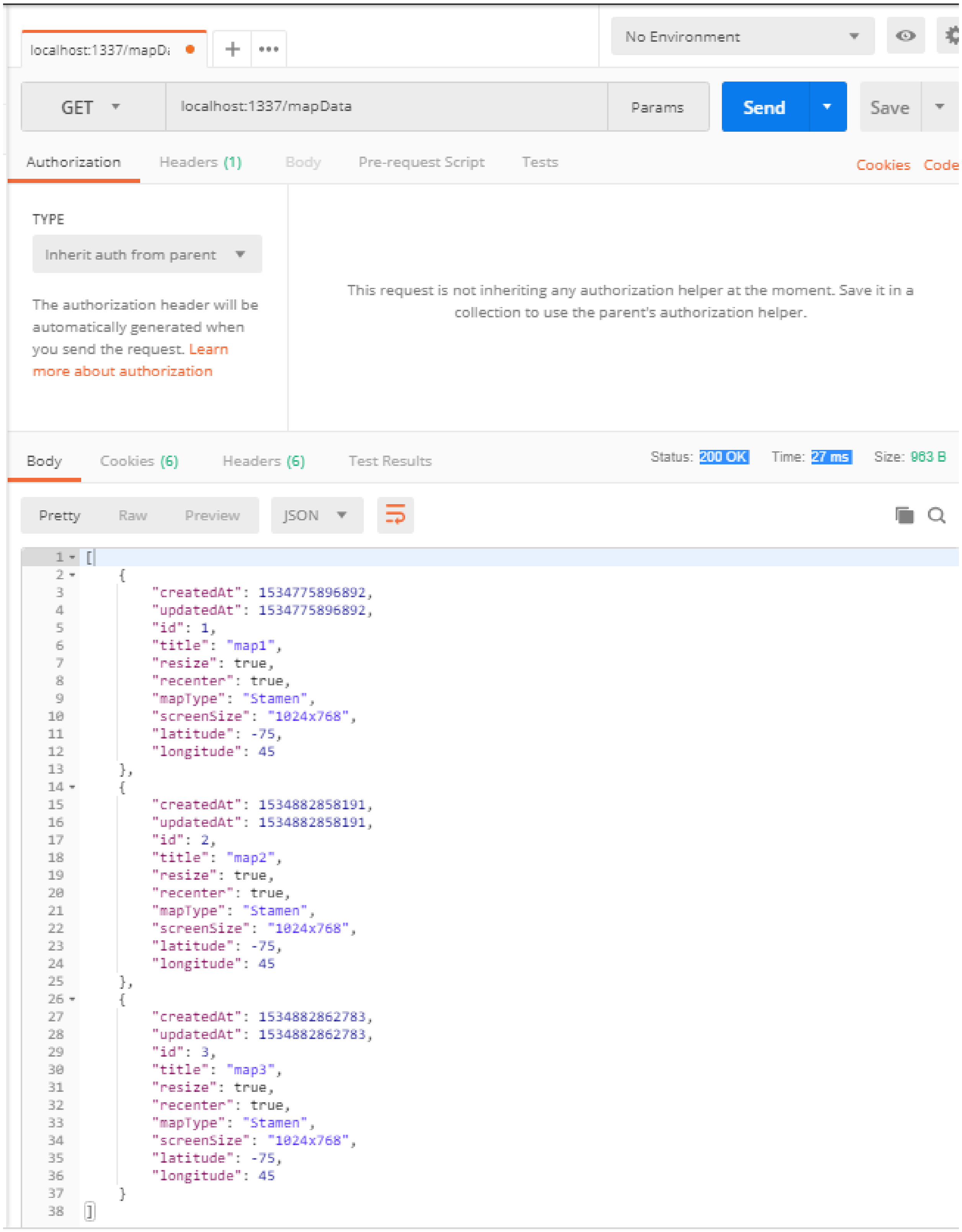
Status: 200 OK Time: 54 ms Size: 444 B

```
{
    "createdAt": 1534775896892,
    "updatedAt": 1534775896892,
    "id": 1,
    "title": "map1",
    "resize": true,
    "recenter": true,
    "mapType": "Stamen",
    "screenSize": "1024x768",
    "latitude": -75,
    "longitude": 45
}
```

Figure 24 – Postman POST request creating a new object on the server

Chapter 7.2.3 Retrieving Data

Now retrieve all of the records placed in the Node.js server and show them in Postman. This is done by sending a GET request, http://localhost:1337/mapData, to the server for MapData. This was done earlier to show that the connection to Node.js was valid and returned a null array as there were no records in the database. Now this query will return an array of the 3 map elements that were posted to the array, as shown in the following figure.



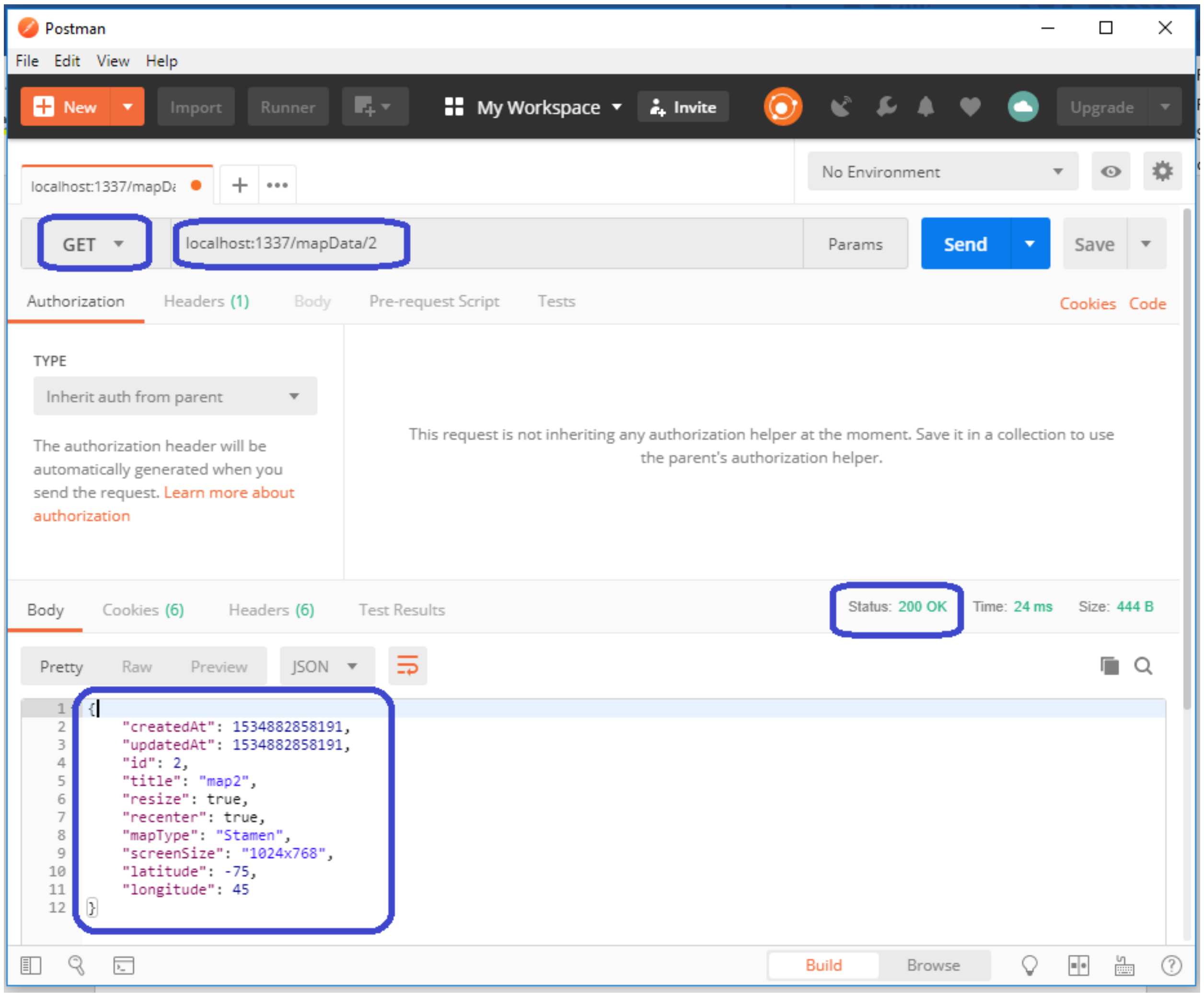
The authorization header will be automatically generated when you send the request. Learn more about authorization

