
Doing Stuff
With
Web
Things
Lee S. Barney

DOING STUFF WITH WEB THINGS

Dedication

This book is dedicated to my wife and children. They make everything worth while. Also to those students who learn because learning is fun.

Other

Information has been obtained by Author from sources believed to be reliable. However, because of the possibility of human or mechanical error by these sources, Author, the editor, or others, Author and the editor do not guarantee the accuracy, adequacy, or completeness of any information included in this work and are not responsible for any errors or omissions or the results obtained from the use of such information.

Question and Arrow Left icons used were created by snap2objects and used under license.
Pig image used was created by Martin Berube and used under license.
Sausage image created by Daily Overview, http://www.dailyoverview.com, and used with permission.
Green check icon was created by VisualPharm and used under license.
White Zombie image used with permission of Plasmaboy Racing.
Snow goose image used with permission of Steve Perry, http://www.backcountrygallery.com.

Snow goose flock image is used with permission.
Nesting dolls image is used with permission.

READ ME. NO, REALLY!

This book is not designed as an exhaustive API or software engineering book, a book of code script-lets to copy, nor a book full of detailed definitions of every possible word. This type of information is readily available on-line. Use the dictionary in your electronic book reader, Google®, and maybe even WikiPedia for definitions that are not explicitly given in the text for unfamiliar words.

This books starts out with line-by-line explanations but gradually allows you, the reader, to learn more deeply by taking more responsibility by remembering stuff you've already learned and applying it in new situations.

The first chapter consists of a series if things you will need to know BEFORE you go on. Other chapters follow a consistent pattern.

Chapter Content Description

Each chapter consists of:

1. A brief description of the material to be covered and why knowledge of this information is important to you.
2. A series of points for consideration as you read and absorb the detailed information. These points are excellent items to discuss with co-workers or team members. Such discussions between peers yields greater depth of understanding and speeds learning.
3. A detailed and well organized description of the material to be learned. This material may have full color images and short videos interspersed within it. Each of these images and videos has been carefully created to clarify difficult or complex concepts.

CHAPTER 1

Stuff You Need to Know

The sections of this chapter present information that you, as a reader, need to understand in order to be successful in learning the remainder of the information presented. Read them and use them as the road to more learning.

Required Tools

Get and Install These

1. A laptop or desktop computer
2. The latest version of the Google® Chrome web browser

Programming tools for JavaScript, HTML, CSS and other languages can come from many sources

Many tools used to create Web applications are open source and free. You can edit and create your applications using any text editor of your choice. My preference, and the one I used to create the examples for this book, is Sublime Text™. The examples in this book use basic HTML, JavaScript, and CSS. This makes it possible to create examples that work on all browsers.

Regardless of what editor and browser you chose to work in, you will need to download and install it. For this book I suggest Sublime and either Google's Chrome or Apple's Safari web browsers.

To share with your teammates and to allow potential employers see what you know and can do, you'll use git and gitHub (http://www.github.com)

as a content management system. If you don't already have a gitHub account go create one.

Learn git. It is a skill that employers of programmers currently want to see in all potential employees. Oh... and by the way.... they want to see that you know how to use it from the command line not a GUI. Pro Git is a good resource book and is available for free in PDF, mobi, and ePub formats from the git website.

Example source code for this book can be retrieved from https://github.com/yenrab/doing_more_with_web_things.

If it is impossible for you to install a compiler on you own machine, you can use an online compiler like compileOnline. This is not as good an option as having your own compiler on your own machine.

Think First

Steps to Success

1. Think - find out what you don't know and learn it by playing with it (sandboxing).
2. Design - use what you have just learned to layout a solution to the problem.
3. Test - create a series of expectations for behavior that will indicate if you have been successful.
4. Create - write the code!

Think before you act

Throughout history, a common approach to creating things in the quickest and easiest way has been used. Why would developing software be any different? It should not.

This historic approach is cyclic. This means that you take a stab creating your item and then go back and revise one or more decisions made and then continue on.

This historic approach is composed of four simple steps.

Think - What does the customer want? What types of knowledge are required to create the thing the customer wants? Do I already have this knowledge? If not, where can I find this knowl-

edge? Playing with the new knowledge in a simple sandbox code set allows you to learn how each portion of the new knowledge behaves and can be used. Example: If you don't know what a 2X4 wooden board is and how it behaves you should not attempt to build a house. Do you think maybe you should play around with a few of boards instead of instantly starting to build? There is no replacement for experience.

Design - Decide how the thing should behave, look, and interact with other things. This is true of buildings, cars, toasters, baseball bats, and any other physical item you can think of, complex, complicated, or simple. Planning is vital to software development. How can you create an application if you have not yet decided what it is going to do, what it will look like, and how it will interact with other pieces of software or hardware?

Test - In this step you decide what standards your product will meet when it is done. For buildings this step would be the national and local building codes. In software this may consist of User Interaction testing, Unit testing, Component testing, System testing, Installation testing, or just figuring out, at a detailed level, what new data should come out of your application when it is given specific sets of data. You may be thinking, "How can I create a test for something that doesn't exist yet? I don't know how it should work." If this thought or one like it passes through your mind, then you have not sufficiently completed step 1, 2, or maybe both. **Go back and do them first.**

Create - Begin building. Notice that this step description starts with begin. Once you have started, you WILL find weaknesses in your knowledge, design, tests, or all of these. When you do, go back and revise your results for steps 1 - 3 as needed. This does not mean 'throw away and start all over again'. It means make modifications and try again.

Instead of following this time honored and good approach what most programmers do goes like this:

1. A customer asks for something.
2. The programmer starts to create it.
3. It doesn't work.
4. The programmer throws it out.
5. The programmer repeats step 2 until it works.
6. The company ships something the customer didn't want.
7. The programmer starts over again, if lucky, or has to find another job.

I call this the 'oh crap!' or 'think last' approach since nearly every time the developer hits step three 'oh crap!', or words like them, are heard. The 'think last' approach all but guarantees that the software ships late, is over budget, and doesn't have the features the customer wants. It is a proven failure method. Don't use it! Use the method that is known to work. **Think, Design, Test, Create**. Until you use it it may seem that it will take longer than the 'Oh crap!' method but it does not. Guaranteed.

Helpful Resources

Steps to Success

1. Install the tools found in the Required Tools section of this chapter.
2. Download the Examples
3. View a web page on your computer in a web browser.

Books such as this one are not your only resource. This section has some resources that you will find helpful as you start creating stuff using HTML, JavaScript, and CSS.

The examples for this book.

Movie 1.1 Downloading the example files

Movie 1.2 How to see your page in a browser

The Basics

Lorem Ipsum

1. What is the purpose of HTML?
2. What is the purpose of CSS?
3. What is the purpose of JavaScript?
4. How do I create a simple page template I can reuse?
5. What is a page?
6. What are the major parts of all HTML pages?

Using HTML, CSS, and JavaScript allows you to make powerful web experiences for your users

There are three things used to create modern web pages. HTML, CSS, and JavaScript. Each of these has its own reason for being. HTML is used to create the things you can actually see on the screen. If you've seen a picture, a button, some text, or watched a video in your web browser HTML was used to show it.

If you've seen pages that weren't completely black and white, that's because CSS was used to 'fancy it up.' Lots of the stuff you are used to seeing on the web is the result of someone applying CSS.

If you've ever been at a website, clicked on something, and what you were looking at changed, that was the result of someone using JavaScript.

JavaScript is what makes the page active and responsive instead of something that just lays there.

So each of the three parts of good web pages have their responsibility.

HTML - define the stuff to be shown to the user.

CSs - make the stuff shown to the pleasant to see.

JavaScript - make the user experience interactive rather than passive.

Don't forget the purpose of each of these tools. If you do, things will get really confusing really fast. If you do forget, come back here and read this again.

When you write HTML, CSS, and JavaScript you do it in what techies call ***a page***. All it really is is just a file with text in it, but you know how techies just love to make up new lingo. All HTML pages you write have the same basic structure as a human; a head and a body. I've put together a very simple page that you can use as a template. I call the file 'template.html.' It is part of the zip file download from gitHub. The template.html Code Sample shows what's in template.html.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Title of Your Page</title>
5     <script>
6
7     </script>
8
9     <style>
10
11     </style>
12  </head>
13
14   <body>
15        <p>Hello</p>
16   </body>
17 </html>
```

Code Sample - template.html

Details:

Let's go over this line by line. Believe me, it will help. Line 1 tells the web browser that is displaying the page that all of the other lines are HTML, not some older HTML version. Don't worry too much about the details of line 1. Converted to English all it says is, "The type of this document is HTML."

Line 2 says, "Hey web browser, here is where the actual HTML stuff starts." Line 2 has a sibling on line 17. Line 17 says, "Hey web browser, here is where the HTML stuff ends." In other words everything you will be writing when you create a page goes between <html> and </html>. By the way, anytime you see anything starting with < and ending with > it is called a ***tag***. Yep...another techie word. Line 2 consists of a starting HTML tag. Line 17 is a closing HTML tag. You can tell a closing tag because it has a / right after its < character and before its name.

Remember how each HTML page has a head and a body? Well if you look on line 3 you'll see the starting head tag, <head>. Look at line 12 and you'll see the head's closing tag, </head>.

"OK," you say. "that explains the head but what about the body?" Excellent question. The body's starting tag is on line 14, <body>, and its closing tag is on line 16,</body>.

Wow...that's a lot to digest in a short amount of time. Stop and take some time to go back over that again. Seriously...please stop. Don't read on until you've spent some time pondering the tags you've seen so far. Its just fine to spend a day pondering at this point. We'll pick this up again and go on when you're ready.

More Details:

All right. Let's take a look at some more tags. If you didn't notice, the template.html Code Sample has tags between the starting and ending of other tags. Techies call this 'nesting tags' in other tags. Nesting??? Really??? Why call it nesting? Well techies don't usually create new words. Mostly they just reuse existing words that have obscure definitions. If you look in a dictionary you will find the primary definition of nesting has to do with birds just like you expected. But it isn't the only definition. There is another.

It defined nesting as putting or fitting one thing inside another--Russian nesting dolls are one example of using the word this way.

Techies grabbed onto this meaning of the word and used it to describe when one HTML tag is between the starting and ending tag of another one. If you look at line 4, you will see a title tag *nested* inside the head tag. That's the techie speak. Regular people might say the title tag is between the head's starting and closing tag. It all means the same thing.

Now that we know that line 4 is 'in the head', we should figure out what it does. It reads, "Put the text 'Title of Your Page' in the tab of the web browser. "Wait a minute." you say, "I have no idea what you just said!" OK. Hang in there with me. I'll show you a picture. It'll be worth a thousand words, right?

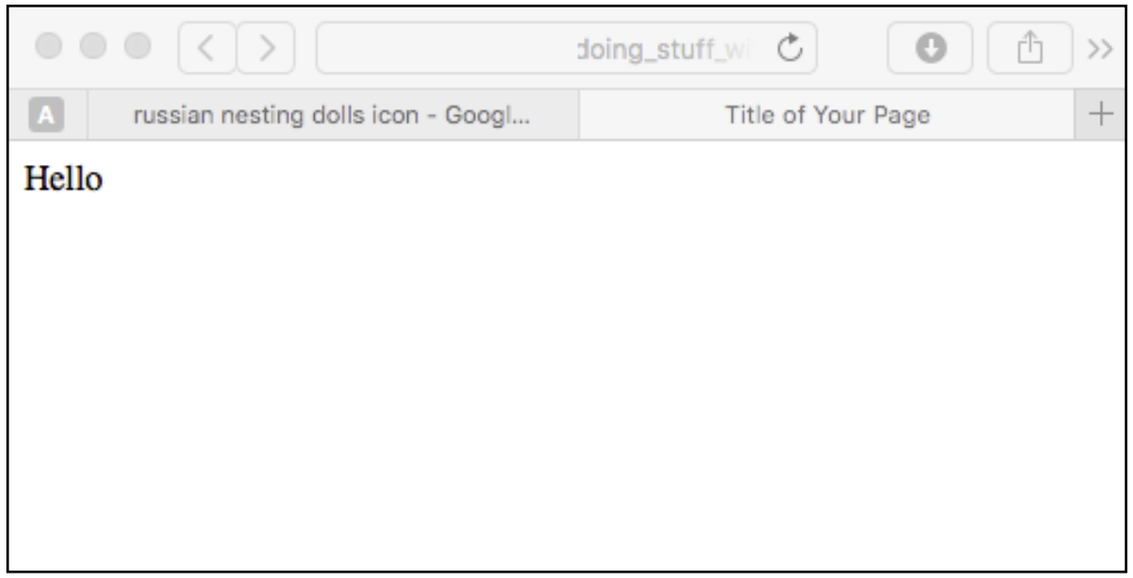


Figure 2. *The template.html file displayed in my web browser.*

Figure 2 shows what happens in the browser when you double click the template.html file. My web browser opened it and read it. If you look at the tabs near the top of the browser, you'll see "Title of Your Page." That is exactly what line 4 told the browser to do. Wow! Now you know how Google® and all the other techies get messages to show up on your web browser's tabs.

Congratulations! You've just seen there is no magic in computing. Everything you see online is because someone or something put a tag in a page. Everything you learn from now on flows from this realization. Want to show your user something? Put a tag in the your html file. That's how it's done.

We're going to look at template.html some more. Just so you don't have to go so far back in the book I'll put it here again.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Title of Your Page</title>
5     <script>
6
7     </script>
8
9     <style>
10
11     </style>
12  </head>
13
14   <body>
15        <p>Hello</p>
16   </body>
17 </html>
```

Code Sample - template.html

More Details:

Line 5 is a starting script tag and line 7 is its ending tag. When you start writing JavaScript, it is between these tags where you'll put you JavaScript code.

Line 9 is the starting style tag and line 11 is its ending tag. This is where the CSS you'll write will eventually go. We won't worry about the script or style tags right now. We'll go in-depth on them later. Let's go on and look at the other major part of every HTML page, the body.

As with everything else in HTML, the body consists of tags. You can see the body's starting tag on line 14. Notice it is between the starting and ending html tags just like the head tags were. It also comes right after the head's ending tag. This is NOT a coincidence. This is where the body should start in all HTML pages. As with the other tags we've looked at, the body has and ending tag. You can find the ending tag on line 16.

Hang in there, we've only got one more tag to learn for now. You can see it on line 15. There you'll see as a starting tag and as an ending tag. There is no special reason why the starting and ending tag are on the same line. It was my personal preference. You can put them on different lines if you like.

But what does ***p*** mean? Line 15 converted into English reads like this, "Put a paragraph in the body that has the text 'Hello' in it." Ahh! p stands for paragraph. "Wait a minute," you say. "Why are we aren't we using complete words anymore?" Excellent question. The only answer I can give you is that programmers are lazy.

Since programmers type a lot on any given day, they want to type as little as possible. It's a lot easier to type p than 'paragraph'. You just have to remember that anytime you see or you are dealing with a paragraph. It's a good idea to put text you want the user to see in paragraph tags. Just do it. It will make your life easier down the road.

Congratulations! You've made it through your first experience with HTML. Now you have a template that you can use every time you start to create a new page. If there was something you didn't understand, go back over that part again. You should also talk about what you do and don't understand with someone. Even if they aren't a techie, just talking about this kind of stuff can trigger things in your brain and help you understand. You should also ask techies you trust questions about what you do and don't understand. There is only one worse way to learn something than talking to people face-to-face....every other way.

CHAPTER 2

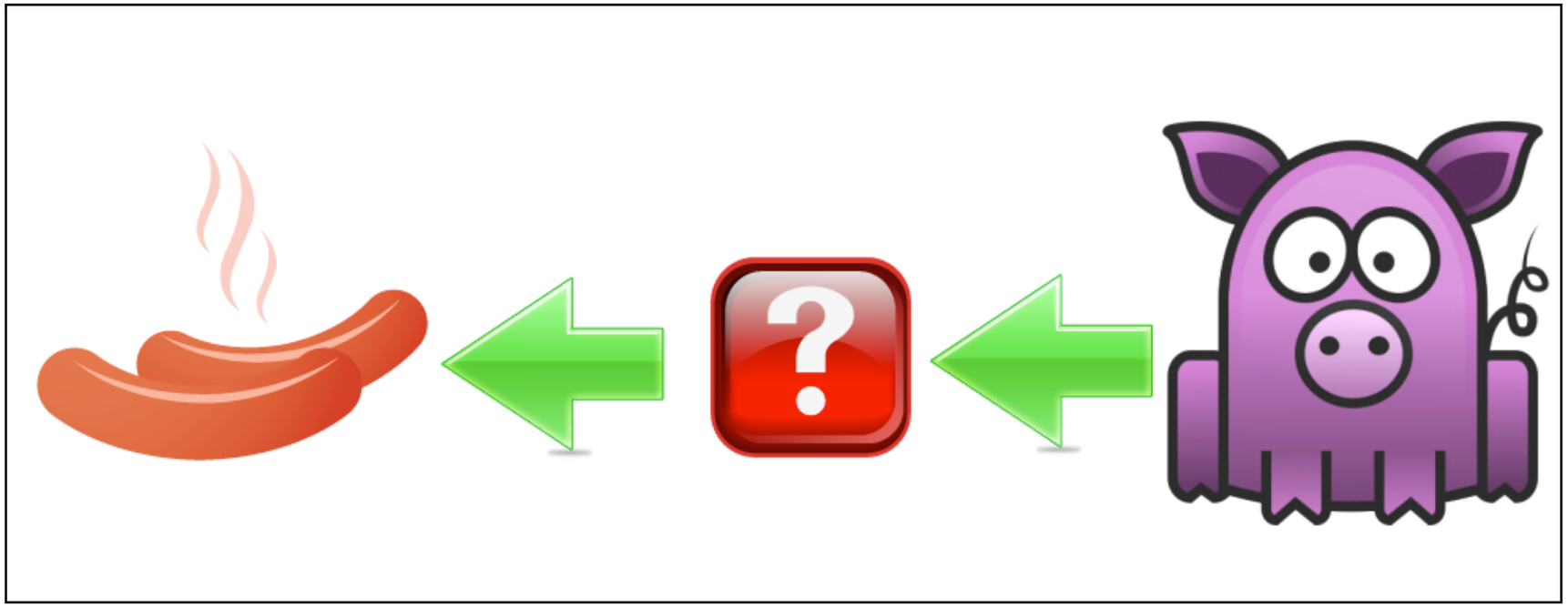
Ready...Set...Go!

White Zombie is the worlds fastest, electric, street legal, drag racing car. It REALLY goes. Some of it is custom assembled but not all. The body, frame, tires, wheels, and other components were created by people other than Plasmaboy racing. Pages you write are built the same way. Some parts are pre-made for you and others you will assemble yourself from pre-made parts.

Functions and Events

Points to Ponder

1. What is a function and what is its purpose?
2. What do functions look like in the JavaScript language?
3. What is an event?
4. How do I execute a function when the user clicks a button?



Functions are one of the basic building blocks of all pages written using HTML, JavaScript, and CSS. Stuff goes into a function and stuff comes out. A sausage factory is a physical manifestation of a makeSausages function.