This request is not inheriting any authorization helper at the moment. Save it in a collection to use the parent's authorization helper.

```
[
    {
        "createdAt": 1534775896892,
        "updatedAt": 1534775896892,
        "id": 1,
        "title": "map1",
        "resize": true,
        "recenter": true,
        "mapType": "Stamen",
        "screenSize": "1024x768",
        "latitude": -75,
        "longitude": 45
    },
    {
        "createdAt": 1534882858191,
        "updatedAt": 1534882858191,
        "id": 2,
        "title": "map2",
        "resize": true,
        "recenter": true,
        "mapType": "Stamen",
        "screenSize": "1024x768",
        "latitude": -75,
        "longitude": 45
    },
    {
        "createdAt": 1534882862783,
        "updatedAt": 1534882862783,
        "id": 3,
        "title": "map3",
        "resize": true,
        "recenter": true,
        "mapType": "Stamen",
        "screenSize": "1024x768",
        "latitude": -75,
        "longitude": 45
    }
]
```

Figure 25 – Postman GET request after several objects have been stored on server

To retrieve the information for an individual record, the id of the record to be retrieved is appended to the URL. For example, the second MapData record inserted into the data store was given the id of 2. To retrieve this record, append the number 2 onto the GET query, http://localhost:1337/mapData, and just the second record will be returned as a JSON object.



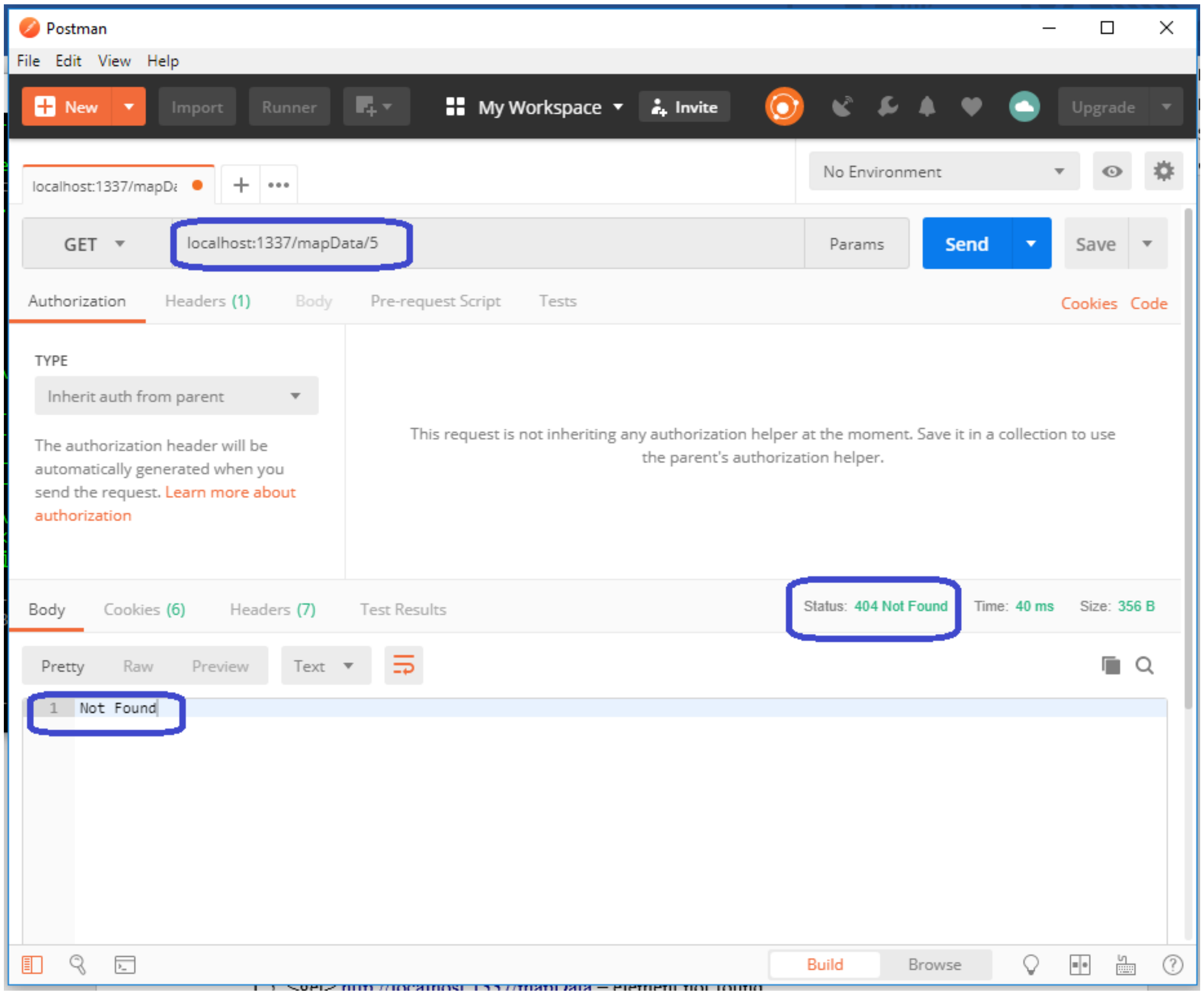
The authorization header will be automatically generated when you send the request. Learn more about authorization

This request is not inheriting any authorization helper at the moment. Save it in a collection to use the parent's authorization helper.

```
{
    "createdAt": 1534882858191,
    "updatedAt": 1534882858191,
    "id": 2,
    "title": "map2",
    "resize": true,
    "recenter": true,
    "mapType": "Stamen",
    "screenSize": "1024x768",
    "latitude": -75,
    "longitude": 45
}
```

Figure 26 – Postman GET request to retrieve a single object

If a request is made for an ID that does not exist, the server will return a 404 status code and return the message “Not Found”.



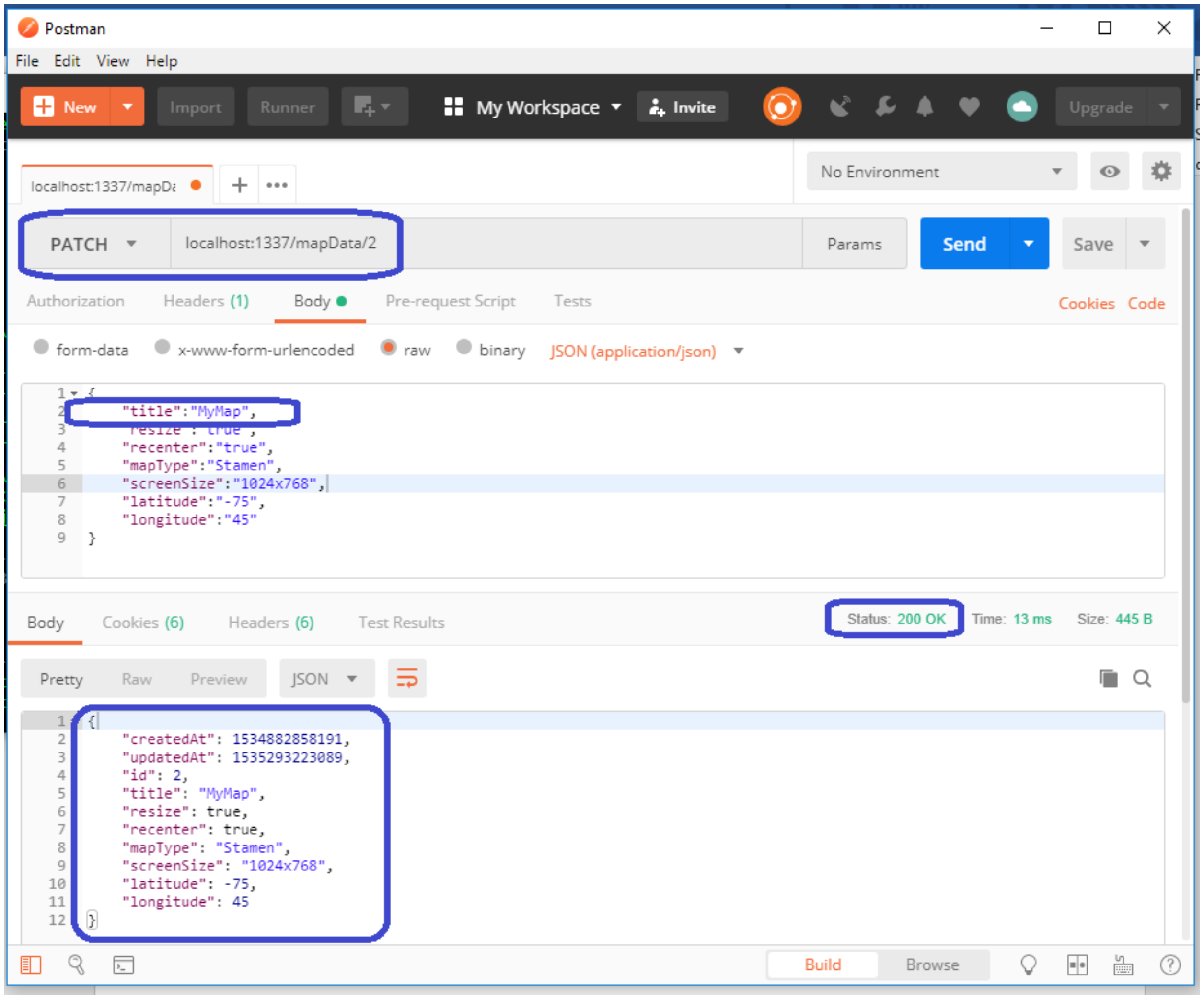
```
Not Found
```

Figure 27 – Postman GET request to retrieve a single object that is not found

Chapter 7.2.4 Updating and Deleting records

The last two operations for a CRUD interface are Update and Delete. To Update a record in the server, the PUT or PATCH method is used with the URL for a specific record in the server. The PATCH request with the URL and fields for ID 2 is shown below. The Update requires all fields in the object to be sent, though in this case only the title is updated.

There are technical differences between PUT and PATCH requests, but either normally will work. If one causes difficulties, try the other.



PATCH localhost:1337/mapData/2

JSON (application/json)

```
{
    "title":"MyMap",
    "resize":"true",
    "recenter":"true",
    "mapType":"Stamen",
    "screenSize":"1024x768",
    "latitude":"-75",
    "longitude":"45"
}
```