When you say you are writing a page that uses JavaScript, what you are really saying is that you are creating a bunch of operations. You are telling the person you're talking to that you're going to arrange operations so you can effectively communicate. With who??? The user of your application.

Do NOT fall into the common error of thinking your application will talk to the computer. The computer is the medium of your communication with the user. Air is the medium when you use English or some other language to speak with someone. The computer is the medium when you write pages.

Functions are a type of operation. They are one of the foundations of everything you will do when you use the JavaScript language. We'll start with them. After all, what fun is a page that doesn't do anything?

Answers to the questions for this section will be provided as you go along and the details of functions are be examined. Hold on to your hat. At times you may feel like a pig in a sausage factory, but I promise, you won't get ground up and spit out.

The grammar of the JavaScript language, called 'the syntax' by techies, declares that functions all have the same structure. They have a name, stuff that goes into the function, stuff that comes out of the function, and stuff that makes up what the function does.

For a pork sausage factory function the structure would be

1. a name - makePorkSausages,
2. stuff that goes in - pigs,
3. stuff that comes out - pork sausages, and
4. stuff that makes up what the factory does - chopping, filling, and a bunch of stuff no one really wants to know about.

Take the time right now to start thinking of your everyday tasks. Each of them is describable as a function. An example is taking a shower or bath. You, soap, shampoo, water, a tub or shower, and the grime you have built up on you over time goes in. What comes out? What do you do that makes up what the functions of taking a bath or shower does?

If you look at what you do on any given day, you'll find functions everywhere. I strongly advise you start doing this NOW. In every spare moment you have, look around you. See what people, animals, buildings, cars, tools, bugs, plants, clouds, etc. are doing. Truly look and think deeply.

Give what you see going on a name. Figure out what, if anything, is going in, what, if anything, is coming out, and what may be happening as part of what is being done. Please write down what you see. It can help you later when you get confused. If you start examining your world now in every spare moment in this way, learning the JavaScript language and building pages will be MUCH easier.

Before reading this book any further go do this for at least two days. ***STOP HERE. PLEASE.*** Don't make your life harder than it has to be.

Where To Begin?

Now that you've spent a couple of days looking for functions and their parts, let's start up again by creating a web page that changes when a button is clicked.

Since you are working in an HTML page, you guessed it, you need a tag to show a button to the user. Thankfully the people who created HTML called this tag 'button.' Yeah! A rational choice!. A button tag looks like this; <button>Click Me!</button>

The english for the tag is "Show a button that has 'Click Me!' on its surface for the user to see." Based on what you learned while reading The Basics, you know that this tag belongs inside the body tag. To do this you start by duplicating the template.html file in the gitHub zip download. You can do this in a bunch of different ways. Here is one.

1. Go to where template.html is on you computer.
2. Select, by clicking once, the template.html file.
3. Copy the file.
    a. Windows - hold down cntrl and c at the same time.
    b. Mac - hold down command and c at the same time.
4. Paste the copied file.
    a. Windows - hold down cntrl and p at the same time.
    b. Mac - hold down command and p at the same time.
5. Rename the copied file to be first_event.html.
    a. If you aren't sure how to rename a file, google "file rename" and add in the name of your operating system, windows, os x, linux, or what ever operating system you are using, to your search.

Go ahead and remove the paragraph tag from the body. Then add in the button tag. Remember, it looks like this.

<button>Click Me!</button>

After you are done adding the button, change what's in the title tag to be "First Event" and then save.

Now click the first_event.html file. You will see something that looks like this.

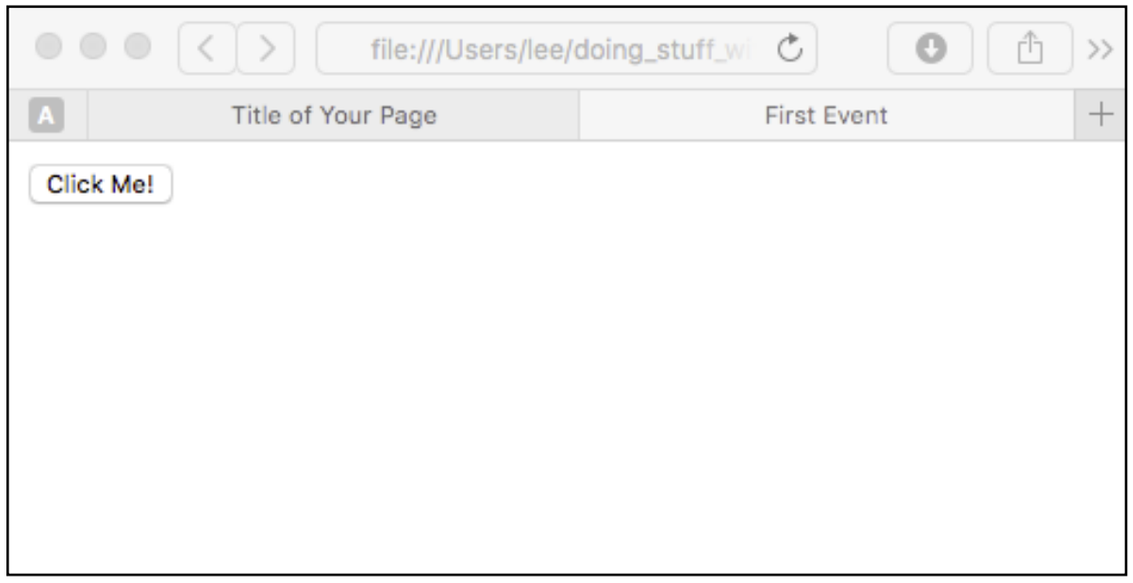


Figure 1. *The first_event.html file displayed in my web browser.*

You've got it working? Great!

You see...It's just like we talked about back in the basics section. When you're creating an HTML page and you want to show the user something, add a tag.

The only problem we've got is, no matter how many times you click the button, nothing happens. Hm...that's no fun. You want this page to do something, right? Let's have the page show a message when the button is clicked. Why? No special reason. Just because we can.

From this point on, it will be easier for you to understand if I show you, step-by-step what to do. At this point, if you followed my instructions, your first_event.html page should look like this.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>First Event</title>
5     <script>
6
7     </script>
8
9     <style>
10
11    </style>
12  </head>
13
14   <body>
15       <button>Click Me!</button>
16   </body>
17 </html>
```

Code Sample - first_event.html

Got it? Good. You want to show the user something. That means.....yep you got it! You need to add another tag. You're going learn an HTML tag you're going to use a LOT in this book. There's a whole bunch of different tags you can use. You can experiment with them later. In this book you'll stick with this one to show the user stuff. This important tag is the section tag. It represents an area, or chunk of the page when its displayed in your web browser.

Like the other tags we've looked at, it has a starting tag <section> and an ending tag, </section>. Let's put it in your page. It should go right AFTER the button tag and look like this.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>First Event</title>
5     <script>
6
7     </script>
8
9     <style>
10
11    </style>
12  </head>
13
14   <body>
15       <button>Click Me!</button>
17       <section></section>
18   </body>
19 </html>
```

Code Sample - first_event.html

For now, don't put anything between the section's starting and ending tags. OK...now you've got all the tags in place to make some text show up when the button is clicked.

If you remember what we talked about back in The Basics, the purpose of JavaScript is to "make the user experience interactive rather than passive." It must be time to add in some JavaScript. :) Back then, I also said that the JavaScript would go between the script opening and closing tags.

Like I said earlier, the JavaScript language's grammar, what techies call its syntax, says a function must have 'stuff that goes in', 'stuff that it does', and 'stuff that comes out'. Because computers are so stupid you also have to tell the computer that you are writing a function. I know...I know... that seems silly but it's just the way it is. Each function also needs a name.

All functions in JavaScript follow the same grammar rule. They all look like this.

function theFunctionName(whatGoesIn)

I'd suggest always giving a function a name that is a verb. Function names should always describe what the function is supposed to do. Please make your function names be verbs. It will make your life so much easier. For this first function, you want to show a congratulations message. Because of this, call the function showCongratulations. See? The function's name is a verb. When you add it to your first_event.html page, it will start out looking like this.

function showCongratulations(theButtonClicked)

Now...there are some strange things here so lets take a look at them.

One is how you've done capitalization in the function name. "What's up with having the second word be capitalized and not the first???" Good question. It is very 'JavaScripty' to start all of your function names with a lower case letter. It's the way it has been done for decades. It works if you choose to capitalize the first letter, but when someone reads your code they'll wonder if you know how to play well with others. The accepted practice is to start all function names with a lower case letter. Please follow this practice.

"OK," you say,"but what about that silly Capital letter in the middle of the name. You haven't explained that yet!" The reason you do that is so it is easy to see where the second word starts.

function showCongratulations(theButtonClicked) is much easier to read than

function showcongratulations(theButtonClicked). Especially when you're reading hundreds or thousands of lines of code.

It is so much easier to read that programmers have a name for this type of capitalization. They call it 'camel case.' Camels have humps in the middle of their backs, and if we capitalize the first letter of each new word in a name, except the first, then the name has humps in the middle to. It's a silly name, but when the first programmer made up the name they probably thought it sounded laugh-out-loud funny. There's no accounting for taste, right?

The other strange things you see are the parentheses, (), after the showCongratulations function's name. These parentheses are used to hold the 'stuff that goes in'. The showCongratulations function has the button that was clicked going in so it is between the parenthesis.

All functions in JavaScript follow the grammar rule we used here. Like I said before, all JavaScript functions look like this.

function theFunctionName(whatGoesIn)

You can depend on this rule always being true. What you haven't seen yet is 'the stuff that makes the function do what it does'. We'll add that in now.

```
function showCongratulations(theButtonClicked){
      document.getElementById(“result_display”).innerHTML =
            “Congratulations, you did it!<p>You got the page to update!</p>”
}
```

Code Snippet 1

Details:

In the JavaScript language’s grammar, the characters { and } are used to contain ‘the stuff that makes the function do what it does.’ They are used to help us humans see, understand, and remember what this function does compared to other functions that may be hanging around in the same page.

Great debates still swirl around the internet about exactly where is the ‘best’ place to put the { and } characters in spite of the debate being over 40 years old. Forty years is WAY too long for debating such an unimportant thing. On the internet and in books you will see examples with these characters in different places based ONLY on the author’s opinion. All of the examples you will see from me will follow the pattern you see in Code Snippet 1.

Now that that’s out of the way, lets examine each of the parts that make up what the showCongratulations function will do when the function executes.

If I read what you see between the { and } characters to you in English, it would sound like this, “Hey document, go find the element who’s id is “result_display”. After you find it, set what is between its starting and ending tags to be “Congratulations, you did it! You got the page to update! .”

Compare what you see in the function with the English you just read. You’ll need to compare these multiple times. When you do, you’ll start to see how JavaScript, and almost all other modern programming languages, are just simplified, condensed English! Compare them again. Read them both out loud.

Here is the way a programmer would read the JavaScript.

“Document dot get element by id, result display, dot inner HTML is equal to “Congratulations, you did it! You got the page to update! .”

Here is how an English speaker would say the same thing.

“Hey document, go find the element who’s id is “result_display”. After you find it, set what is between the starting and ending tags to be “Congratulations, you did it! You got the page to update! .”

These two readings mean exactly the same thing. “Well,” you ask, “if they mean the same thing then why isn’t JavaScript just regular English?” Good question. The answer is, computers are stupid. Have you tried using the voice commands on your phone recently? Programmers have tried for decades to get computers to understand regular old English. Computers are still too stupid to be able to understand it consistently. Believe me, you don’t want the applications you use every day to be as inconsistent as your phone’s ‘personal assistant’ and programmers live and die by how consistent their applications are.

Anyway...get used to what document.getElementById means. You’ll be using it in every one of the pages you create and in all the examples for this book. Just remember what it does. Remember it in regular old English.

Now put the showCongratulations function in between the script tags in your first_event.html page and give the section an id of 'result_display'. When you do you'll get something that looks like this.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>First Event</title>
5     <script>
6               function showCongratulations(theButtonClicked){
7                       document.getElementById("result_display").innerHTML =
                                                "Congratulations, you did it!
                                                 <p>You got the page to update!</p>"
8       }
9     </script>
10
11     <style>
12
13     </style>
14  </head>
15
16   <body>
17        <button>Click Me!</button>
18        <section id="result_display"></section>
19   </body>
20  </html>
```

Code Sample - first_event.html

Since you have to tell the computer the id of the element you want found, you are responsible for there being an element that has that id. If you forget to add an id or send getElementById the wrong id, you're going to get a strange error when you click the button. The error will mention ***null***. This is a common beginner mistake. Try and avoid it but don't get frustrated when it happens. Even those of us who've been writing this stuff for a long make this mistake every once in a while.

When your page matches what you see above, save your version of the first_event.html page and load it in your browser by double clicking the file. Then go ahead and click the button. "It still doesn't do anything!" you say. Yep...that's because we're missing one huge idea and one tiny change.

You've put tags in first_event.html that display stuff like buttons and text to the user. You've also put JavaScript into the page to make the changes to the page you want. What you haven't done is link the two together. To do this, you're going to need to understand events.

Page Events

Humans tend to record and remember our lives by using events. We remember births and birthdays, deaths, falling in love, marriages, jobs we've been offered or lost, career changes we've made, and a whole bunch of other life events. These large events are not the only ones we experience. Small ones such as eating a meal, while not necessarily as memorable, are just as important. Don't think so??? Consider what it would be like to go without eating for a month, a week, or even a day.

As humans we also observe external events like sunrises and sunsets, rain storms, earthquakes, and the blossoming of flowers. The sum total of all of the human, internal and external events make up what we do and don't remember of our lives; they give us experiences.

Computers are designed and built by humans. Since we have such a strong ability to see and react to events it shouldn't be surprising that the computers we make are controlled and driven by events. Each time you press a key on a keyboard, click a mouse, touch a screen, or speak to a computer you generate an event. HTML takes advantage of this.

We aren't going to look at the large number of events available in HTML in this book. Instead we'll stick with the ***onclick*** event. There

is nothing wrong with the other events. In fact I strongly encourage you to play with them and find out what they are and when they happen. We'll stick with just ***onclick*** so you don't get over-whelmed.

Details:

To link the button on line 17 to the showCongratulations function on lines 6 - 8, all you need to do is tell the button what to do when it is clicked; in other words, when an ***onclick*** event happens. The link to make this happen is done by modifying line 17. Change it to like this.

```
<button onclick="showContratulations(this)">Click Me!</button>
```

In English the modified line 17 reads, "Each time the 'Click Me!' button is clicked, execute the showCongratulations function and send in the 'Click Me!' button to the function." On this line, the ***this*** word represents the 'Click Me!' button itself.

You can see the completed version of first_event.html below. It is also available in the zip file download from the gitHub repository.

```
1  <!doctype html>
2  <html>
3    <head>
4      <title>First Event</title>
5      <script>
6              function showCongratulations(theButtonClicked){
7                      document.getElementById("result_display").innerHTML =
                                                    "Congratulations, you did it!
                                                     <p>You got the page to update!</p>"
8      }
9    </script>
10
11     <style>
12
13     </style>
14   </head>
15
16   <body>
17       <button onclick="showCongratulations(this)">Click Me!</button>
18       <section id="result_display"></section>
19   </body>
20 </html>
```

Code Sample - first_event.html

There you have it. You've seen all the basic parts of a function and a page. You've also seen how to get the page to change based on an event. All of other functions you'll ever write using the JavaScript language will have the same parts as this one. They'll follow the same grammar and they'll be executed due to events.

This is all fine and dandy but still pretty boring. Let's add in something else. Let's change what the button says after it has been clicked once.

To do this you only have to add one line of JavaScript. If you take a look at the button_change.html code sample, available below and in the zip file for the book, line 8 has been added and the title of the page has been changed.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>First Event</title>
 5     <script>
 6               function showCongratulations(theButtonClicked){
 7                         document.getElementById(“result_display”).innerHTML =
                                                  “Congratulations, you did it!
                                                   <p>You got the page to update!</p>”
 8                         theButtonClicked.innerHTML = “Click Me Again!”
 9      }
10     </script>
11
12     <style>
13
14     </style>
15  </head>
16
17   <body>
18        <button onclick=”showCongratulations(this)”>Click Me!</button>
19        <section id=”result_display”></section>
20   </body>
21 </html>
```

Code Sample - button_change.html

Details:

Since button and section are both tags, you can change what is in between their starting and ending tags using the same thing; innerHTML. Hey! Similar code yields similar behavior! Yeah! The old guys with beards that created JavaScript got this one right. :)

The line number **8** says,”Change what is in between the clicked button’s starting and ending tag. Set it to be ‘Click Me Again!’”

With the code like this, when you double click the button_change.html file you will see “Click Me!” on the button. After you click the button, in the result_display section you will see “Congratulations, you did it! You got the page to update.” The button will change too. Inside it you will see “Click Me Again!”

So far you’ve created a function, sent in to it the button that got clicked, changed what’s inside the result_display section and what’s inside the button that got clicked...lots of function stuff.

Now, not leaving functions behind, let’s move on to something else, CSS. After all, button_change.html is pretty ugly.

Make it Nice

Points to Ponder

1. What is the purpose of CSS?
2. What is a CSS class?
3. How do I control the color of a portion of a page?
4. How do I control the size of an element?
5. How do I control where on the page an element is shown to the user?

A world without color, shading and depth of field would be awfully boring. Everything in black and white and no visible distance? Yuck.

What we did in Section 1 works but it looks terrible. Maybe you'd call it utilitarian. It does the job you wanted it to, but it's not nice to look at.

CSS and its earlier versions let us change things up. There are a lot of things we won't cover in this book. Instead we'll focus on just a couple of basics; color, width,height, and position. Focusing on these will get you started in a good direction. You can explore the others later.

In Section 1 you learned that JavaScript is placed inside script tags in the head. This time You'll be working in the other tag mentioned in The Basics--the style tag. Yep. In no time at all

you'll be coding with style. :) I know...I know. Bad pun. I just couldn't help myself.

When you want to change the style of your page, what it looks like, you put stuff inside the style tags. You won't be putting tags in there. You didn't put tags inside the script tags either. Instead, you'll put descriptions of how you want specific things to appear inside the style tags. That is what we call CSS.

Each description begins with what it is you are describing. It might be a specific HTML element or it may be a group of them. Either way, you always begin by saying what you are describing.

You remember the body tag from the last Section? If not, go back and look at it again. You can create some CSS that effects everything inside the body tag. Let's give it a shot.