Body Cookies (6) Headers (6) Test Results

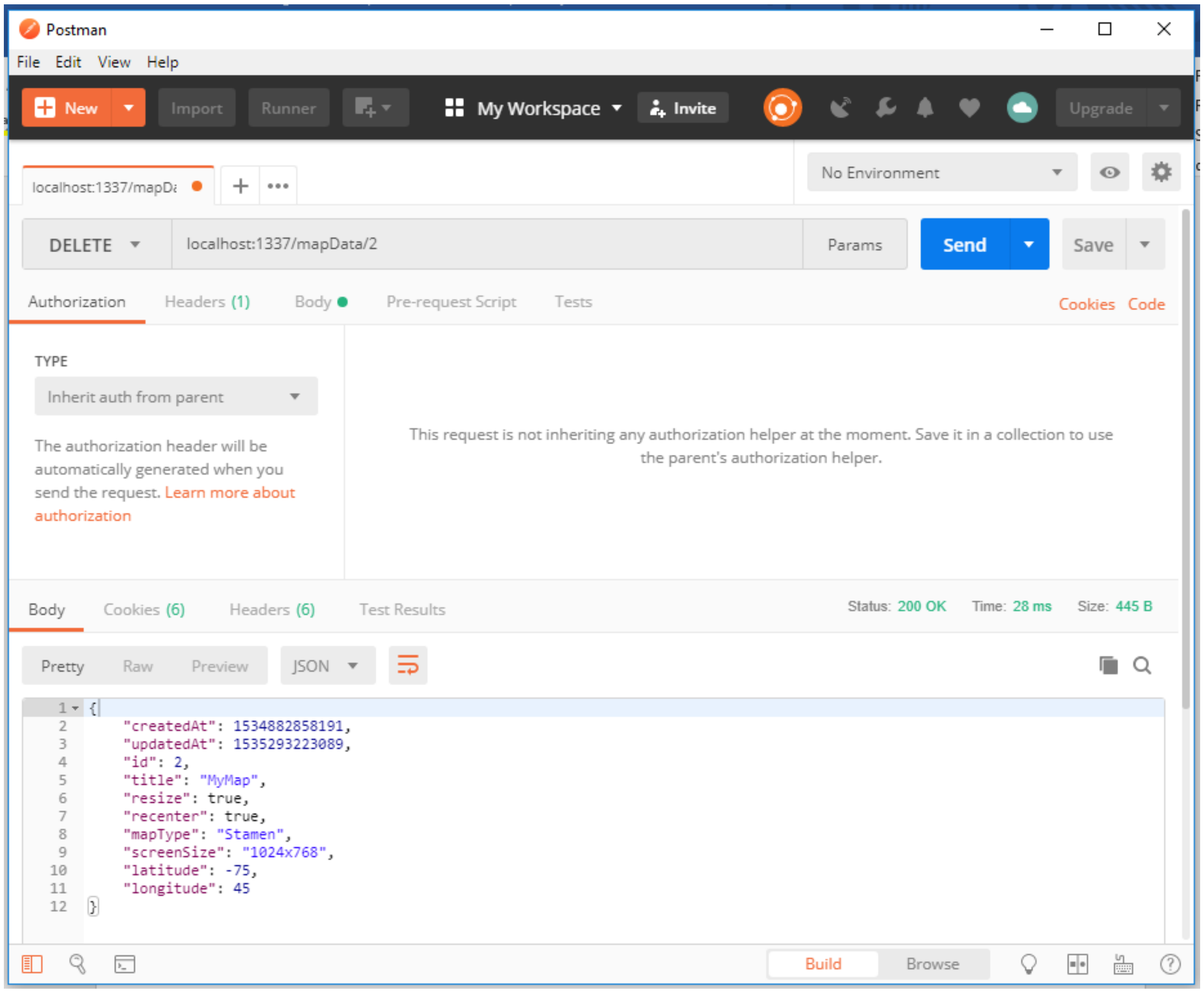
Status: 200 OK Time: 13 ms Size: 445 B

```
{
    "createdAt": 1534882858191,
    "updatedAt": 1535293223089,
    "id": 2,
    "title": "MyMap",
    "resize": true,
    "recenter": true,
    "mapType": "Stamen",
    "screenSize": "1024x768",
    "latitude": -75,
    "longitude": 45
}
```

Figure 28 – Postman PATCH (or Put) request to update an object

Finally, the Delete CRUD option is accomplished by using the URL for a specific record and the DELETE method. The following shows an example of deleting the record with an ID of 2. In this case, the server sent back a Status code of 200, saying the transaction worked, and the JSON of the record that was deleted.

Different servers will choose different Status codes for these transactions, and the Delete transaction will often return a 404 saying the record was not found because it was deleted. This can be confusing, as it could say that the record was not found, OR the record was deleted. What status codes are returned for different operations, especially when there is a complex transaction, can be confusing, and even the documentation as to what a server will return can be hard to understand. The best advice might be to use trial-and-error to find out what status codes will be returned.



localhost:1337/mapData/2

TYPE

Inherit auth from parent

The authorization header will be automatically generated when you send the request. Learn more about authorization

This request is not inheriting any authorization helper at the moment. Save it in a collection to use the parent's authorization helper.

Status: 200 OK Time: 28 ms Size: 445 B

```
{
    "createdAt": 1534882858191,
    "updatedAt": 1535293223089,
    "id": 2,
    "title": "MyMap",
    "resize": true,
    "recenter": true,
    "mapType": "Stamen",
    "screenSize": "1024x768",
    "latitude": -75,
    "longitude": 45
}
```

Figure 29 – Postman DELETE request to delete an object

Chapter 7.2.5 Summary of CRUD REST server transactions

The REST transactions for a CRUD interface are summarized in the following table.

Method	URL	Meaning
GET	/:model	Return an array of JSON objects that are stored for this model
GET	/:model/:id	Return an JSON object for this model with this ID
PUT	/:model	Create a record for this model in the server
POST or PATCH	/:model/:id	Update the record for this model with this ID
DELETE	/:model/:id	Delete the record for this model with this ID

Figure 30 – REST transaction summary

Chapter 7.3 Communicating with the server using JavaScript

This section will describe how to send a transaction using JavaScript from a web browser to the server.

Chapter 7.3.1 Sending a transaction to the server

There are 4 steps involved in writing a program to send a transaction to the server from JavaScript:

1. An HTTP object is created, and the proper values set in that object.
2. A callback function (or listener) is set to receive the data back from the server
3. The request is sent asynchronously[44] to the server.
4. The server will process the request, sending back (multiple) responses as it proceeds. Once the request has completed, the server will send back a ready state of 4 and a status code categorizing the result of the transaction.

These 4 steps are shown in the following JavaScript program that sends a request for an array of JSON map objects. This code should be placed in a file named "ReadMaps,html" in the assets directory of your Sails application, and can be accessed using the URL http://localhost:1337/ReadMaps.html.

```
<script>
  let xmlhttp = new XMLHttpRequest();
  xmlhttp.open("GET", "/mapdata", true);
  xmlhttp.setRequestHeader("Content-Type", "application/json");
  xmlhttp.onreadystatechange = function ()
  {
    console.log("State = " + xmlhttp.readyState +
        ", and status = " + xmlhttp.status);
    if(xmlhttp.readyState === 4)
    {
      console.log("State = " + xmlhttp.readyState +
        ", and status = " + xmlhttp.status);
    }

  }

   xmlhttp.send();
</script>
```

Program 114 – XMLHttpRequest example

The following will explain each part of the transaction.

1. An object variable, in this case named xmlhttp, is created using the constructor function XMLHttpRequest. This object will be used to build the request to be sent to the sever.

[44] There is a synchronous version that can be used to send the transactions to the server and using the synchronous version does not require the callbacks and raise the issues involved in processing the asynchronous version. However, using the asynchronous version will tie up the browser so that the user cannot interact with the browser until the request has finished. It is highly recommended that the synchronous version never be used, and it has been deprecated in most of the major libraries.

2. The xmlhttp variable is *opened* (or initialized) to represent a GET method request on the server, using /mapData as the URL. This is the same request made in Postman to retrieve all of the records in the server. The third parameter is whether this request is asynchronous. This parameter should always be true. Note that the request has only been built, it has not yet been sent to the server.

3. A method is defined and attached to the onreadystatechange event to respond to the server after the request is sent. Note that the request is to be sent asynchronously, which means that the request will be running concurrently to the rest of the program. There will be a larger discussion of the implications of this shortly, but for now know that the program will send the request and continue running. The program will not wait for a response from the server. The purpose of this method is to respond to messages coming back from the server.

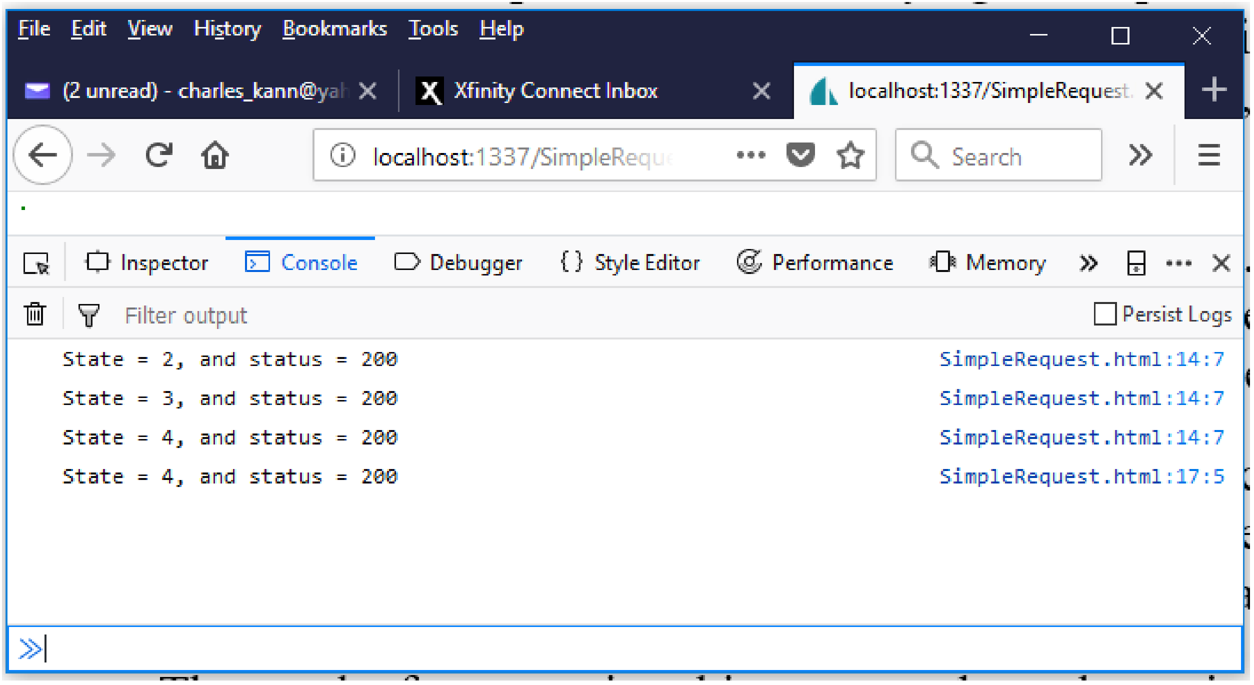
   3.1. The server will send back a number of messages to report the state of the transaction. Generally, there will be at least 3 responses to any request, a state of 2 saying the request was accepted, a state of 3 saying the request is being processed, and a state of 4 saying the request is complete, though you can receive other state information. Your program should always wait until a state 4 is received, meaning the transaction is complete, before continuing[45].

   3.2. A status will be returned along with the state. A request that completes normally will have a state of 4 and a status of 200. However, the status can be different depending on the server, and it is a good idea to check these values.

4. Once the transaction has been initialized and the callback set, the transaction can be sent to the server using the send method. In this example, the program completes, but the callback function will linger until the request is complete and the xmlhttp object is freed.

The results from running this command are shown in the following figure.

[45] For more information about request state and status, see https://www.w3schools.com/xml/ajax_xmlhttprequest_response.asp


File Edit View History Bookmarks Tools Help
(2 unread) - charles_kann@yah
Xfinity Connect Inbox
localhost:1337/SimpleRequest
localhost:1337/SimpleRequ
Search
Inspector
Console
Debugger
Style Editor
Performance
Memory
Filter output
Persist Logs

```
State = 2, and status = 200          SimpleRequest.html:14:7
State = 3, and status = 200          SimpleRequest.html:14:7
State = 4, and status = 200          SimpleRequest.html:14:7
State = 4, and status = 200          SimpleRequest.html:17:5
```

Figure 31 – JavaScript output illustrating asynchronous request and responses

Chapter 7.3.2 What it means to be asynchronous

Early it was said that this transaction was asynchronous. This has large implications for the program that are not apparent, particularly to programmers not familiar with concurrency. The following example will point out what is meant by concurrency, and why it is an important concept to understand in JavaScript.

The previous program to send a transaction to the server returned the records in the server, however there is no place in the program that displays them. This problem will now be rectified. To fix this problem, the records will be written to the console by using the responseText method of the xmlhttp object. This will contain the records current stored on the server.