Make a small change to the button_change.html page and save it as fancy_button.html. Add what you see on lines **13** - **15**.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>First Event</title>
 5     <script>
 6               function showCongratulations(theButtonClicked){
 7                         document.getElementById("result_display").innerHTML =
                                                  "Congratulations, you did it!
                                                   <p>You got the page to update!</p>"
 8                         theButtonClicked.innerHTML = "Click Me Again!"
 9      }
10     </script>
11
12     <style>
13               body{
14                    background-color:DarkSlateBlue;
15               }
16     </style>
17   </head>
18
19   <body>
20       <button onclick="showCongratulations(this)">Click Me!</button>
21       <section id="result_display"></section>
22   </body>
23 </html>
```

Code Sample - fancy_button.html

If you make this change and save it, you'll discover the white background of you page is gone. Instead it will look like this.

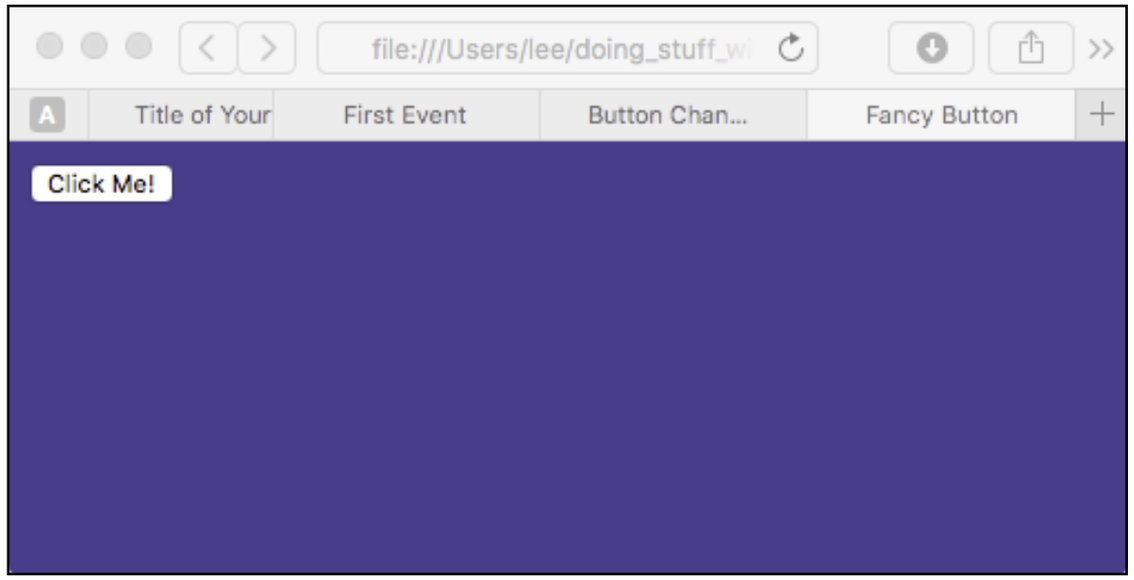


Figure 1. *The first change in fancy_button.html in my web browser.*

Now, I don't claim to be an expert at color schemes, but that background color is certainly not white. It is a color known to the web world as DarkSlateBlue. There are a whole lot of colors available to use that are pre-defined. White, black, yellow, green, blue, red, and many variations of these in color and hue. DarkSlateBlue just happens to be the one I've chosen for this example.

Details:

Let's read lines **13** - **15**, of fancy_button.html. They go like this, "Find all the body tag pairs and set their background colors to be DarkSlateBlue."

"Wait a minute Barney," you say. "There's only one body tag pair. What's up with that 'all the body tag pairs' nonsense?" Another excellent question on your part!

You are right. There is only one body tag pair in a well written page but there might be multiple buttons...or multiple sections...or multiple instances of other kinds of tags. Then this ability to change how all of them look with one little bit of CSS will come in very handy. If you list the name of a type of tag then the next bit between the { and } characters applies to all tags in that group. Back at the beginning of this section I told you you'd see how to effect groups of things. This is one way you can do that.

All right...go ahead and click the button. That black text is a little hard to read isn't it? Those in the business would say there isn't enough contrast between the text and the background color.

Now, we could change the background color back to white and there would be great contrast but who wants to live in a black-and-white world? Let's fix it another way. Let's change the color of the text.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>First Event</title>
 5     <script>
 6             function showCongratulations(theButtonClicked){
 7                     document.getElementById("result_display").innerHTML =
                                                 "Congratulations, you did it!
                                                  <p>You got the page to update!</p>"
 8                     theButtonClicked.innerHTML = "Click Me Again!"
 9      }
10     </script>
11
12     <style>
13             body{
14                 color:white;
15                 background-color:DarkSlateBlue;
16             }
17     </style>
18 </head>
19
20   <body>
21       <button onclick="showCongratulations(this)">Click Me!</button>
22       <section id="result_display"></section>
23   </body>
24 </html>
```

Code Sample - fancy_button.html

Take a look at line **14**, it is a little confusing. You'd think it would say 'text-color' or even 'font-color' but it doesn't. This is because of the long history of web pages. Way back in the day, the 1990's, there was a time when the web was new and the only color you could change was the color of the text. There were no pictures, no Google, no entering of data. Only text and colors of text. You couldn't even change a font! It was primitive, but that is why the keyword color is used to change the color of text. It's a hold over from those primitive times. Since you could only change the color of the text why waste everyone's time by making them type 'text-color' or 'font-color'? You see the logic?

Let's read line **14**. It says,"Set the color of all text in the body to be white." When you make these changes, save the file, and double click it, this is what you see.

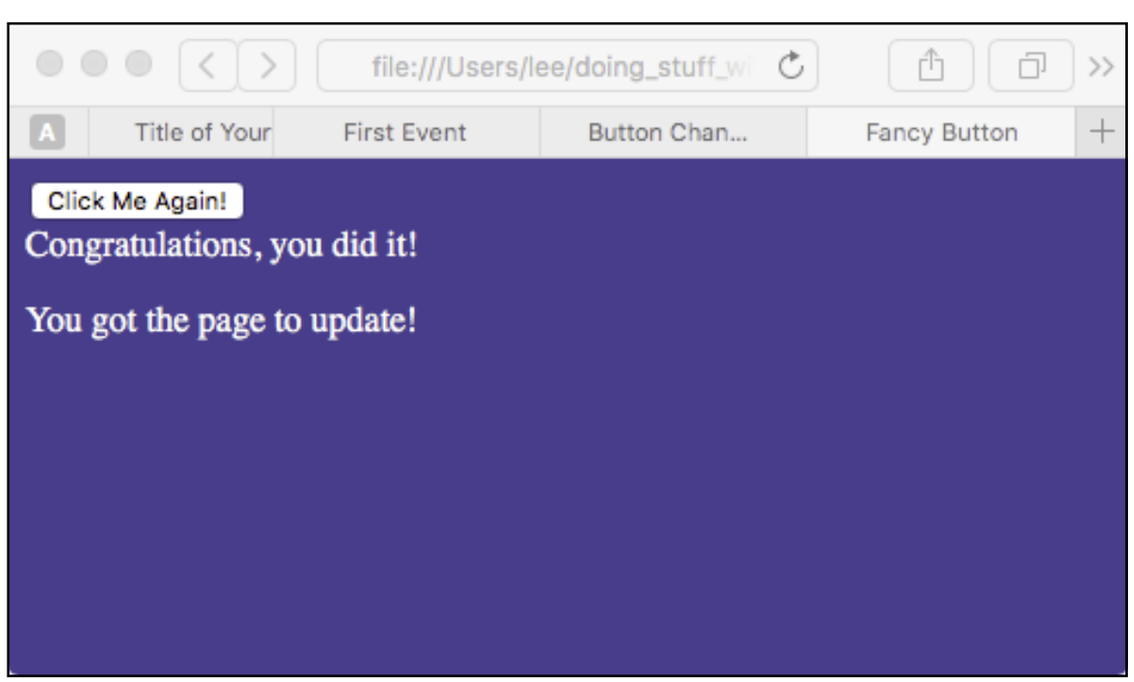


Figure 2. *The text color change to fancy_button.html in my web browser.*

That's lots easier to read but still not very fancy. Be patient. We'll gradually add in a few more things. It'll end up showing you how those super fancy pages are built up using small and simple things. Let's play around with how the the section tag is shown.

Changing Information Displayed

This time you're messing with a single tag pair. What you're going to do you don't want to apply to all section tag pairs. You may have added a whole bunch to the page. Let's just make the change apply to one section. From Section 1 you know you can us an id to find a specific tag using JavaScript. You can apply this same principle in CSS. Way to go CSS creator guys! That make your job easier.

Instead of starting your next piece of CSS with section, which would find all of the section tag pairs, start it with '#result_display'. The # character is CSS's way of saying findElementById. That's really nice. Lots less typing. So let's try this...find the result_display section and set it's background color to DarkGrey. When you do, the code will change to look like this.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>First Event</title>
 5     <script>
 6               function showCongratulations(theButtonClicked){
 7                         document.getElementById(“result_display”).innerHTML =
                                                  “Congratulations, you did it!
                                                   <p>You got the page to update!</p>”
 8                         theButtonClicked.innerHTML = “Click Me Again!”
 9      }
10     </script>
11
12     <style>
13               body{
14                    color:white;
15                    background-color:DarkSlateBlue;
16               }
17               #result_display{
18                    background-color:DarkGrey;
19               }
20     </style>
21   </head>
22
23   <body>
24        <button onclick=”showCongratulations(this)”>Click Me!</button>
25        <section id=”result_display”></section>
26   </body>
27 </html>
```

Code Sample - fancy_button.html

So lines **17** - **19** read like this. “Find the element who’s id is result_display and set its background color to be DarkGrey.” At this point, the CSS makes the body’s background DarkSlateBlue, any text in the body white, and the result display’s background DarkGrey.

Save and double click and what do you see? No change!! “You lied to me Barney!” Ahhh...nope. I didn’t. Click the button. Now what do you see? The section with a DarkGrey background color. So what’s up with that?

Well, as you learned in The Basics, sections represent an area, chunk, or part of the page. A physical, viewable part. When there is nothing inside of a section, it collapses and ends up having no height. Since the result_dislpay section is empty before you click the button it has no height. You can’t see its DarkGrey background color because it has no height. When you do click the button you see this.

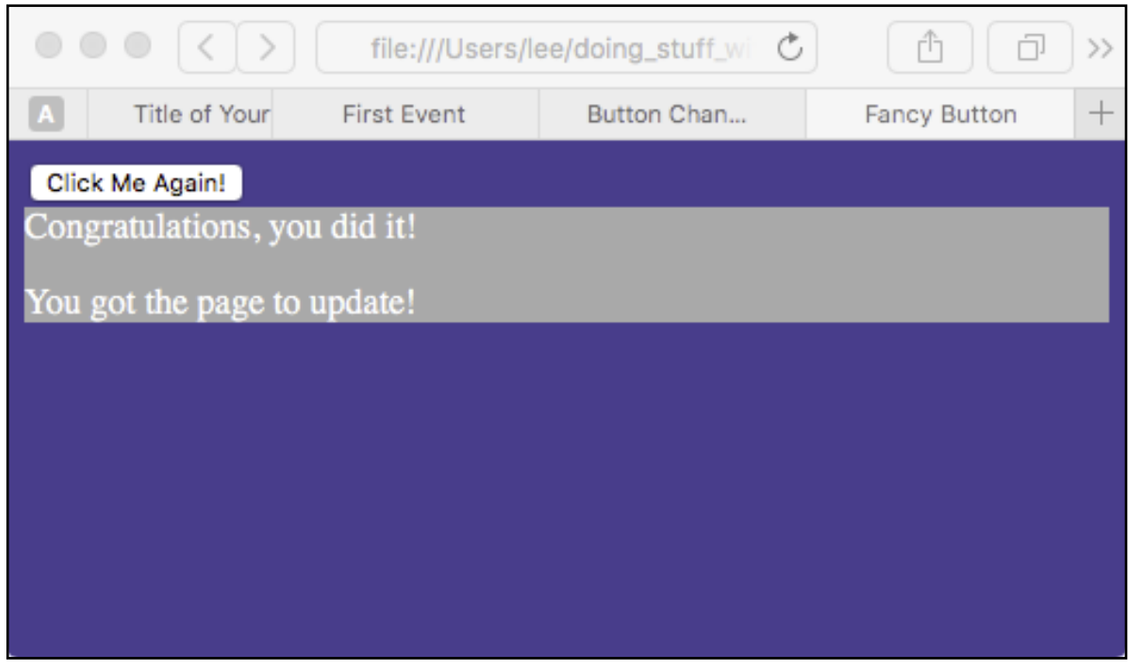


Figure 3. *The section background color color change to fancy_button.html in my web browser.*

Now the DarkGrey background shows up but the page is still very ugly. That grey band going across the whole page is awful. Let’s clean that up a bit. How about you make it so the result_display section always shows and only takes up a part of the width of the page.

Making The Section Look Better

What you’ll do is define a specific height and width for result_display. To do that you’ll need to add the height and width CSS proper-

ties to the description of result_display. When you do, your file will look like this.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>First Event</title>
 5     <script>
 6               function showCongratulations(theButtonClicked){
 7                         document.getElementById(“result_display”).innerHTML =
                                                  “Congratulations, you did it!s
                                                   <p>You got the page to update!</p>”
 8                         theButtonClicked.innerHTML = “Click Me Again!”
 9      }
10     </script>
11
12     <style>
13               body{
14                    color:white;
15                    background-color: DarkSlateBlue;
16               }
17               #result_display{
18                    background-color: DarkGrey;
19                    width: 300px;
20                    height: 150px;
21               }
22     </style>
23   </head>
24
25    <body>
26         <button onclick=”showCongratulations(this)”>Click Me!</button>
27         <section id=”result_display”></section>
28    </body>
29 </html>
```

Code Sample - fancy_button.html

Notice that the width and height defined both end in ‘px’. Px stands for pixels. Pixels are the little dots that display the colors on your computer screen. If you get a magnifying glass out and take a look at your computer screen you can see them. Line **19** tells your computer to make the result_display section 300 pixels wide. Line **20** tells it to make the result_display section 150 pixels tall.

Double click the fancy_button.html file and you’ll see your page looks like Figure 4. Now you’re starting to get something that looks somewhat decent. We aren’t done yet but progress is progress!

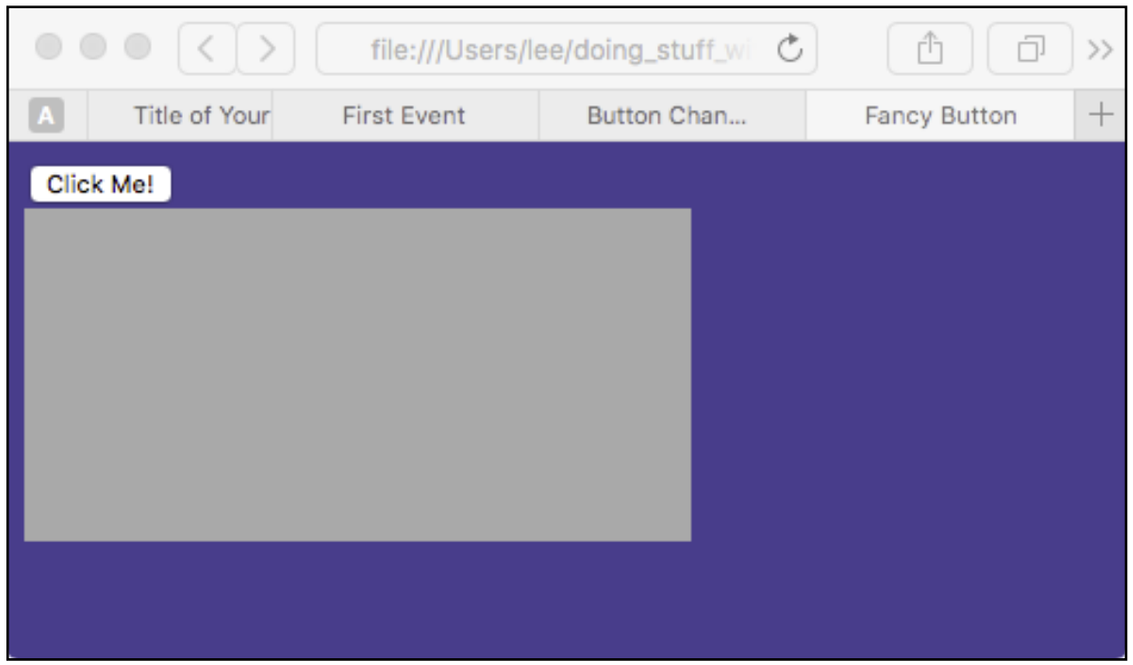


Figure 4. *The section width and height change to fancy_button.html in my web browser before the button is clicked.*

After you click the button it looks like this.

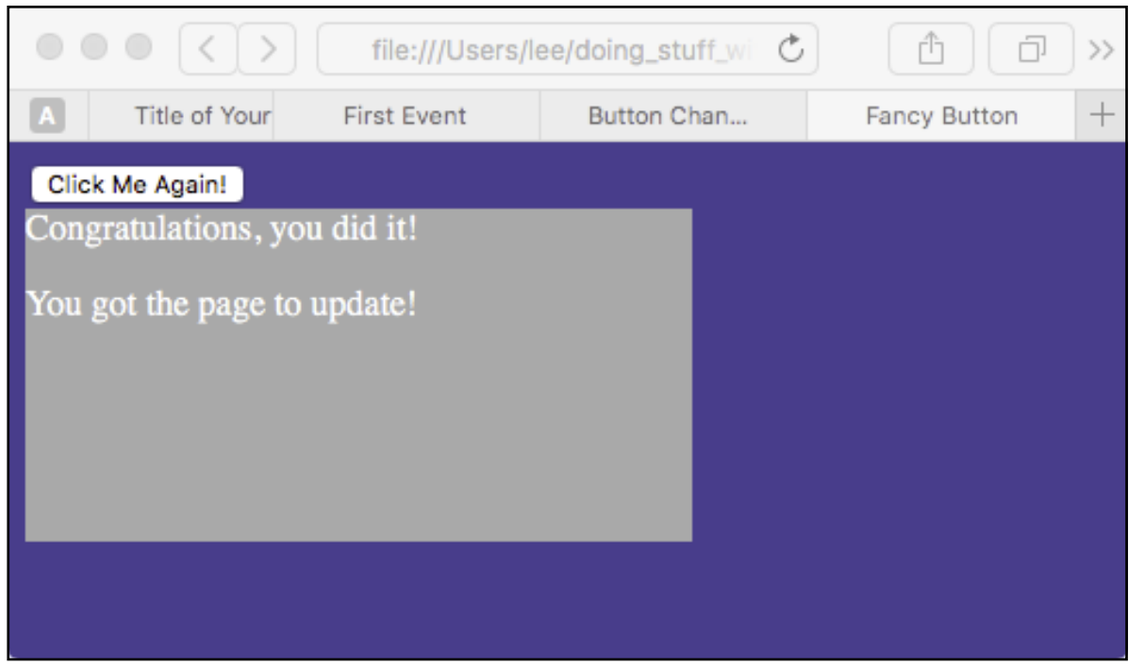


Figure 5. *The section width and height change to fancy_button.html in my web browser after the button is clicked.*

The page is getting better, but it would be nicer if the result-display section was inset from the left a little and had a small gap between the section and the button.