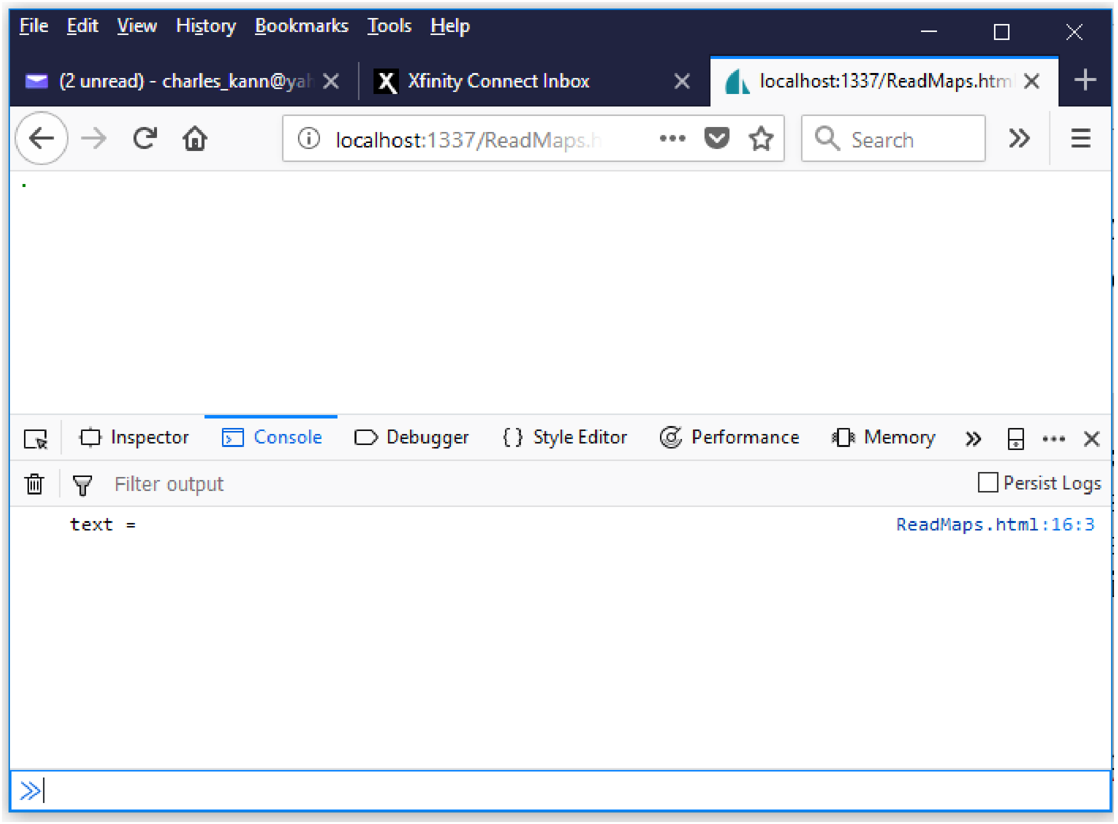
The most obvious way to do this would be to place the code to print out the data from the server after the request has been made, as in the following program. This program will not produce the expected output[46].

```
<script>
  var xmlhttp = new XMLHttpRequest();
  xmlhttp.open("GET", "mapdata", true);
  http.setRequestHeader("Content-Type", "application/json");
  xmlhttp.onreadystatechange = function ()
  {
  }
  xmlhttp.send()
```

[46] This program represents a race condition, and it is actually possible for the program to have completed the communication with the server before the output condition is executed. If this happens, the program will produce the correct results. The odds of this happening are so vanishingly small that the possibility can be practically discounted.

```
  console.log(" text = "  + xmlhttp.responseText);
</script>
```

Program 115 – XMLHttpRequest example with a race condtion



```
text =                                                    ReadMaps.html:16:3
```

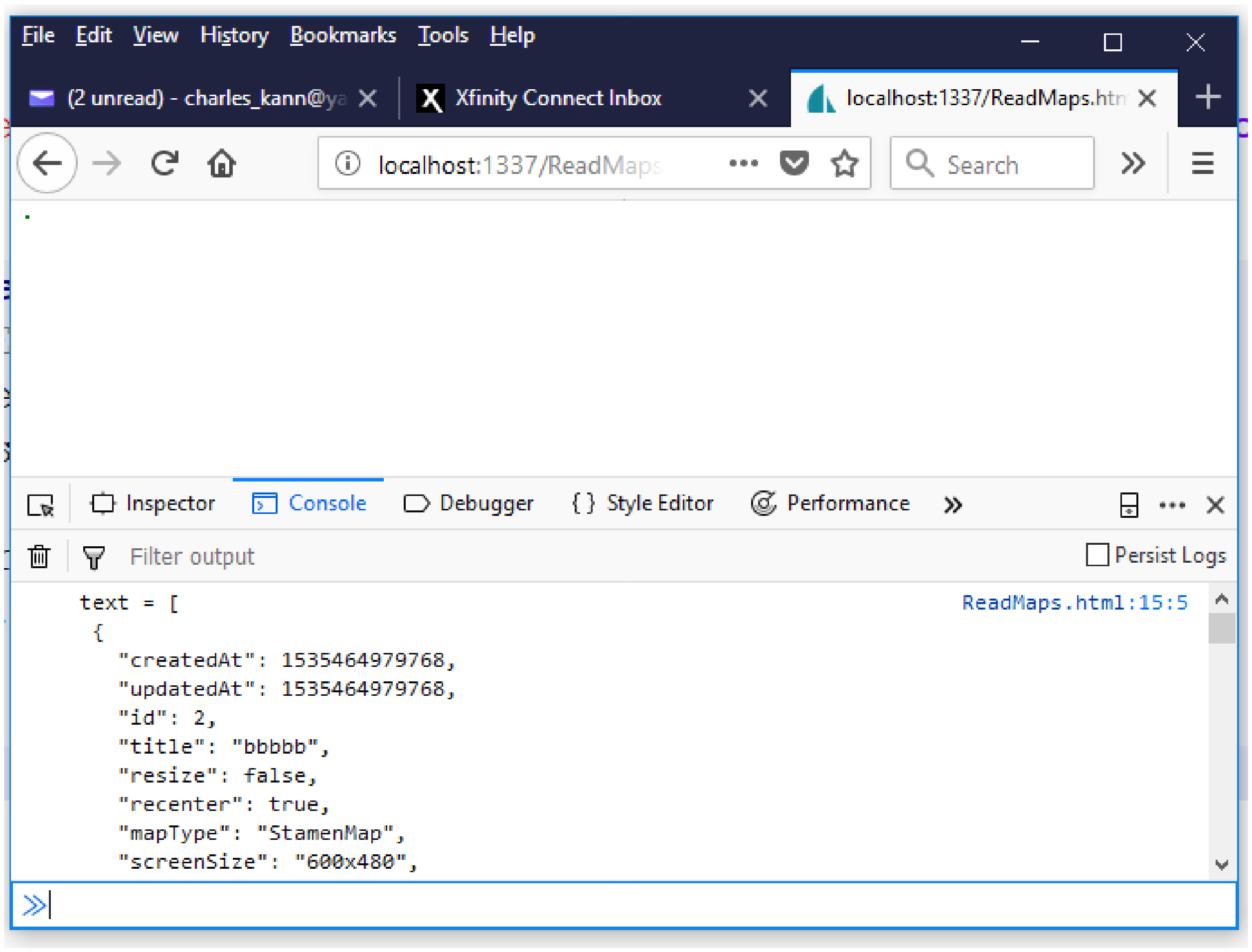
Figure 32 – XMLHttapRequest server request with a race condition