To do this you'll need to tell the section to be displayed somewhere else other than its default position. Three lines of CSS will do this. You'll add the three lines into fancy_button.html. The three new lines are numbers **21** - **23**. Let's go over them one by one.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>First Event</title>
 5     <script>
 6             function showCongratulations(theButtonClicked){
 7                     document.getElementById("result_display").innerHTML =
                                                "Congratulations, you did it!s
                                                <p>You got the page to update!</p>"
 8                     theButtonClicked.innerHTML = "Click Me Again!"
 9      }
10     </script>
11
12     <style>
13             body{
14                 color:white;
15                 background-color: DarkSlateBlue;
16             }
17             #result_display{
18                 background-color: DarkGrey;
19                 width: 300px;
20                 height: 150px;
21                 position: relative;
22                 top: 10px;
23                 left: 10px;
24             }
25     </style>
26  </head>
27
28   <body>
29       <button onclick="showCongratulations(this)">Click Me!</button>
30       <section id="result_display"></section>
31   </body>
32 </html>
```

Code Sample - fancy_button.html

Details:

Line **21** tells the result_display section that it should be prepared to display itself in a different spot than usual. In fact, by adding line **21** you can move the section all over the page. There are other types of positioning you can explore, but for now 'relative' is all you'll need to know.

Line **21** reads, “Hey result_display, don’t position yourself normally, instead you should allow yourself to be shifted around. Oh..and when you are told to shift, shift from where you normally would be displayed.”

This means that line **22** in English is, “Hey result_display, put your left edge 10 pixels further to the right than you normally would.” And then line **23** is, “Hey result_display, put your top edge 10 pixels further down than you normally would.”

There are a lot more ways to shift elements of your page around. You don’t have to use top and left. Neither does the amount of the shift have to be in pixels. You should explore those other ways but for this book we’ll stick with top, left, and position relative.

With these lines added in, result_display has shifted down and right a bit. When you double click fancy_button.html you’ll see Figure 6.

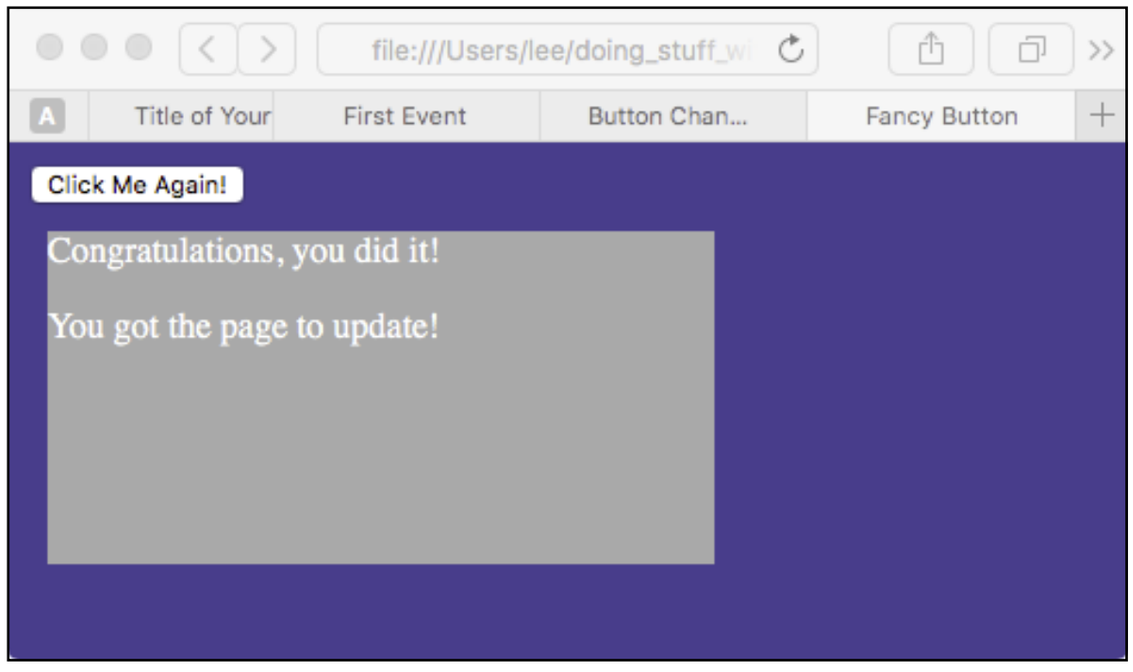


Figure 6. *The section position change to fancy_button.html in my web browser after the button is clicked.*

There is still one thing that is bugging me about how fancy_button.html looks in my browser. The text in the result_display section is crammed right up against the top and left of the section. If there was a lot of text in the section it would go all the way to the right side and the bottom too. That just doesn’t seem right to me so let’s go in and fix it. Don’t worry, this is the last change.

An easy way to make text have space around it in a section is to add some space to the inside edges of the section where the text isn’t allowed to be. That means the text will shift away from the edges. You can do this by adding only one more line of CSS.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>First Event</title>
 5     <script>
 6               function showCongratulations(theButtonClicked){
 7                         document.getElementById(“result_display”).innerHTML =
                                                  “Congratulations, you did it!s
                                                  <p>You got the page to update!</p>”
 8                         theButtonClicked.innerHTML = “Click Me Again!”
 9      }
10     </script>
11
12     <style>
13               body{
14                    color:white;
15                    background-color: DarkSlateBlue;
16               }
17               #result_display{
18                    background-color: DarkGrey;
19                    width: 300px;
20                    height: 150px;
21                    position: relative;
22                    top: 10px;
23                    left: 10px;
24                    padding:5px;
25               }
26     </style>
27   </head>
28
29    <body>
30         <button onclick=”showCongratulations(this)”>Click Me!</button>
31         <section id=”result_display”></section>
32    </body>
33  </html>
```

Code Sample - fancy_button.html

Details:

Line **24** is the new one. It says, “Hey result_display, put 5 pixels worth of padding all around the inside of your edges so other things can’t be there.” You don’t have to use pixels for your padding, there are other options, and you don’t have to have the same amount of padding on each of the edges. You can use padding in any way that is right for your situation. With a 5 pixel padding, fancy_button.html looks like Figure 7 in your browser.

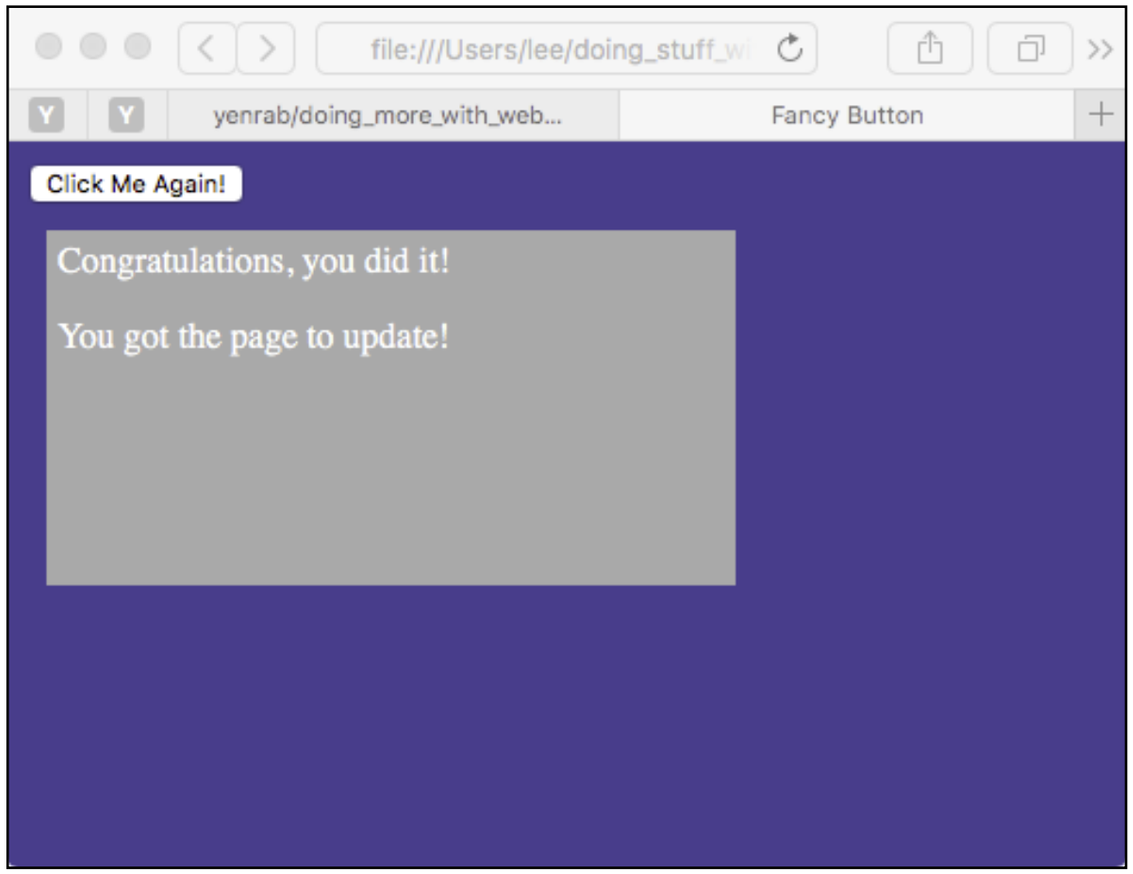


Figure 7. *The padding added to the section in the fancy_button.html*

So that’s a brief introduction to the huge number of possibilities you will find when you dive into CSS. “Brief?” you say? “It took 33 lines of code to do that?” I know. When you first start doing CSS, HTML, and JavaScript 33 lines of code is a lot. 33 elephants in your back-yard would be a lot but 33 grains of rice to eat isn’t a lot. The more you play with what you have learned about in this chapter and the more you explore other CSS and HTML things the smaller those elephants will get. Before too long they’ll all be grains of rice.

STOP HERE. PLEASE.

Do not go on to the next chapter. Instead, spend several days exploring the different things you can do with CSS. There are going to be a huge number of them. Remember...you don't need to know them all today. Pick some of them that seem simpler than others and play with them. Most of all, HAVE FUN. Play around with this CSS stuff.

Remembering User's Info

Points to Ponder

1. How can I store data so it isn't lost when the application stops?
2. How do I get data back into my application once it has been stored?
3. What can I do to allow the user to communicate?

Information is vital to our lives. Even small choices like where to go on a hike require information. Be ready for a workout if you hike Damnation Creek Trail.

When you write an application you are communicating with the user. For there to be true communication it must be bidirectional. The user needs some way to speak to you.

Another point to remember is computers are stupid. They can sometimes seem smart because they are so fast at being stupid. They can make all kinds of stupid mistakes, evaluate and reject them according to some rule they've been given, and then finally end up making a good choice.

Because of their stupidity, computers can't remember anything. When you stop an applica-

tion, anything it has been working on is gone...unless the application is designed to store what it has done.

In techie speak the information stored or shared with the user is called output. Anything coming in from where it was stored or anything the user enters is called input. To techies, input and output are called IO, Input-Output. You've already done output in Section 1. You'll need to remember how to do that. If you don't, PLEASE go back and review how to show stuff to the user. We'll build on that in this section.

HINT: If you haven't read Sections 1 and 2, GO BACK!!!

Building The HTML

You're going to make a new project in this section so make a copy of template.html and name it story_editor.html. What this app is going to do is allow people to write, save, edit, and show a story they write. What I'm trying to do by having you create this, is show you how you can get, store, and display users' data. Don't start thinking this only works with text or stories...you can use these ideas any kind of project with any kind of data.

OK...so here we go. Let's make some more computer magic go away. :)

Like you did in the other stuff we've done, put the HTML together first. You'll need to include a tag that allows the user to tell you the name of their story. You'll also need to add in a tag that will let them type in their story. Since we want to be flexible, we'll let them type in any HTML they want as part of their story. That should make it more fun for them.

So at this point it looks like you need to learn two new types of tags. If you're a speaker of English you'll have an advantage with the first new tag. If you were going to name a tag that allowed the user to do input, what would you call it???

Well the old guys with long white beards decided to call it the input tag. Way to go old guys! You got this one right! The other tag, the one where the user will write their story, needs to be big and cover a large area on the screen. The user needs to be able to enter the text for their story there too. Guess what the old white bearded guys called this one??? They called it textarea. Not a bad name at all...not the greatest, but not bad either.
We know we need them so let's put them in.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>Story Writer</title>
 5     <script>
 6
 7     </script>
 8
 9     <style>
10
11     </style>
12  </head>
13
14   <body>
15       Story Name:<input id=name_input></input>
16       <textarea id="story_editor" placeholder="Write a totally awesome story
                                                                    here."></textarea>
17   </body>
18 </html>
```

Code Sample - story_editor.html

Details:

Take a look at line 4. The title is changed to match what the page is going to do. Make sure you always update the title of the page template. If you don't it will look weird. People might start thinking you were new at this. :)

Line 15 is where that new input tag is. It has a starting and ending tag just like the others you've seen. It also has an id like we talked about in Section 2. We'll need the input tag's id later. The 'Story Name:' stuff right in front of the tag is there to give a hint to your user...so they know what they should type in.

Line 16 looks a little stranger. It is the text area tag, and it also has an id. "But what is that placeholder thing???" you ask. Again a good question from you. You're getting better at this. The 'placeholder thing' is a property specific to textarea's and input tags. You can use it to put in a hint to your user. Tell them what they should type in the text area. In this case, 'Write a totally awesome story here.' Will be in the text area to begin with, and will go away as soon as they start typing.

So far, your page should look like this.

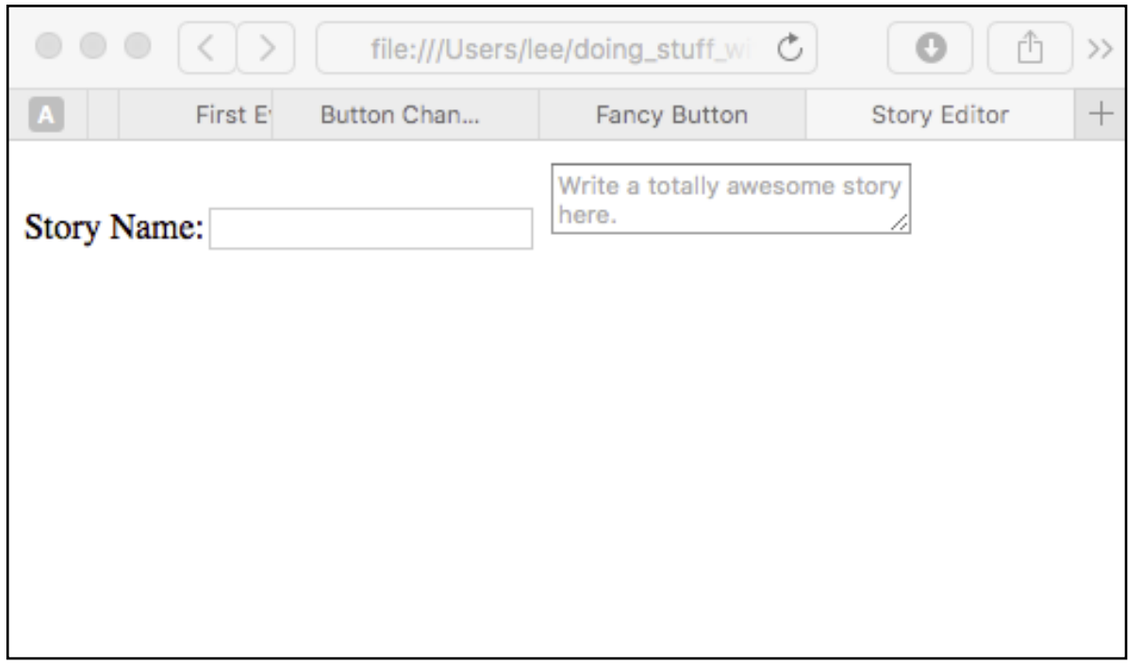


Figure 1. *The story_editor.html page in my web browser.*

Yep. The page is pretty ugly. We should add some more tags to make this better. Let's start by adding something at the top of the page that describes what the page is for. It's called a header tag. Header tag??? Not so great a name old guys. Especially when you make us write it like this.

<h1>Story Editor</h1>

Wow...epic fail.

What they mean by h1 is a level one header. "OK....but that doesn't explain it any better," you say. Yep you're right. Let's try again. Header levels are numbers. The level 1 header is the largest, level 2 is smaller, 3 is smaller than that, and 4 is the smallest. You're right, you're right. Why didn't they keep going with 5, 6, and more? You'd have to ask the old white bearded guys. I don't know.

Oh...and you are probably wondering if there is a way to do in between sized headers. Something like <h1.5>Hello</h1.5>. Nope. You can't do that. If you want detailed control the size of the text in a header you'll have to use some CSS. You won't do that in this example, but you will later. There are lots of examples on the web if you want to take a look now. Just google 'CSS font size'. That should get you started.

Anyway, the code should now be this.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Story Writer</title>
5     <script>
6
7     </script>
8
9     <style>
10
11    </style>
12  </head>
13
14   <body>
16       <h1>Story Editor</h1>
17       Story Name:<input id=name_input></input>
18       <textarea id="story_editor" placeholder="Write a totally awesome story
                                                    here."></textarea>
19   </body>
20 </html>
```

Code Sample - story_editor.html

And the page should look like this.

Figure 2. *The header change to story_editor.html page in my web*

Yeah...I know. Still not so great, but we'll work on this some more.

Let's add in three buttons so the user can find and edit, save, and display their story. We won't create the JavaScript to link to the buttons just yet. We'll add that in later, but it would be nice if we could get the page to look like this.

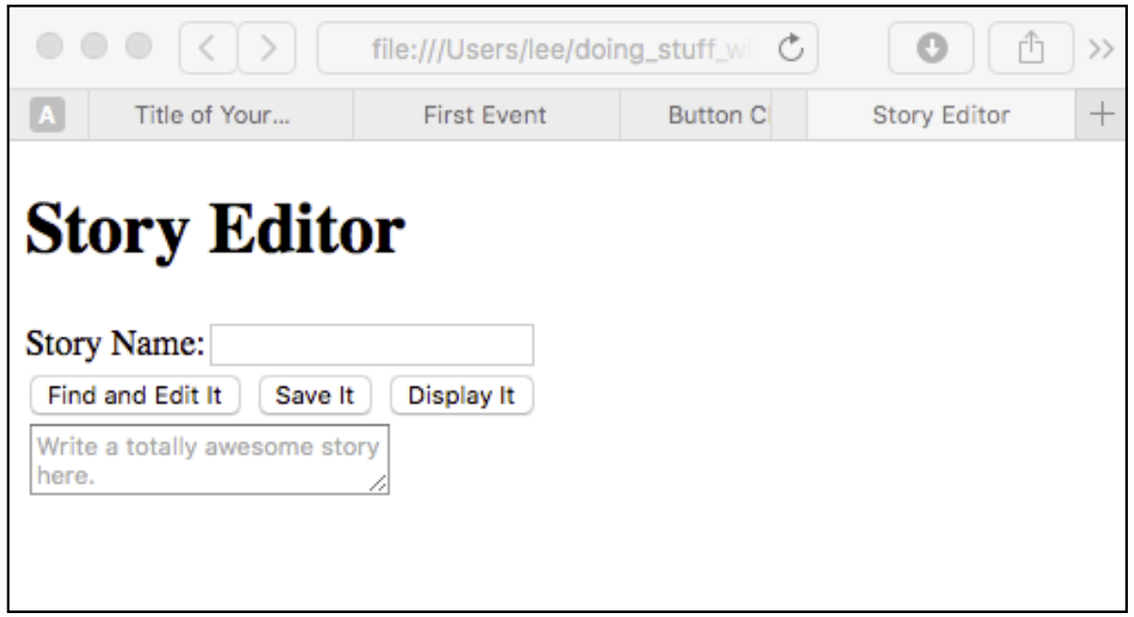


Figure 3. *The temporary goal for the story_editor.html page.*

If you can get it to look like this, at least the input areas, buttons, and header are all going from top to bottom. That's better than spread horizontally across the page.

If you remember back in Section 2, ***section*** tags fill the screen all the way across. Take advantage of this. Put the buttons inside a section tag. Since the section tag requires the full width of the page, the buttons can't be on the same line as the two inputs. In other words, the page will HAVE TO display the way we want!

If you put the section and the buttons after the input tag and before the textarea tag your story_editor.html file will look like this. Oh...and you should add in a section to display the story in while you're at it.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Story Writer</title>
5     <script>
6
7     </script>
8
9     <style>
10
11    </style>
12  </head>
13
14  <body>
16      <h1>Story Editor</h1>
17      Story Name:<input id=name_input></input>
18       <section>
19            <button onclick="loadStory()">Find and Edit It</button>
20            <button onclick="saveStory()">Save It</button>
21            <button onclick="displayStory()">Display It</button>
22      </section>
23      <textarea id="story_editor" placeholder="Write a totally awesome story
                                                                here."></textarea>
24       <section id="story_display"></section>
25  </body>
26 </html>
```

Code Sample - story_editor.html

Details:

Line 18 is the starting tag for the section containing the three buttons. The section's ending tag is on line 22. The ***onclick*** event listeners for the three buttons are set to three different functions, one for each button. The functions don't exist yet, you'll add them in a little bit, so clicking on the buttons won't do anything right now.

Adding The CSS

There's just one more thing I think you should do to make the story editor page work better. The text area is way too small. If it

were bigger it would be easier for the user to type in their story. Our job is to make the user's job easier, so let's add some CSS to make the page nicer. While we're at it, why don't we change the background color of the textarea so it isn't white.

We can do it. The background color of any tag can be changed and textareas are tags.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Story Writer</title>
5     <script>
6
7     </script>
8
9     <style>
10    textarea{
11              width:800px;
12              height: 250px;
13              background-color: lightyellow;
14     }
15    </style>
16  </head>
17
18   <body>
19       <h1>Story Editor</h1>
20       Story Name:<input id=name_input></input>
21        <section>
22              <button onclick="loadStory()">Find and Edit It</button>
23              <button onclick="saveStory()">Save It</button>
24              <button onclick="displayStory()">Display It</button>
25       </section>
26       <textarea id="story_editor" placeholder="Write a totally awesome story
                                                                    here."></textarea>
27        <section id="story_display"></section>
28   </body>
29 </html>
```

Code Sample - story_editor.html

Details:

So at this point, lines 10 - 14 in English say, "Hey text areas, set your width to be 800 pixels, your height to 250 pixels, and your background color to light yellow." (Remember I warned you you should go back and see Section 2 if you hadn't already? CSS is introduced there. If this new stuff is confusing you, go read, learn, and understand that section.)

So this is all the 'making it pretty' to do for this page. You can add as much more as you want. Feel free. It won't hurt anything and will help you learn the CSS stuff. For now, the page looks like this.

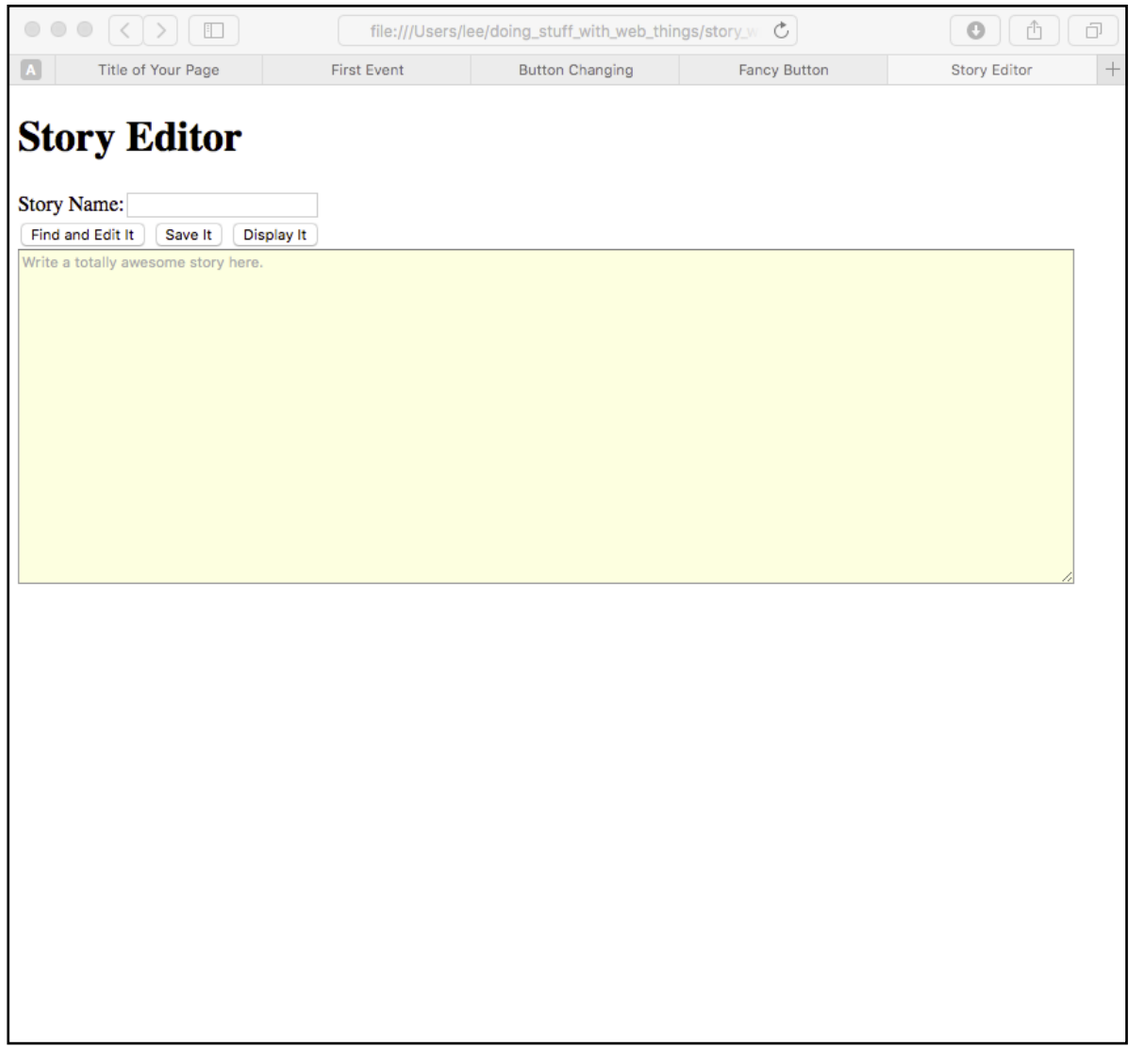


Figure 4. *The final look of the story_editor.html page.*

With the page like this, the user can see what they need in order to create, save, edit, and display stories. When you're all done, when they enter a story and click the 'Display It' button they see something like this.

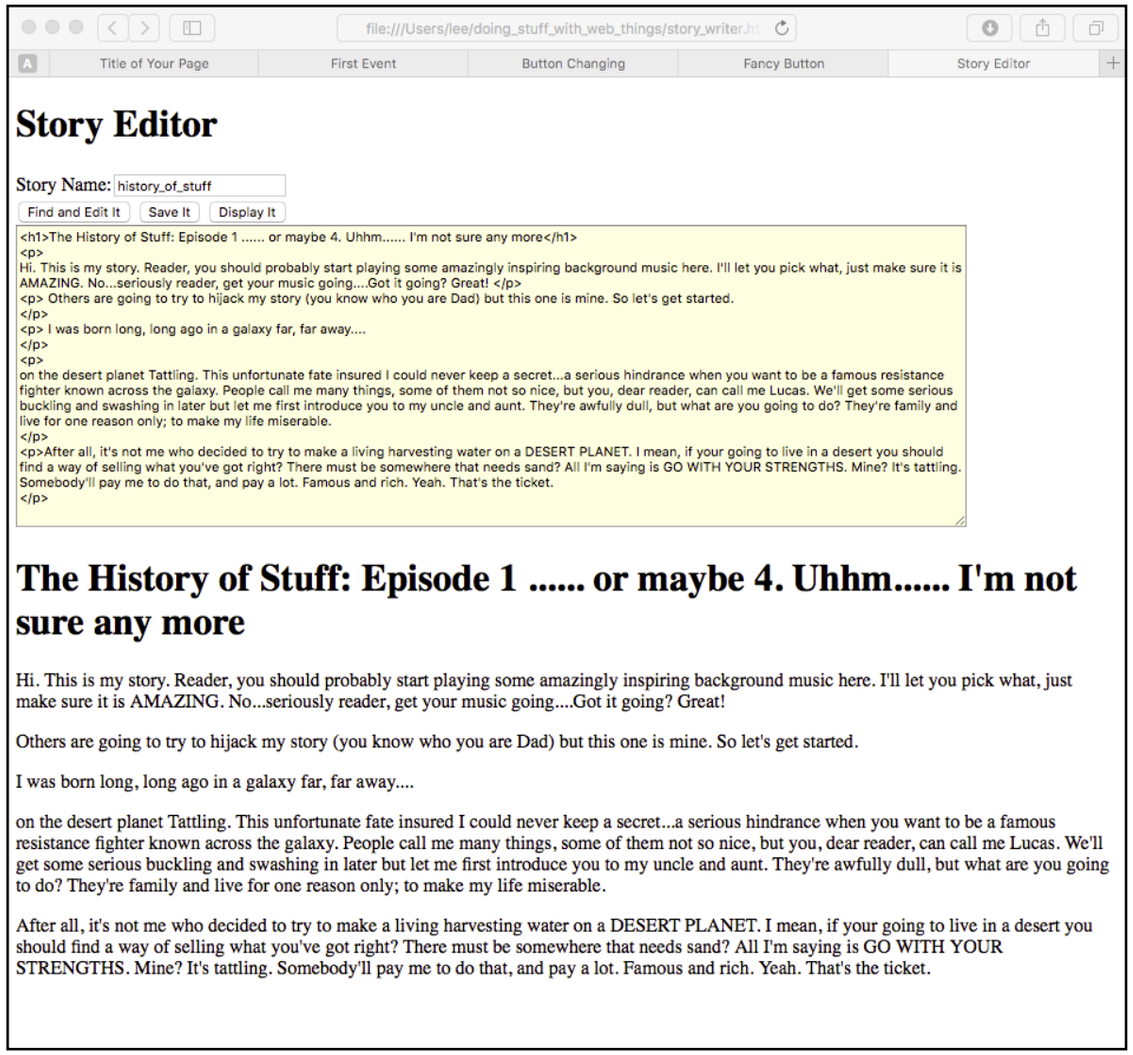


Figure 5. *A story created, edited and displayed in story_editor.html*

Now The JavaScript

With all this HTML and CSS in place, let's add the JavaScript code that makes the display active. You'll need three methods to match those listed in the HTML as being the ***onclick*** listeners for the buttons.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>Story Writer</title>
 5     <script>
 6         function loadStory(){
 7               var storyName = document.getElementById(“name_input”).value
 8               var storyHTML = localStorage.getItem(storyName)
 9               document.getElementById(“story_editor”).value = storyHTML
10         }
11         function saveStory(){
12               var storyName = document.getElementById(“name_input”).value
13               var storyHTML = document.getElementById(“story_editor”).value
14               localStorage.setItem(storyName, storyHTML)
15          }
17          function dsiplayStory(){
18               var storyHTML = document.getElementById(“story_editor”).value
19               document.getElementById(“story_display”).innerHTML = storyHTML
20          }
21     </script>
22
23     <style>
24     textarea{
25               width:800px;
26               height: 250px;
27               background-color: lightyellow;
28      }
29      </style>
30   </head>
31
32    <body>
33         <h1>Story Editor</h1>
34         Story Name:<input id=name_input></input>
35          <section>
36               <button onclick="loadStory()">Find and Edit It</button>
37               <button onclick="saveStory()">Save It</button>
38               <button onclick="displayStory()">Display It</button>
39         </section>
40         <textarea id="story_editor" placeholder="Write a totally awesome story
                                                                    here."></textarea>
41          <section id="story_display"></section>
42    </body>
43 </html>
```

Code Sample - story_editor.html

Put the three functions in between the starting and ending script tags like we learned in Section 1 and saw again in Section 2. It doesn't actually matter what order you put the functions in, JavaScript can handle any order, but let's look at the displayStory function first.

Details:

OK, OK... don't panic. That's the most important thing right now. DON'T PANIC. Take a deep breath and I'll step you through this.

Line **17** looks familiar. You saw how to read lines like this in Section 1. This one adds a function called displayStory and as you learned earlier, what this function does is stated in the lines between the { and } characters.

However, line **18** adds a new wrinkle. "What is that ***var*** thing?" you ask. That's the new wrinkle. That ***var*** thing is what JavaScript programmers use to tell the computer to create a 'variable'. Hmmm...var....variable....yep, that's programmers being lazy again. Don't try to write 'variable' in you code the computer won't understand it if you do. Use 'var' instead.

A variable is like a box you can put stuff in. In the kitchen in my house I have boxes I put stuff in. To help me keep track of what is in the boxes, I put labels on the boxes. So I have one box with flour in it and a label on it that says 'flour.' I've got another box with raisins in it and a label on it that says 'raisins.' You get the idea. I put raisins in

the raisin box and I put flour in the flour box. Variables are like boxes with labels on them. Since you are the programmer you are the one who decides what label to put on the box, but just like in my kitchen, it is very wise to have a label that matches what the box has in it. If you don't, then things get confusing really fast.

Let's read line **18** in English. "Hey document, go get the element who's id is 'story_editor'. Take what you find inside it, its value, and store what you find in a variable (box) named 'storyHTML'."

Wow. We made it through that one. Notice that when reading this kind of a line of code you read all of the stuff on the right-hand side of the = sign first. By the way, that's the stuff the computer will do first. Read through the English several times and compare it to line **18**. Don't worry if you don't get it right off. We'll do several more of these by the time we get done discussing the three functions.

Now that you've got what the user wrote in the story_editor and put it in the storyHTML box, lets read line **19** to see what to do with it. Since the user can write any HTML they want, the user should be able to see what their HTML looks like. Line **19** does this and reads,"Hey document, go get the element who's id is 'story_display' and put what you find in the storyHTML variable between the story_display's starting and ending tags."

Hey...that's almost exactly like what you did in Section 1 when you first learned about functions. The difference here is you're using a variable to hold the text the user entered and in the Section 1 example you put the text directly in between the section's beginning and ending tags.

Let's move on to the saveStory function. It will give you more practice at understanding variables. So you don't have to keep looking back, I've copied the JavaScript code for the function in Code Snippet 1.

```
11      function saveStory(){
12        var storyName = document.getElementById("name_input").value
13        var storyHTML = document.getElementById("story_editor").value
14        localStorage.setItem(storyName, storyHTML)
15      }
```

Code Snippet 1

Details:

Line **11** is where the function starts and line **15** is where it ends. Line **13** is exactly the same as line **18** that you just got done with but line **12** is different. Let's look at that one. It says, "Hey document, get the element who's id is 'name_input'. Take what you find inside the input box on the screen, its value, and store what you find in a variable named 'storyName'." So this one reads almost exactly the same as line **18** that you saw before. In both cases, the elements the document gets are input elements. 'name_input' is an input tag into which the user can type a name for the story. 'story_editor' is the textarea tag where the user types their story.

When you want to get the user's input from any input or text area tag, you use the special word 'value.' When you want to show something to a user, you use a section (see Section 1) and the special word 'innerHTML.' So...if your dealing with an input or textarea use value; a section, use innerHTML. Don't get these mixed up. If you do, your code will run without an error but your page won't get or display anything. Be careful. You'll just have to remember which word to use in these two different situations, input or textarea use value and section uses innerHTML.

Now for the new bit. Using JavaScript you can store any information you want to use later. Even if your user refreshes the page or goes to some other site and then comes back, anything you've saved will still be there. Handy! I'm pretty sure the user wants to save their story so they can edit it or add to it even if they have left the page. Line **14** is the code that stores the story. It uses something called 'localStorage.'

Line **14** in English is, "Hey, localStorage, store what's in the storyHTML variable (box) and label that stored piece of information with what you find in the storyName variable (box)."

JavaScript's local storage uses what techies call 'key-value pairs.' That's fancy talk for 'description-stuff pairs.' If you've ever seen an old fashioned phone book you've seen this in action. The 'key' would be the name of the person and the 'value' would be their address and phone number.

I know you've got an audio player. If it is anything like mine, you can see a list of music titles and when you select the title the music plays. The title is the 'key' and the audio file is the 'value.'

In grade books, teachers use a student's name as the 'key' and all of their grades as the 'value.' If you think about it you'll find you use these sorts of pairs all the time. Go find some that you use. Describe them to yourself. It will help you remember what is meant by 'key-value pairs.' Do this for a couple of days to cement the concept into your head.

STOP HERE. PLEASE.

Don't go on until you are very comfortable with key-value pairs. It you choose to ignore this advice, the next bit won't make any sense.

Now that you're back and REALLY understand key-value pairs we'll take a look at the function that retrieves, techies would say 'loads', a story the user saved. I've copied the function here to keep us from having to go back to the Code Sample.