The issue is that the output to the console is occurring while the program is still processing the request, and the xmlhttp property responseText is not yet set when the console.log method is run. To fix this, the traditional answer is to place the output inside the callback so that it is only after the request has completed that the console is written to. This change is represented in the following program:

```
<script>
  var xmlhttp = new XMLHttpRequest();
  xmlhttp.open("GET", "mapdata", true);
  xmlhttp.setRequestHeader("Content-Type", "application/json");
  xmlhttp.onreadystatechange = function () {
    if (xmlhttp.readyState == 4)
      console.log(" text = "  + xmlhttp.responseText);
  }
  xmlhttp.send();
</script>
```

Program 116 – XMLHttpRequest server request with race condition fixed

Now the program will produce the expected output.



```
text = [                                              ReadMaps.html:15:5
  {
    "createdAt": 1535464979768,
    "updatedAt": 1535464979768,
    "id": 2,
    "title": "bbbbb",
    "resize": false,
    "recenter": true,
    "mapType": "StamenMap",
    "screenSize": "600x480",
```

Figure 33 – Server request with the race condition fixed

This need to put code inside of a callback can result in callback being inside of a callback inside of a callback. This situation is sometimes call *callback hell*. It is addressed in ECMA6 by the addition of *promise* and *await* operations. This will be covered in a later section.

The last change to this program is to add a check to see if the request worked correctly. If it did, an expected status of 200 will (hopefully) be returned. Otherwise the readyState 4 will have a different status code and message.

Chapter 7.3.3 Create a record

Now create the file CreateMap.html in the assets directory. This file will contain the following program.

```
<script>
  let xmlhttp = new XMLHttpRequest();
  let url = "/mapData"
  xmlhttp.open("POST", url, true);
  xmlhttp.setRequestHeader("Content-Type", "application/json");
  xmlhttp.onreadystatechange = function (){
    if(xmlhttp.readyState === 4)
    var allText = xmlhttp.responseText;
    console.log("status =  " + xmlhttp.status + " text = "  + allText)

    }
  }

  let newData = {
    "title": "map2",
    "resize": false,
    "recenter": false,
    "mapType": "Stamen",
    "screenSize": "1024x768",
    "latitude": -155,
    "longitude": 30
  }
  xmlhttp.send(JSON.stringify(newData))
</script>
```

Program 117 – XMLHttpRequest Post example

The major difference between this example and the ReadMaps.html is that now data is being sent with the request. A JavaScript object is created, and that object is serialized to a JSON object and sent along with the request to the server. This will create this new object in your server.

Chapter 7.3.4 Read, Update, and Delete

The Read, Update, and Delete operations are shown in the following three files, ReadMap, UpdateMap, and DeleteMap. These transaction look similar to the other ReadMaps and Create transactions, but now the URL is changed to include the ID of the item to be acted on.

```
<script>
  let xmlhttp = new XMLHttpRequest();
  let url = "mapData/1"
  xmlhttp.open("GET", url, true);
  xmlhttp.setRequestHeader("Content-Type", "application/json");
  xmlhttp.onreadystatechange = function (){
    if(xmlhttp.readyState === 4){
      let allText = xmlhttp.responseText;
      console.log("status =  " + xmlhttp.status + " text = "
           + allText)
      }
  }

  xmlhttp.send()
</script>
```

Program 118 – XMLHttpRequest example to read a record

```
<script>
   let xmlhttp = new XMLHttpRequest();
   let url = "/mapData/1"
   xmlhttp.open("Delete", url, true);
   xmlhttp.setRequestHeader("Content-Type", "application/json");
   xmlhttp.onreadystatechange = function (){
     if(xmlhttp.readyState === 4){
       let allText = xmlhttp.responseText;
       console.log("status =  " + xmlhttp.status + " text = "
          + allText)
      }
  }
  xmlhttp.send()
</script>
```

Program 119 – XMLHttpRequest example to delete a record

Chapter 7.4 Integrating the CRUD application with the server

The last step in developing the CRUD program to access the server is to integrate the HTTP transactions into the program. This section will do this.

Chapter 7.4.1 Changes to the application

There some minor differences between the CRUD application from Chapter 6 and the application as presented here. The differences all relate to the removal of the Save to File and Read from File buttons. These buttons were removed to better match the operations of the REST interface to the server.

The first thing the application now does when it loads is read the data from the server. There is no need to specify that the data is to be read, and the Read from File button was removed.

Next the read, update, and delete operations will immediately affect the data in the server, and not the local copy in an array of the data. All operations are therefore automatically saved, and the entire modified copy of the data in the local array does not need to be save back at the end of the execution of the program, and the Save to File button was removed.

This requires that one other change be made. When a user changes the data on the server in any way, all of the data from the server is requested and used to repopulate the list of items on the server.

Chapter 7.4.2 Populating the drop-down list of map items

The drop-down list of map items is the first GUI component that will be implemented. This component will be created and populated when the application first comes up, and each time the data server is changed, and so is abstracted as a function. The code to implement this drop-down list is as follows:

```
function readDataFromServer() {
  // clear out the list and array
  $("#dataRecords").empty();
  records = new Array();

  // get the records array from remote server
  let xmlhttp = new XMLHttpRequest();
  let url = "mapData"
  xmlhttp.open("GET", url, true);
  xmlhttp.setRequestHeader("Content-Type", "application/json");
  xmlhttp.onreadystatechange = function (){
    if(xmlhttp.readyState === 4) {
      if(xmlhttp.status === 200) {
        let arr = JSON.parse(xmlhttp.responseText);
        // Get each map, parse it, and put it in the records
        // array and list.
          for (var i = 0; i < arr.length; i++) {
            records[i] = new MyMap(arr[i]);
            $('#dataRecords').append($("<option></option>")
                  .attr("value", records[i].id)
                  .text(records[i].title));
          }
       }
       else {
         var allText = xmlhttp.responseText;
         $("#UserMessage").val("status =  " + xmlhttp.status
              + " text = "  + allText)
       }
    }
  }
  xmlhttp.send()
}
```

Program 120 – XMLHttpRequest example to load the drop-down

In the GUI code for accessing the server, note that the *value* field in the drop-down is now the id. This is hidden from the user but is how the database will be accessed.