```
6       function loadStory(){
7           var storyName = document.getElementById("name_input").value
8           var storyHTML = localStorage.getItem(storyName)
9           document.getElementById("story_editor").value = storyHTML
10      }
```

Code Snippet 2

Your user will have to type a story name into the page's input box to load the story. For now, they will just have to remember the names of their stories. Later you can make this better for them by giving them a list of stories they've already stored but right now you don't want to complicated example. Right?

Details:

Line 6 starts the loadStory function and Line 10 is where it stops. You saw Line 7 in the description of the saveStory function. Can you still read it? Give it a try. Convert it into English. If you struggle with this go back to the saveStory description. If not, let's keep going.

You used localStorage to save a story in the last function. This time you'll use it to get a saved story. You do this by using the key for the stored value that is the story your user stored. If you don't remember or understand key-value pairs, **GO BACK**. Look over the saveStory description again.

Line 8 uses local storage to get a saved story. It reads,"Hey localStorage, get the item stored using what's in the storyName variable (box) as a key. Put the story you find in a variable (box) called 'storyHTML'."

Up to this point you've used a key and a value to store and retrieve a story. Now all you need to do is show your user the story they previously saved. That's line 9. "Hey document, get the element who's id is 'story_editor' and put in the visual box on the screen (in its value) what is in the 'storyHTML' variable."

There you go. With this example your app should be able to remember, in other words read and write, information gathered from the user. Use this in your apps. Nobody wants to use an app that can't remember what they already told it.

Making Choices

Points to Ponder

1. How do JavaScript applications choose between different behaviors depending on the situation?
2. How do I structure the choices so my application's behavior is predictable?

If you go out on a limb you will eventually have to choose a branch to follow.

Choosing between things is always difficult. Choosing one thing means not choosing the other options. There are a lot of studies about humans and options. They show that the more options people have the worse they feel about the choice they made. Think of the last time you bought ketchup. So many options.

Computers don't have the problem of not being satisfied with the choice they make. But then again, they're stupid.

JavaScript has a structure that allows it to choose between any number of options. Being a simplified version of English, JavaScript uses english words to express choice making. In English you

could phrase a choice between doing multiple things based on some different situations. You'd say, 'If ....or if....or if....or.' In JavaScript it is written 'if....else if....else if....else.'

If what? What kinds of situations can JavaScript check? Well, you could check to see if a number is even or odd. To do this you'd use the Modulo operator, %. The Modulo operator returns the remainder of an integer division. 5%2 yields 1. 4%2 yields 0. So any number %2 is even if the remainder, what you get from using the modulo operator, is 0.

For the first example in this section, start by duplicating template.html like you did before. Then name it even_or_odd.html. You'll use this new file to add in the HTML elements you'll need. You'll need an input tag so the user can enter a number. You'll also need a button and a section to show the user if the number is odd or even. When you get all of those added, even_or_odd.html will look like this.

```
<!doctype html>
<html>
  <head>
    <title>Even or Odd???</title>
    <script>

    </script>

    <style>

    </style>
  </head>

  <body>
      <h1>Even or Odd???</h1>
      Enter a Number: <input id=name_input></input>
            <button onclick="checkNumber()">Check It!</button>
       <section id="result_display"></section>
  </body>
</html>
```

Code Sample - even_or_odd.html

If you skipped right to here and are confused PLEASE go back and go through the previous sections. The HTML used here is explained in them.

With this HTML ready to go, add in the JavaScript to use modulo to check if the number the user entered is even or odd. When you do it will look like what you see in the expanded Code Sample below.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Story Writer</title>
5     <script>
6         function checkNumber(){
7               var numberAsText = document.getElementById(“number_input”).value
8               var remainder = numberAsText % 2
9               if(remainder == 0){
10                    document.getElementById(“result_display”).innerHTML =
                            numberAsText + “is even.”
11              }
12              else{
13                    document.getElementById(“result_display”).innerHTML =
                            numberAsText + “is odd.”
14              }
15        }
16    </script>
17
18    <style>
19
20    </style>
21  </head>
22
23   <body>
24       <h1>Even or Odd???</h1>
25       Enter a Number: <input id=name_input></input>
26              <button onclick="checkNumber()">Check It!</button>
27        <section id="result_display"></section>
28   </body>
29 </html>
```

Code Sample - even_or_odd.html

Details:

Once again, DON’T PANIC. We’ve got through some hard stuff before. We can do it again. Let’s take this JavaScript one line at a time.

Line 7 should look very familiar by now. It reads, “Hey document, go get the element who’s id is ‘number_input’. Take what you find inside it, its value, and store what you find in a variable named ‘numberAsText’.” If the ***var*** and the = are confusing you, go back and check out what variables are. If not, let’s keep going.

Line 8 is where you use modulo. Remember, using it results in a 0 if numberAsText contains an even number because the number after the % (modulo) operator is 2. If the number in numberAsText is odd, you’ll get a 1. Let’s read line 8 together. “Apply modulo 2 to what ever number is inside numberAsText and put the result into the remainder variable.” Read over the English several times and compare it to the JavaScript code. It can also help if you try to explain it to someone else...even if they don’t understand what you’re talking about. I’ve done that a bunch of times and, believe it or not, it works. Give it a shot.

Now comes the nitty-gritty stuff. Lines 9 - 16 are where your code makes a choice. The choice to be made is whether you should tell the user the number they entered is either even or odd. To do this you will need to use what techies call a comparison operator.

JavaScript has a bunch of operators. You’ve used the % (modulo) and = (assignment) operators already. If you want to see and try out the ones you haven’t used yet, take a break and go ahead and do so. In fact, I’d suggest you should take a break right now, click on the link above, and play with them all in some JavaScript. Especially the + operator. It will make what comes next easier.

STOP HERE AND GO PLAY. PLEASE.

Your back? OK, let's move on.

Line **9** is where the first choice is made. It uses the ***equals comparison operator*** (==), not the ***equals assignment operator*** (=). Be careful. They are different and if you accidentally put a = in your code instead of a == it will be hard to find your bug and fix it. Make sure you use == to compare if two things are equal and = to put something in a variable.

Anyway, line **9** says, "Check to see if what is in the remainder variable is equal to zero." That wasn't so bad, right? Now the part that seems to confuse a lot of people. You remember from what you've seen before that function does is what's between its { and } characters. It is the same with the ***if*** comparison on line **9**.

Lines **9** - **11** have the == comparison in them AND what to do if what is in the remainder variable is a zero. I've copied those lines here to make it easier for us to go over them one-by-one.

```
9          if(remainder == 0){
10               document.getElementById("result_display").innerHTML =
                       numberAsText + " is even."
11         }
```

Code Snippet 1

In English you'd read lines **9** - **11** like this. "If the number in the remainder variable is equal to zero append the text 'is even.' to what you find in the numberAsText variable. Then, document, go get the element who's id is 'result_display'. Use the element and put the stuff you appended together between result_display's beginning and ending tags."

Wow. That's a mouthful. Go over it several times.

A techie would read line **9** like this. "If remainder is equal to zero..." They would then complete the sentence by reading line **10**. "then concatenate the text 'is even.' to numberAsText and assign it as the HTML inside of the 'result_display' element."

What this comes down to is this statement. Comparisons have { and } characters. Between those characters are what the computer will do if the comparison is true. Is it true that what is in remainder is equal to zero? If it is, then do what is between the { on line **9** and the } on line **11**.

"But what if the number they entered is Odd?"you ask. Wow! you always have such good questions!

If it isn't true that what is in remainder is zero you need to add an 'else'. This is techie speak again. Back at the beginning of this section I mentioned that 'else' is like 'or' in normal English.

"If the number is even, I'll show them their number and 'is even' or I'll show them their number and 'is odd'." That's the regular English. You've seen the JavaScript for the if it is section, lines **9** - **11**. Now lets look at Lines **12** - **14**. They are the else part of the choice.

```
12          else{
13                  document.getElementById("result_display").innerHTML =
                            numberAsText + "is odd."
14          }
```

Code Snippet 2

Line **12** just says 'else'. This is the 'else' connected with the 'if' on line **9**. In other words, this is what should happen if the number in remainder ISN'T equal to zero. Any 'else' that you create for an if also has { and } characters. You put the JavaScript between them for this situation. You can see on line **13** the change. Instead of 'is even.', 'is odd.' is going to be displayed along with the number. Other than that, the JavaScript, in this situation is exactly the same as the JavaScript for when the remainder is equal to zero.

I'd suggest going back to the previous Code Sample and seeing how it all fits together. I'd also suggest you double click on the file you created and try it out.

Now that you've played with this stuff a bit, you probably found it has some bugs. For example, if you put a zero in and click the button, you see '0 is even.' on the page. This is obviously wrong. Mathematically, 0 is neither even nor odd. We'll fix this one together.

Let's think this through.

Dealing With Zero's

If what's in numberAsText isn't zero and remainder has a zero in it, then what's in numberAsText must be even. ***(Statement A)***

If what's in numberAsText isn't zero and remainder has a 1 in it, then what's in numberAsText must be odd. ***(Statement B)***

If neither of those statements are true, then what's in numberAsText must be zero. ***(Statement C)***

Wow...three possibilities. "How do you handle that Barney? 'if' and 'else' can only handle two possibilities!" You are absolutely right. That is why the old guys with white beards created the next two things you're going to see; AND and 'else if'.

Let's start with AND. You already know the '==' comparison operator. Now you'll learn the AND operator. Remember how lazy programmers are? AND is obviously way too long to type. I mean its three whole characters...and they are capital letters. How awful! So instead, you use '&&' to represent AND. Like most of the other operators in JavaScript, the old guys didn't come up with this themselves, they stole it from Mathematics. In math, && means AND.

So let's fix up line **9**. You should make your version look like this.

```
 9          if(numberAsText != 0 && remainder == 0){
10                document.getElementById(“result_display”).innerHTML =
                        numberAsText + “ is even.”
11          }
```

Code Snippet 3

Now line **9** reads, “If what’s in numberAsText is not equal to zero and what’s in remainder is zero...” The JavaScript matches ***Statement A*** of our ‘think it through’ session. “That means ‘!=‘ must mean ‘is not equal to’!,” you exclaim. You are so right! That’s exactly what it means! In fact, any time you come across an exclamation point (!) in JavaScript it always means ‘not’. That’s a great and handy thing to know! Oh...by the way, in case you forgot, ! doesn’t mean ‘not’ in English. :)

So you’ve fixed line **9**, but there is more to go. You’re JavaScript needs to match ***Statements B*** and ***C***. Let’s start by matching ***Statement B***. You’ll need to modify line **12** to do this.

Don’t freak out on me. It’s OK. Changing things like this is very common when you are writing code. It’s how we fix bugs. “But why didn’t you just show me how to do it right the first time?!” I understand your frustration, but if I showed you the end from the beginning, you would have been VERY confused. It’s better to do this a little at a time. After all, it’s by small and simple things that great things are brought to pass.

OK. So here is how you could fix up line **12**. It uses another ‘something new’ so get ready. Here it is.

```
12          else if(numberAsText != 0 && remainder == 1){
13                document.getElementById(“result_display”).innerHTML =
                        numberAsText + “is odd.”
14          }
```

Code Snippet 3

The new part is ‘else if’. It allows you to do more than one check. In English line **12** , completing what was started on line **9**, reads, “...or if what’s in numberAsText is not equal to zero and what’s in remainder is 1...” There. Now you’ve matched ***Statement B***. You’re code is getting better, but still doesn’t handle ***Statement C***.

So... Should you use another ‘else if’ and check if numberAsText is equal to zero or just use ‘else’ like you did before? Both will work, but do you need to do the check? According to ***Statement C***, no you don’t. ***Statement C*** doesn’t have any comparisons in it. Is essentially says, “..then numberAsText must contain a zero.” If that’s true, then you can use an ‘else’ since ‘else’ *doesn’t do any comparison checks*. Be careful. If you try to do a comparison check with an else, you code will fail. It’s a common mistake for people new to writing JavaScript. ‘if’ and ‘else if’ do comparison checks. ‘else’ does not. ‘else’ is there for when none of the other comparison checks are true.

The ‘else’ for this situation, according to ***Statement C***, only happens when the number the user entered is zero so your message should tell the user that zero, mathematically, isn’t even or odd.

```
15          else{
16                  document.getElementById(“result_display”).innerHTML =
                            “0 is neither even nor odd.”
17          }
```

Code Snippet 3

With these changes in place, even_or_odd.html looks like this.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>Story Writer</title>
 5     <script>
 6         function checkNumber(){
 7               var numberAsText = document.getElementById(“number_input”).value
 8               var remainder = numberAsText % 2
 9               if(numberAsText != 0 && remainder == 0){
10                     document.getElementById(“result_display”).innerHTML =
                              numberAsText + “ is even.”
11               }
12               else if(numberAsText != 0 && remainder == 1){
13                     document.getElementById(“result_display”).innerHTML =
                              numberAsText + “ is odd.”
14               }
15               else{
16                     document.getElementById(“result_display”).innerHTML =
                              “0 is neither even nor odd.”
17               }
18         }
19     </script>
20
21     <style>
22
23     </style>
24   </head>
25
26   <body>
27         <h1>Even or Odd???</h1>
28         Enter a Number: <input id=name_input></input>
29                 <button onclick="checkNumber()">Check It!</button>
30          <section id="result_display"></section>
31   </body>
32 </html>
```

Code Sample - even_or_odd.html

We’ll the code is fine and dandy, but what if your user does something stupid? What if they click the button without entering any thing? Oops. In that case, the ‘if’ and ‘else if’ comparisons are false and so the user is told “0 is neither even nor odd.” That’s going to confuse them. You’d better take care of that situation.

Dealing With Empty Input Boxes

I'm going to suggest you use something called 'embedded ifs'. I think you should take lines **8** through **17** and put them inside another if-else. Let me show you what I mean.

```
1  <!doctype html>
2  <html>
3    <head>
4      <title>Story Writer</title>
5      <script>
6          function checkNumber(){
7              var numberAsText = document.getElementById("number_input").value
8              if(numberAsText != ""){
9                  var remainder = numberAsText % 2
10                 if(numberAsText != 0 && remainder == 0){
11                     document.getElementById("result_display").innerHTML =
                           numberAsText + " is even."
12                 }
13                 else if(numberAsText != 0 && remainder == 1){
14                     document.getElementById("result_display").innerHTML =
                           numberAsText + " is odd."
15                 }
16                 else{
17                     document.getElementById("result_display").innerHTML =
                           "0 is neither even nor odd."
18                 }
19             }
20             else{
21                 document.getElementById("result_display").innerHTML =
                       "Please enter a number."
22             }
23         }
24     </script>
25
26     <style>
27
28     </style>
29   </head>
30
31   <body>
32       <h1>Even or Odd???</h1>
33       Enter a Number: <input id=name_input></input>
34             <button onclick="checkNumber()">Check It!</button>
35        <section id="result_display"></section>
36   </body>
37 </html>
```

Code Sample - even_or_odd.html

In this version of the even_or_odd.html Code Sample, I've added lines **8** and **19** - **22**. I also moved the if-else if-else code to the right

to make it easier for you to see that the if-else if-else code you wrote before only happens if line 8 is true.

Remain calm...you don't need to hold all the code code you see here in your head at one time. Like ogres and onions code has layers. You only need to deal with one layer of if's and else's at a time. Let me read line 8 to you. That and a brief explanation should help. It reads, "If what is in numberAsText is not some blank text..."

If the user leaves the input field on the page blank and clicks the button, numberAsText will be blank text. If numberAsText isn't blank then the user must have entered something. If they have, then check it to see if what they entered is odd or even. See? Only worry about lines 9 - 18 IF the user entered something. Move down a layer in your reading of the code only if the comparison check is true. Otherwise you can skip directly to the else for the if.

Pretend the user didn't enter anything and clicked the button. If that's the situation You're thinking about, you'd read line 7, and then 8, and then skip to line 20. You wouldn't need to worry about lines 9 - 18. Try reading through the current code sample several times using this situation. It will help you understand how the computer skips around when making choices.

Done that?? OK. Let's move on again.

Dealing With Stupid Stuff

This next bit is a little tricky. Not because the ideas are hard but because you're going to have to change multiple lines of your code.

So here's the situation. The user enters 'Sally' or some other text instead of a number! "Wait a minute Barney. User's won't do that. They know they are supposed to enter numbers!" Well sad experience tells us otherwise. If something can be done some user or bunch of users will do it. If you and I don't handle each situation our users can put us in then it will be us, not them, that look silly. Believe me, your life will be much simpler if you take care of as many of these situations as you can think of BEFORE you ship a product to your users.

You've probably had an application crash on you before. Almost every time an application crashes it is because whoever wrote the software didn't think of the situation you were in. Did you enjoy it when the app crashed? Nope. So the general rule of thumb, "***Don't do to your users what you don't like done to you***" applies. Try to find all of the possible situations the user could put your JavaScript in and deal with them BEFORE you ship your product.

Don't worry. There aren't an infinite number of situations. You can find them all if you think about what the user could do to your javaScript. See why you need to handle the user entering some text? Then let's get started and deal with it. To do this, you need to learn another pre-existing JavaScript function. It's called parseInt.

When you use JavaScript's parseInt function you send in to it some text and it gives you back an integer (that's an int in techie speak). If you've forgotten about functions, sending them stuff, and getting stuff back from them, go back to Section 1 and review what you learned there. If not, let's keep going.

So the parseInt function tries to convert any text to an integer. If you send it "1" it will give you back the number 1. If you send it

"253" it will give you back the number 253. "444" yields 444 and "-15" yields -15. Get the idea? But what if you send it "Sally"?

Well, the people who wrote the parseInt function did exactly what you've been doing. They sat down and tried to think of all the terrible, horrible, nasty things you could do to their function. And guess what? They put a bunch of comparisons and if-else if-else's in their function just like you are doing. They did their job well.

If you send "Sally" to parseInt it obviously can't covert it to a meaningful integer. Instead it gives you back something special named NaN. NaN stands for 'Not a Number.' This is an error indicating what you sent in is 'Not a Number.'

"Ah!"you exclaim. "Then in my code I could send into parseInt whatever the user types in and check to see if the result of calling parseInt is NaN!" Yep. You could. Good thinking. Anything they type in that can't be converted to a meaningful integer would cause parseInt to send you back NaN.

Now here is something to be careful of. You would think you could write the check you're thinking of like this.

```
var aNumber = parseInt(numberAsText)
if(aNumber == NaN{
        ......
}
```

Code Snippet 4

Or maybe even something like this.

```
var aNumber = parseInt(numberAsText)
if(aNumber != NaN){
        ......
}
```

Code Snippet 5

Well guess what. You can't. The JavaScript guru's decided not to allow that to work. They had their reasons but I'm not sure you want to know all of the technicalities of why they don't allow this.

"So then how do you do that kind of a comparison?" you wonder. They gave us another function to help us out. They called it 'isNaN'. Using it, the comparisons from Code Snippets 4 and 5 look like this

```
var aNumber = parseInt(numberAsText)
if(isNaN(aNumber)){
        ......
}
```

Code Snippet 6

and this.

```
var aNumber = parseInt(numberAsText)
if(!isNaN(aNumber)){
        ......
}
```

Code Snippet 7

Let me read them for you. The first is, “If what is in the aNumber variable is NaN...” and the second is, “If what is in the aNumber variable is not NaN...” I know. It reads inside out compared to == and != but it’s what we have to deal with.

It also uses the NOT operator we talked about before when you first had to deal with the != operator. Once again, the ! character always means NOT in JavaScript. Got it? You may need to go over this a couple of times and then try to explain it to someone else. That will help you understand parseInt, NaN, isNan, and ! better.

In the situation of your even_or_odd.html application here is how to think this through before you write the code. First, convert numberAsText to an integer using parseInt. If what it sends back is not NaN, then go ahead and check to see if what it sent back is even or odd. Otherwise tell the user they should enter a number.

Wow. that sounds almost the same as the check you just added. Remember??? The one about numberAsText not being “”??? I’d suggest using the AND operator in that check so you don’t have to add lines of code for an entirely different if-else. Here is how I’d suggest you do it. Hang in there. The major changes are only on lines **8** and **9** and they are nearly identical to what you saw in Code Snippet 7.