The rest of the code for this GUI is in the file CRUDOnLoad.js file in the WebApplication directory that codes with this textbook. It is presented without comment.

```
$(function() {
    /*
        display the map
     */
    let map = new ol.Map({
      target: 'map',
      layers: [
        new ol.layer.Tile({
          source: new ol.source.OSM(),
        })
      ],

      view: new ol.View({
        center: ol.proj.fromLonLat([-77.2113732910156, 39.849999999999994]),
        zoom: 10
      })
    });


    /*
        Get records from server
     */
    function readDataFromServer() {
      // clear out the list
      $("#dataRecords").empty();
```

```
  // get the records from remote server
    var xmlhttp = new XMLHttpRequest();
    var url = "mapData"
    xmlhttp.open("GET", url, true);
    xmlhttp.setRequestHeader("Content-Type", "application/json");
    xmlhttp.onreadystatechange = function ()
    {
        if(xmlhttp.readyState === 4)
        {
            if(xmlhttp.status === 200)
            {
                 let arr = JSON.parse(xmlhttp.responseText);

                // Each record was set to JSON independently.  Get
                // each record, parse it, and put it in the list
                for (var i = 0; i < arr.length; i++) {
                    $('#dataRecords').append($("<option></option>")
                        .attr("value", arr[i].id)
                        .text(arr[i].title));
                }
            }
            else
            {
                var allText = xmlhttp.responseText;
                $("#UserMessage").val("status =  " +
                    xmlhttp.status + " text = "  + allText)
            }
        }
    }
    xmlhttp.send()
}

readDataFromServer();

/*
    Implement the ability to change options
*/

function changeOptions() {

    let center = new Array(2);
    center[0] = parseFloat(document.getElementById("longitude").value);
    center[1] = parseFloat(document.getElementById("latitude").value );

    let zoomLevel = parseInt(map.getView().getZoom());
    let maxZoom = zoomLevel;
    let minZoom = zoomLevel
    if ($("#resize").prop("checked")) {
      maxZoom = 28;
      minZoom = 0;
    }

    if ($("#recenter").prop("checked")) {
      let view=new ol.View({
      center: ol.proj.fromLonLat(center),
      zoom: zoomLevel,
      minZoom: minZoom,
      maxZoom: maxZoom,
      extent: ol.proj.transformExtent([-180, -90, 180, 90],
            'EPSG:4326', 'EPSG:3857'),
      });
      map.setView(view);
```

```
        }
        else {
          let view=new ol.View({
            center: ol.proj.fromLonLat(center),
            zoom: zoomLevel,
            minZoom: minZoom,
            maxZoom: maxZoom,
            extent: ol.proj.transformExtent([center[0], center[1], center[0],
                center[1]],
                'EPSG:4326', 'EPSG:3857'),
          });
          map.setView(view);
        }
    }

    $("#recenter").click( () => {
        changeOptions();
    });

    $("#resize").click( () => {
        changeOptions();
    });

  /*
      Function:  Delete a Map record
      Purpose:   To respond to the delete
                 button
   */
    $("#delete").click( () => {

      // get record to read from list
      let myID = $("#dataRecords").val();
      let myName= $("#dataRecords option:selected").text();

      // Record is not found, throw exception
      if (myID == null)
      {
        $("#userMessage").show();
        $("#userMessage").val("Record does not exist - Make sure  a record is
selected" );
          throw "The title does not exist";
      }

      // Confirm Delete
      let retVal = confirm("Do you want to delete " + myName + " ?");
      if( retVal == true ){
        var xmlhttp = new XMLHttpRequest();
        var url = "/mapData/" + myID;
        xmlhttp.open("delete", url, true);
        xmlhttp.setRequestHeader("Content-Type", "application/json");
        xmlhttp.onreadystatechange = function ()
        {
            if(xmlhttp.readyState === 4)
            {
                if(xmlhttp.status === 200)
                {
                    var allText = xmlhttp.responseText;
                    console.log("status =  " + xmlhttp.status + " text = "  +
                         allText);
                    readDataFromServer();
                }
                else
                {
                    var allText = xmlhttp.responseText;
                    console.log("status =  " + xmlhttp.status + " text = "  +
                         allText);
                    readDataFromServer();
                }
            }
        }
        // send request
        xmlhttp.send()
      }

    });

        /*
        Function:  Save record data from new button
        Purpose:   To respond to the New Button
   */
    $("#new").click( ()=> {

        var saveObj = readDataFromForm();
        var xmlhttp = new XMLHttpRequest();
        var url = "/mapData"
        xmlhttp.open("POST", url, true);
        xmlhttp.setRequestHeader("Content-Type", "application/json");
        xmlhttp.onreadystatechange = function ()
        {
            if(xmlhttp.readyState === 4)
            {
                if((xmlhttp.status === 200) || (xmlhttp.status === 201))
                {
                    var allText = xmlhttp.responseText;
                    console.log("succeeded - status =  " + xmlhttp.status +
                       " text = "  + allText)
                    // hide the input form
                    $("#inputForm").hide();
                    readDataFromServer();
                }
                else
                {
                    var allText = xmlhttp.responseText;
                    console.log("failed - status =  " + xmlhttp.status +
                       " text = "  + allText)
                    readDataFromServer();
                }
            }
        }

        xmlhttp.send(JSON.stringify(saveObj));
    });

    /*
      respond to the list box
    */

    $("#dataRecords").click( () => {
      // get record to read from list
      let myID = $("#dataRecords").val();
      let myName= $("#dataRecords option:selected").text();

      // Retrieve record from server
     // get the records from remote server
        let xmlhttp = new XMLHttpRequest();
        let url = "mapData/"+ myID;
        xmlhttp.open("GET", url, true);
        xmlhttp.setRequestHeader("Content-Type", "application/json");
        xmlhttp.onreadystatechange = function ()
        {
            if(xmlhttp.readyState === 4)
            {
                if(xmlhttp.status === 200)
                {
                    let allText = xmlhttp.responseText;
                    let obj1 = JSON.parse(allText);
                    let center = [obj1.longitude, obj1.latitude]
                    console.log(obj1);
                    writeDataToForm(obj1);
                    let view=new ol.View({
                        center: ol.proj.fromLonLat(center),
                        zoom: 10,
                        // This does not work and needs to be fixed...
                        //minZoom: minZoom,
                        //maxZoom: maxZoom,
                        //extent: ol.proj.transformExtent([-180, 90, 180, 90],
                           //'EPSG:4326', 'EPSG:3857'),
                    });
                    map.setView(view);
                }
                else
                {
                    var allText = xmlhttp.responseText;
                    console.log("status =  " + xmlhttp.status + " text = "
                         + allText)
                }
            }
        }
        xmlhttp.send()
    });

    /*
        Set a listener for when the map is moved.
     */
    map.on('moveend', function() {
      let center = ol.proj.toLonLat(map.getView().getCenter());
      document.getElementById("longitude").value = center[0];
      document.getElementById("latitude").value = center[1];
      return true;
    });

});
```

Program 121 – Complete Map CRUD application with server to persist data