```
1  <!doctype html>
2  <html>
3    <head>
4      <title>Story Writer</title>
5      <script>
6          function checkNumber(){
7              var numberAsText = document.getElementById(“number_input”).value
8              var aNumber = parseInt(numberAsText)
9              if(numberAsText != “” && !isNaN(aNumber)){
10                 var remainder = numberAsText % 2
11                 if(aNumber != 0 && remainder == 0){
12                     document.getElementById(“result_display”).innerHTML =
                           aNumber + “ is even.”
13                 }
14                 else if(aNumber != 0 && remainder == 1){
15                     document.getElementById(“result_display”).innerHTML =
                           aNumber + “ is odd.”
16                 }
17                 else{
18                     document.getElementById(“result_display”).innerHTML =
                           “0 is neither even nor odd.”
19                 }
20             }
21             else{
22                 document.getElementById(“result_display”).innerHTML =
                       “Please enter a number.”
23             }
24         }
25     </script>
26
27     <style>
28
29     </style>
30   </head>
31
32   <body>
33       <h1>Even or Odd???</h1>
34       Enter a Number: <input id=name_input></input>
35           <button onclick="checkNumber()">Check It!</button>
36        <section id="result_display"></section>
37   </body>
38 </html>
```

Code Sample - even_or_odd.html

Details:

Line 8 reads, “Take what is in the numberAsText variable and try to parse (convert) it into an integer. Put what parseInt sends back out into a variable called ‘aNumber’.”

Line 9 reads, “If numberAsText is not blank AND what is in aNumber is a number....” Did you get that tricky bit in line 9? !isNaN reads “is a number.” Why??? Let’s take a look.

If something is not true it is false. If something is not false it it true. If only true and false are possible, the two previous statement must be right. Real life can get muddy. There may be times when something might be true or it might be false or it might be true and false at the same time. JavaScript is simpler than real life. It avoids Philosophy and only deals with true and false.

So if you used isNaN and sent it NaN, isNaN would send back true. NaN is NaN after all. If you sent 15 to isNaN it would send back false since 15 is a number. Got that bit? Think it over. It can take a while to sink in.

Let’s say your user enters 15 and clicks the button. On line 9 numberAsText is not equal to “” so that part of the condition works. the 15 is then passed into isNaN. It sends back ‘false’ since 15 is not NaN. You need this part of the condition to be true when 15 or some other number is sent into isNaN, so.....you have to apply the ! character (NOT) to force ‘false’ to be ‘true’.

“Wow. That’s complicated.” Yeah, I agree, but if’s and else if’s only work when conditions end up being true so we gotta do what we gotta do. After you do enough of these if, else if, else things you get used to thinking this way. Practice, practice, and even more practice is the way to change your thought process. Since you will need to do these kinds of checks every time you get a piece of information from your user you will have lots of practice.

One last thing and then we’re done. If your user types in -5 or any other negative, odd integer they get the “0 is neither even nor odd” message. Oops. I’ll tell you why. the mod operator, %, give you a negative remainder for any negative number. That means -7 %2 is going to be -1. Since your if statement on line 13 only checks to see if an number is equal to 1, your code, as it is now, will never tell your user that negative odd numbers are odd. There are several ways to solve this problem. I’m going to suggest the easiest way.

JavaScript has another built-in function that can help you out. It’s ‘Math.abs’. ‘Math.abs’ is sent a number and sends you back the absolute value of the number you sent in. -1 becomes 1. -500 becomes 500. 231 becomes 231. Essentially, any negative number is converted to a positive number and positive numbers aren’t changed. I think you should use Math.abs.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Story Writer</title>
5     <script>
6         function checkNumber(){
7             var numberAsText = document.getElementById(“number_input”).value
8             var aNumber = parseInt(numberAsText)
9             if(numberAsText != “” && !isNaN(aNumber)){
10                var remainder = Math.abs(aNumber % 2)
11                if(aNumber != 0 && remainder == 0){
12                    document.getElementById(“result_display”).innerHTML =
                          aNumber + “ is even.”
13                }
14                else if(aNumber != 0 && remainder == 1){
15                    document.getElementById(“result_display”).innerHTML =
                          aNumber + “ is odd.”
16                }
17                else{
18                    document.getElementById(“result_display”).innerHTML =
                          “0 is neither even nor odd.”
19                }
20            }
21            else{
22                document.getElementById(“result_display”).innerHTML =
                      “Please enter a number.”
23            }
24        }
25    </script>
26
27    <style>
28
28    </style>
30  </head>
31
32   <body>
33       <h1>Even or Odd???</h1>
34       Enter a Number: <input id=name_input></input>
35              <button onclick="checkNumber()">Check It!</button>
36        <section id="result_display"></section>
37   </body>
38 </html>
```

Code Sample - even_or_odd.html

Details:

Take a look at line **10**. It’s changed. Now it reads differently because of the change. In English it is, “Calculate numberAsText modulo two. Then get the absolute value of the result and put that positive value into the remainder variable.”

Got it? Line **10** has three steps instead of the two it used to have. “Why,”you ask, “do you read what’s inside the parenthesis first?” Again, good question. Remember back in junior high or grade school when your teacher taught you about order of operations in your math class and you said, “When am I ever going to need that?!” Well guess what, today is your lucky day. The order of operations rules she tried to teach you included this one, “Always do what is inside parenthesis first.” ‘number % 2’ is inside parenthesis. That means the computer will always calculate number % 2 BEFORE it does the absolute value stuff.

OK. We’re done. That’s an awful lot of stuff to know. I’d suggest going over this section several times. I’d also suggest creating your own app that gets a value from your user and then checks to make sure they didn’t do something stupid. Practice, practice, and more practice is the way to understand this so....

STOP HERE AND GO PLAY. PLEASE.

That was complicated. Let's take a little break from JavaScript and improve your user's experience with your app. After all, it looks awful.

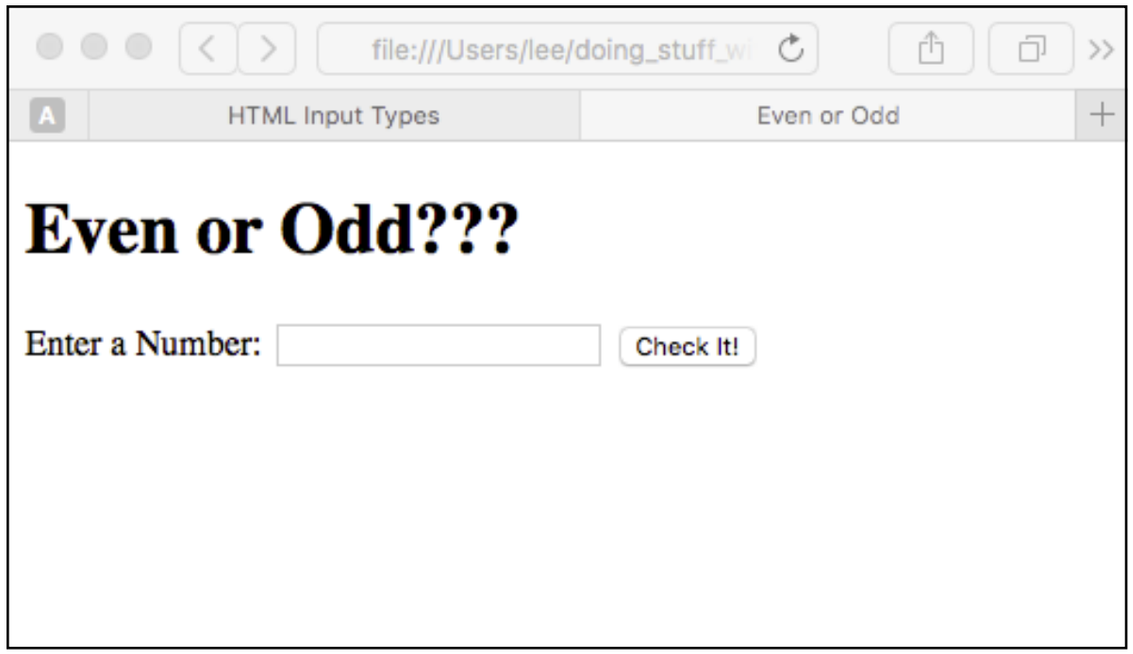


Figure 1. *The even_or_odd.html page with a bad user experience.*

The elements are in good spots on the page, but yuck!! Let's go add some CSS so it looks nicer. I think if you could get the page to look like Figure 2 it would be much more enjoyable and you get to learn more about CSS too! So let's do it.

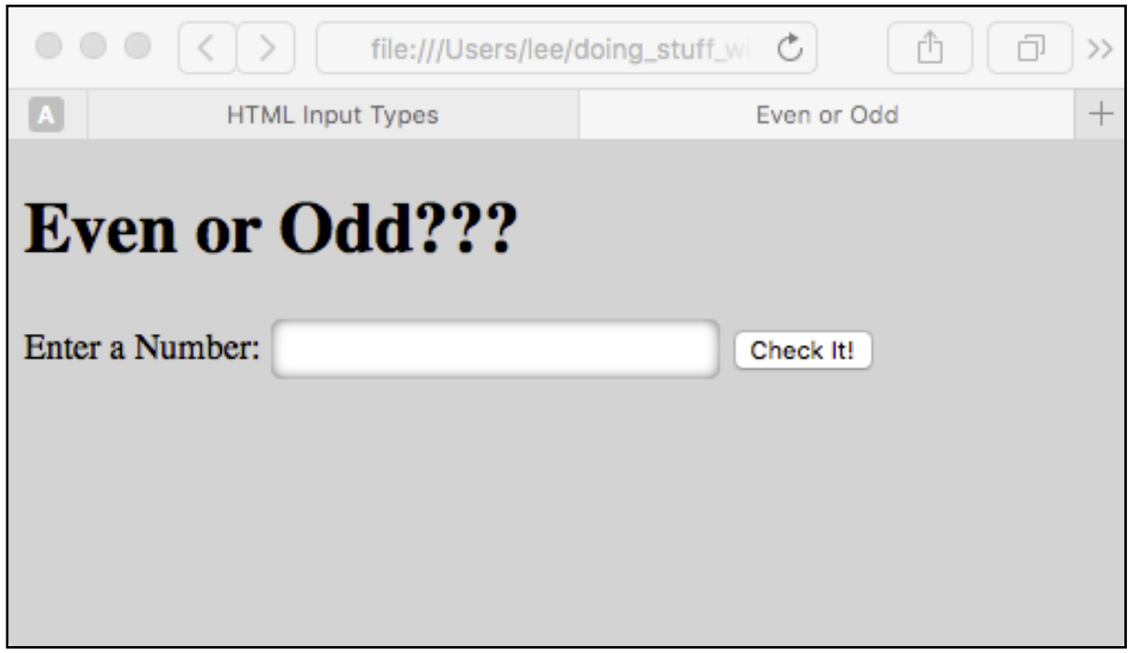


Figure 2. *The even_or_odd.html page with a better user experience.*

You can find the completed version of even_or_odd.html in the zip file download. In this book I'll only be showing you what is inside the style starting and ending tags for this page. If you want to see the code for the whole page, look in even_or_odd.html in the zip file.

So let's get started and begin with something you already know, setting the page's background color. To do this, just like you did before, set the background color of the body.

```
<style>
        body{
            background-color: LightGrey;
        }
</style>
```

Code Snippet 1

The only difference here is the color is set to LightGrey. Feel free to choose any other color you'd like. Now let's work on the input box. After all, square corners with a single pixel black border is so 1990's.

Let's start by fixing a height and width so those won't change when things get more interesting. You do this the same way you did it when you set the height and width for the section in Section 2.

```
<style>
        body{
            background-color: LightGrey;
        }
        input{
            height: 25px;
            width: 200px;
        }
</style>
```

Code Snippet 2

I think a height of 25 pixels and a width of 200 pixels looks about right to me but you can choose what you want.

With the stuff you already know out of the way, let's play with some new stuff. First, let's get rid of that ugly 1 pixel black border by turning off the input's border completely. "Whoa..." you say. "If you do that they can't see where to type." Well, they could. It would be white on a grey background. We'll work up to better looking stuff, but here is the CSS and what the page looks like so far.

```
<style>
        body{
            background-color: LightGrey;
        }
        input{
            border: none;
            height: 25px;
            width: 200px;
        }
</style>
```

Code Snippet 3

The CSS reads, "Hey all body tags, set your background color to be LightGrey." and "Hey all input tags, turn off your border, set your height to be 25 pixels, and set your width to be 200 pixels."

So far, the page looks like this.

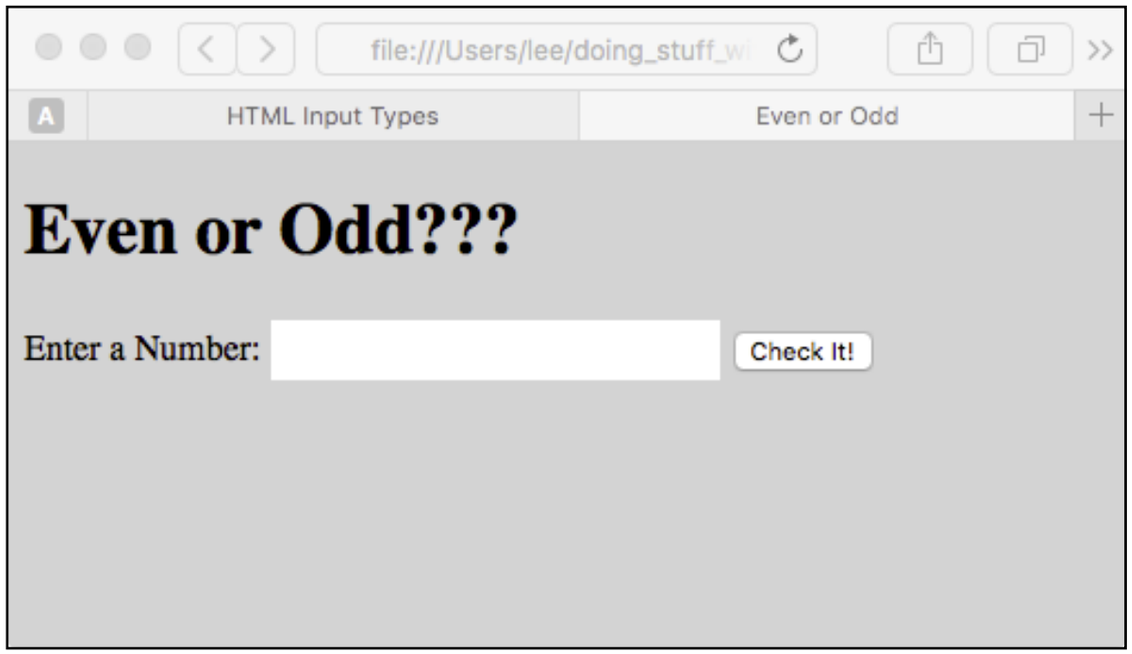


Figure 2. *The even_or_odd.html page part way done.*

The input box still looks square and blocky. Let's round off its corners next. You do this with the border-radius property. If you imagine the corners as having circles over them and the circle is touching the top and side this will make more sense. You're imagined version should look something like Figure 3.

Figure 3. *The even_or_odd.html page with imagined circles.*

The circles I've added to Figure 3 each have a radius of 5 pixels. If you were to trim off off the corners poking out beyond the circles you'd have an input box with rounded corners. That's exactly what will happen if you set the border radius property of the input tags to be 5 pixels.

```
<style>
        body{
            background-color: LightGrey;
        }
        input{
            border: none;
            border-radius: 5px;
            height: 25px;
            width: 200px;
        }
</style>
```

Code Snippet 4

One of the nice things about border radius is it applies even when you've turned borders off. Now the page is getting close.

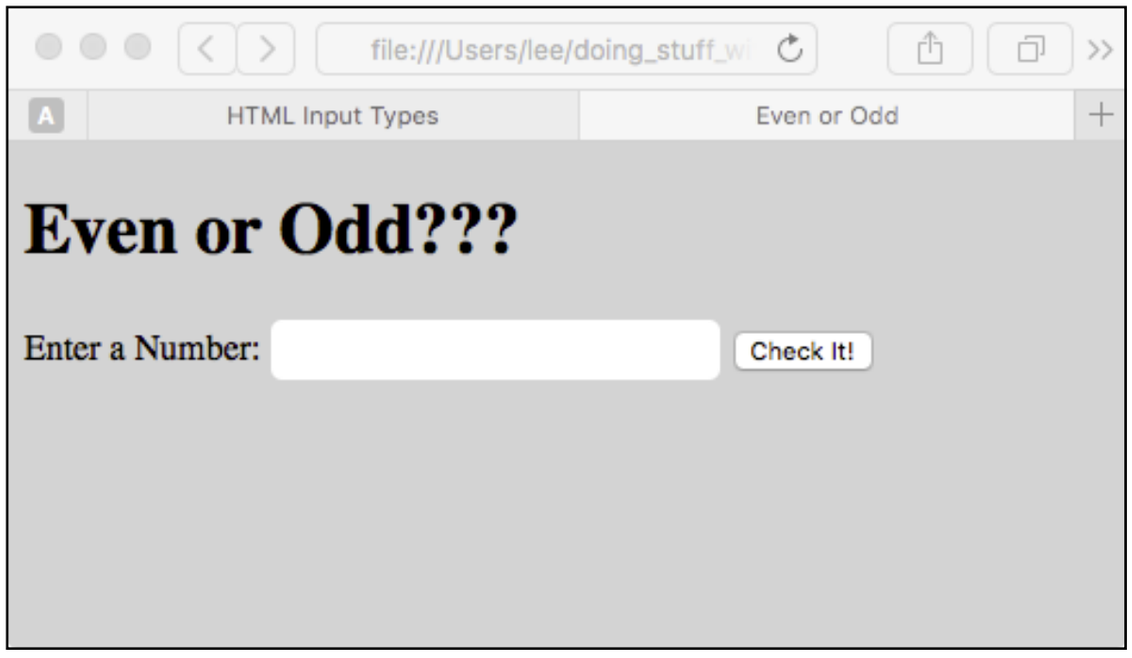


Figure 4. *The even_or_odd.html page with border radius at 5 pixels.*

One thing about adding border radii, if the user tries to type in a number right now, part of the left-most digit can get cut off by the

rounded corners. Let's quickly fix that. The CSS text-indent property is the way to go.

```
<style>
        body{
            background-color: LightGrey;
        }
        input{
            border: none;
            border-radius: 5px;
            height: 25px;
            width: 200px;
            text-indent: 5px;
        }
</style>
```

Code Snippet 5

You can choose any sized indent, I've played with this page and settled on 5 pixels. I think, on this page, 5 pixels looks good.

Now for the last bit. In Figure 2, the input box has a fuzzy edge to it. It is actually a shadow. It is common to use shadows to show a height change in pages. Take a look at Code Snippet 5 box. It looks like it is above the rest of the text in this chapter because of the shadow that has been added to the box. For the input box, let's put the shadow inside the box instead of outside. This makes it look like the box is lower that the rest of the page. CSS's box shadow property is what you'll use. Code Snippet 6 has an example.

```
<style>
        body{
            background-color: LightGrey;
        }
        input{
            border: none;
            border-radius: 5px;
            box-shadow: 0px 0px 5px #666666 inset;
            height: 25px;
            width: 200px;
            text-indent: 5px;
        }
</style>
```

Code Snippet 6

The box shadow CSS property has five things you can send to it, much like you send things into a function. The first two things are required, the rest are optional. I'll describe the five things sent to box shadow for you.

The first is how far the shadow is shifted horizontally.

The second is how far the shadow is shifted vertically.

The larger these two values are, the more the element they are applied to appears to be above or below the page. Code Snippet 6 has these values set to zero pixels.

The third value sent to box shadow is how many pixels should be used to blur out the shadow. Rarely do real-life shadows have a shape that exactly matches the object. Instead, their edges are blurry. The smaller the number you put for this value the less blurry the shadow. Code Snippet 6 uses a value of 5 pixels.

The fourth value sent to box shadow is the shadow color before it is blurred.

Don't let the #666666 throw you off. All it does is describe a custom, non-named color. If you take a look at some named web colors you will see they can be described by their name or a hexadecimal number. #666666 is a hexadecimal number. Using hexadecimal descriptions, White is #FFFFFF, Black is #000000, and the color I used, #666666 is 40% of the way from Black to White. There are lots of places online that explain how the computer interprets these hexadecimal numbers as CSS web colors so I won't explain them here. As you know, this section is already WAY too long. Feel free to explore them on your own.

Last of all is the indicator saying that you want the shadow on the inside of the box. If you want the shadow on the outside, get rid of this value.

So let's read the box shadow property. "Hey input, have a shadow that has no horizontal offset, no vertical offset, blur the shadow over 5 pixels, use a grey color that is more towards black than white, and put the shadow inside all input boxes."

The result looks like this.

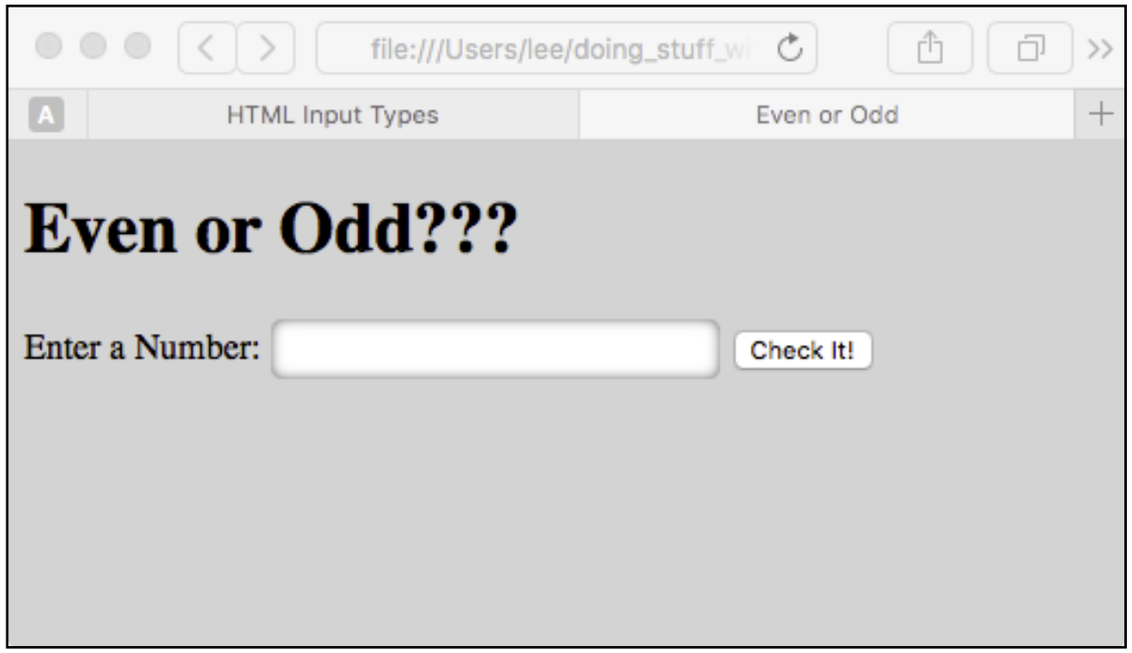


Figure 5. *The final version of even_or_odd.html page.*

So that is how computers make choices. Your code can tell the computer how to make do stuff based on situations by comparing and using if-else if-else. No magic here, but it can get complicated REALLY fast so be careful. There are other options when doing comparisons beyond AND (&&). You will sometimes need OR (||) and infrequently EXCLUSIVE OR (^). You can google for examples of those to find out more.

Groups and Loops

Lorem Ipsum

1. What is an array?
2. How do you make an array?
3. How do you use an array?
4. What are JavaScript's two types of loops used with arrays and when are they used?

Two geese in this image are different. Unlike some other languages, groups of things in JavaScript, called arrays, don't require all of their elements to be of the same type.

So what if you need to represent a bunch of things instead of using a variable to hold one thing? Maybe you need to represent a bunch of numbers. You might be averaging two years of monthly income in an awesome application. Is there a way to do that without creating a whole bunch of different variables? Yep. There is. You'd think someone thought through this problem before. :)

In JavaScript groups of things, like the the group of numbers in your app representing income by month, are called arrays. This is one of only two pre-existing ways to group things in JavaScript. You could write code to create other types

groups. If you wanted to you could make things techies call lists and trees. We'll stick with arrays for now. OK, lets get started by looking at how you can create an array.

```
7   var someNumbers = [25, 11, -2, 14,
                       "what?? This isn't a number!", -1, -10]
```

Code Snippet - group_loop.html

Line 7 in the group_loop.html example shows how to make an array. It reads a little backwards. "Create an array of seven things, put 25, 11, -2, 14, 'what?? this isn't a number!', -1, and -10 in the array, and then put the array in the someNumbers variable."

Any time you put something in a group you eventually need to get something out of the group. When you put dollars, euros, or what ever your currency is into a bank, don't you eventually intend to take them back out?

How then can you get a value out of an array? Code Snippet 1 shows how.

```
var aNumber = someNumbers[1];
```

Code Snippet 1

It reads, "Use the array in the someNumbers variable and get the value at the ' one'th position. Then put what you find in the aNumber variable." Now you might think aNumber would now have 25 in it. That is a very logical conclusion. It actually has 11 in it. "What?! That makes no sense!"

We'll....actually it does, just not in the way you were thinking. Notice I wrote "one'th" rather than "first." Hmm... tricky. Arrays in JavaScript, like in some other languages, use zero to represent their first position instead of 1. Code snippet2 shows how to get aNumber to hold the array's first value, 25.

```
var aNumber = someNumbers[0];
```

Code Snippet 2

This means that even though there are seven elements in the someNumbers array, assigning the last element of the someNumbers array, -10, to aNumber is done using a 6. Code snippet 3 shows you how to do this.

```
double aNumber = someNumbers[6];
```

Code Snippet 3

Weird...I know, but you get used to it if you work with JavaScript arrays long enough. In JavaScript arrays, the elements in the array are kept in the order they were added. That way you can always get back out the one you want. All you have to do is remember which one you want. Sometimes that's not as easy as it sounds.

Now...lets think about combining arrays and the even_or_odd.html example from the Making Choices section. This means you need

some way of checking a bunch of numbers for being odd or even. You could do it really poorly by repeatedly copying and pasting the even/odd check code for each number in an array, or you could do it with what techies call 'a loop'.

In programming, a loop is something that makes stuff happen over and over again. This means you could use a loop do the check over and over again with a different number each time.

The type of loop specifically designed to work with arrays is called a 'for loop.' It is called this because when you create one you use the word 'for.' Line 8 in Code Snippet 4 shows how to create a for loop that would work with the array in someNumbers on line 7. Line 8 has the 5 distinct parts of all well-formed for loops.

```
7  var someNumbers = [25, 11, -2, 14,
                                  "what??", -1, -10]
    1       2       3     4
8  for (var i = 0; i < 7; i++){
       .
       . 5
       .
26 }
```

Code Snippet 4

The standard for loop parts when working with an array are:

1. the for keyword,

2. a variable to keep track of the location you are at in the array,

3. a check to see if the loop should continue,

4. an update that increases the location tracking variable by one, and

5. the lines of code to execute every time you go around the loop, lines 8 though 25 in Code Snippet 4.

Line 8 reads, "create a for loop where i, the location tracking variable, starts out as zero, the loop stops if i reaches seven, i goes up by one each time the loop goes around, and the code in the {} characters is also executed each time the loop goes around."

Line 7 is a standard structure in JavaScript. By tradition, the location tracker is named 'i'. It stands for 'index.' No need to memorize this little bit of information. It's just some cultural trivia. It does defy the 'name things what they are' and 'don't use single character variable names' rules. But culture is culture and cultures have strange things in them. In my culture people go around asking each other 'How are you?' when they don't actually care. What they really mean is 'Greetings.' Weird isn't it?

Using i in for loops is part of techie culture. Go ahead and use i instead of 'index'. Everyone will know what you mean if they are part of the techie culture.

So let's put the group loop code all together now and call the result group_loop.html.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Group Loop</title>
5     <script>
6         function checkNumbers(){
7               var someNumbers = [25, 11, -2, 14, "What?? This isn't a number!", -1, -10]
8               for(var i = 0; i < 7; i ++){
9                   var numberAsText = someNumbers[i]
10                  var aNumber = parseInt(numberAsText)
11                  if(numberAsText != "" && !isNaN(aNumber)){
12                        var remainder = Math.abs(aNumber % 2)
13                        if(aNumber != 0 && remainder == 0){
14                              document.getElementById("result_display").innerHTML +=
                                      "<p>" + aNumber + " is even.</p>"
15                        }
16                        else if(aNumber != 0 && remainder == 1){
17                              document.getElementById("result_display").innerHTML +=
                                      "<p>" +aNumber + " is odd."
18                        }
18                        else{
20                              document.getElementById("result_display").innerHTML +=
                                      "<p>0 is neither even nor odd.</p>"
21                        }
22                  }
23                  else{
24                        document.getElementById("result_display").innerHTML +=
                                      "<p>Please enter a number.</p>"
25                  }
26            }
27        }
28    </script>
29
30    <style>
31
32    </style>
33  </head>
34
35   <body>
36       <h1>Even or Odd???</h1>
37             <button onclick="checkNumbers()">Check It!</button>
38        <section id="result_display"></section>
39   </body>
40 </html>
```

Code Sample - group_loop.html

Details:

You've looked how to do the odd or even check before so I won't go over it again. If you skipped Section 4 or are confused by how the odd or even check works, go back and understand Section 4.

The major difference between what you saw in Section 4 and what is in group_loop.html is the check has been put inside of a for loop. That way it can be applied to all of the things in the someNumbers array. Don't let the bigger number of lines of code confuse you. Remember, all your doing is checking each thing in someNumbers to see if it is odd or even and you'r using a loop so you don't have to duplicate lines **10** - **25** over and over again.

Don't forget, the variable i is used to keep track of where you are in the array as you go through the things in the array inside of someNumbers. In programming speak, going through the things in an array is called 'iterating over the array.' You will hear that phrase a lot in programming books and tutorials. Ready? Lets dive in.

You've already seen most of what is new in Code Snippet 4 but line **9** is new so let's start there. It reads, "get the i^{th} thing from the array in the someNumbers variable and put it in the aNumber variable." Now, i will start out being zero and then because you used i++ in line 8, it with then increase AFTER the $zero^{th}$ thing in the array is checked. So the second time lines 9 - 25 run, i will be 1. Then it will be 2, 3, 4, 5, and 6. This means each thing in the array will eventually be put in the aNumber variable and checked to see if it is even or odd.

Think this through a couple of times. It also helps if you double click the group_loop.html example file and see each thing in the array being checked.

"That's not all that's new here," you claim. "What's that '+=' thing on line 14?" Wow. You're really paying attention. Thanks for asking.

'+=' is an operator you haven't seen before. It is similar to the + operator. Imagine you had a variable called aNumber with the number 17 in it. If you used aNumber and wrote a line of code like this

aNumber += 3

aNumber would now have a 20 in it. The += operator takes the thing on its right-hand side and adds it to whatever is in the variable on the left-hand side. Let's try it again.

aNumber += 10

Since aNumber had the number 20 in it because of the last time you used +=, aNumber will now have the number 30 in it.

That's how += works when you are dealing with numbers. += increases the number in the variable by the number on its right-hand side. With text, the += operator works a little differently. This time imagine you have a variable called aName and it has "Lee" in it. If you used aName and wrote a line of code like this

aName += " Barney"

aName would now have "Lee Barney" in it. Let's do that again.

aName += " wrote this book."

Since aName already had the text "Lee Barney" in it, it would now have the text "Lee Barney wrote this book." in it. This is similar to using += with numbers. With numbers += adds to the number in the variable. With text += adds to the text in the variable. Techies would say += 'appends' the text on its right-hand side to the text in the variable.

Here's the code from line **14** of group loop. I'll read it to you in English.

```
document.getElementById("result_display").innerHTML +=
                        "<p>" + aNumber + " is even.</p>"
```

Code Snippet 5

"Hey document, get the element who's id is 'result_display' and append some text to what is already between its starting and ending tags. The text to append is a paragraph tag then the number in aNumber and the text 'is even.' End the text with an ending paragraph tag."

Lines **17**, **20**, and **24** also append to what is already between the starting and ending tags of result_display. In my browser, the group_loop.html page looks like this.

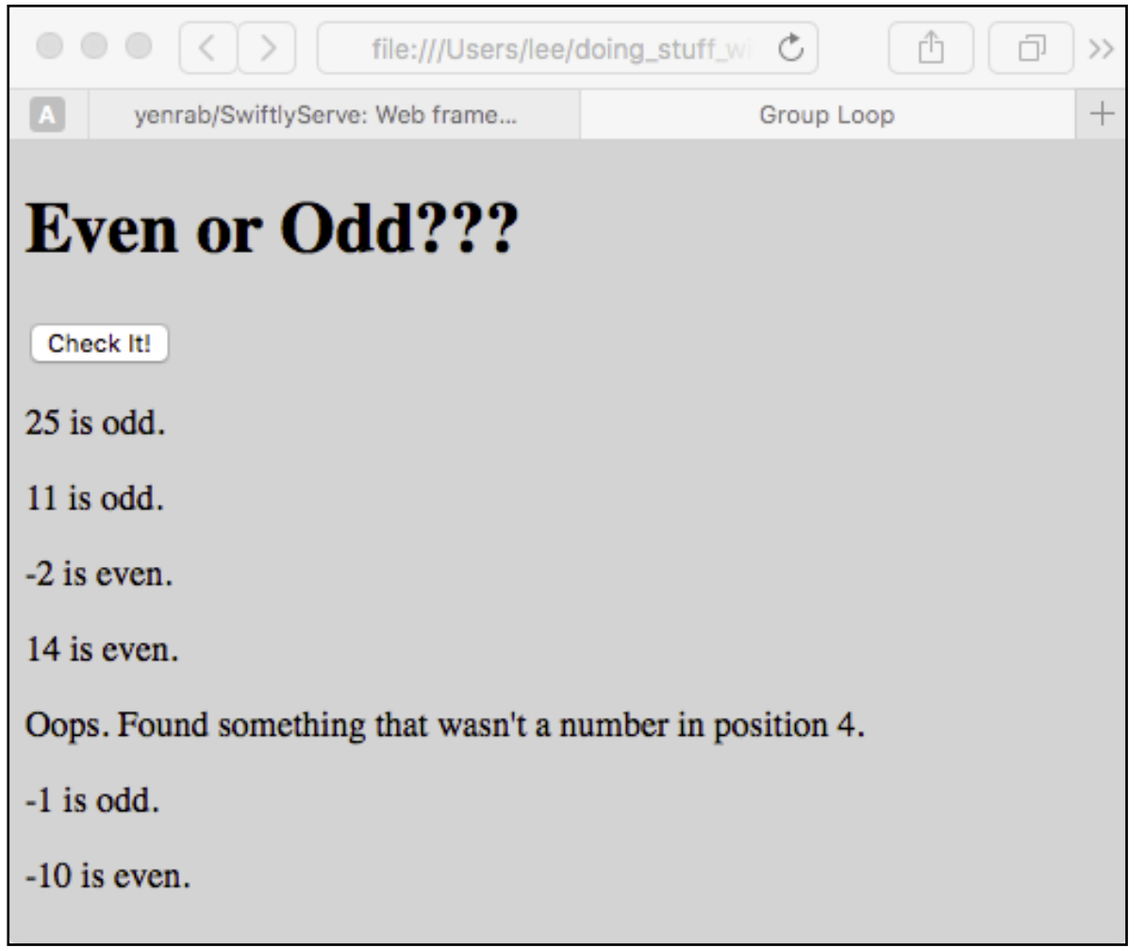


Figure 1. *The group_loop.html page.*

Be careful. If you accidentally use = instead of += you will end up only seeing "-10 is even." (Figure 2) instead of the results of evaluating all the things in the someNumbers array (Figure 1). "Why?" you ask. Remember, = assigns stuff to variables and += appends to or adds stuff to variables.

Figure 2. *The group_loop.html page with the = errors in the code.*

So at this point, you should understand what an array is, have seen how to iterate over the things in an array using a for loop, and have learned a new operator +=. That's a lot of stuff. You should stop reading here and create your own page that uses an array of things. Take your time. Getting through the book isn't important. Understanding what you are doing is vitally important.

STOP HERE AND GO PLAY. PLEASE.

"This is fine and dandy, Barney, but not very interesting. Who wants an app that checks numbers for being even or odd?" You are right. Let's see if we can liven this up a bit.

A Journaling App

This isn't what you might think. An app for keeping a diary is only one type of journaling application. There are lots of others. If you've ever used an app to write yourself notes or reminders, it was a journaling app. If you've used an app to track contacts with customers for your company, it was a journaling app. Even apps that track expenses or time spent on work projects are journaling apps. Pretty much any app that has entries, creates a time-stamp for the entry, and then stores the entry and the time stamp together is a journaling app. So what kind of journaling app do you want to create today? I'll help you create one that tracks notes. It's one of the least complicated kinds of journaling apps to create. This first version won't be really fancy, but in Section 6 you'll step it up and make it nicer.

Let's follow the same path you went down as you created the app in Section 4; start with the HTML, add in the JavaScript, and then improve the user experience by adding CSS. A lot of the things we're going to use, you already know if you've read, experimented with, and understood the stuff in the previous sections. If you are running into problems with the stuff that isn't explained in this section I'd suggest going back and reviewing.

Using the template, create a page called note_it.html. When you're all done adding in the HTML and double click the note_it.html file the page in your browser will look like Figure 3.

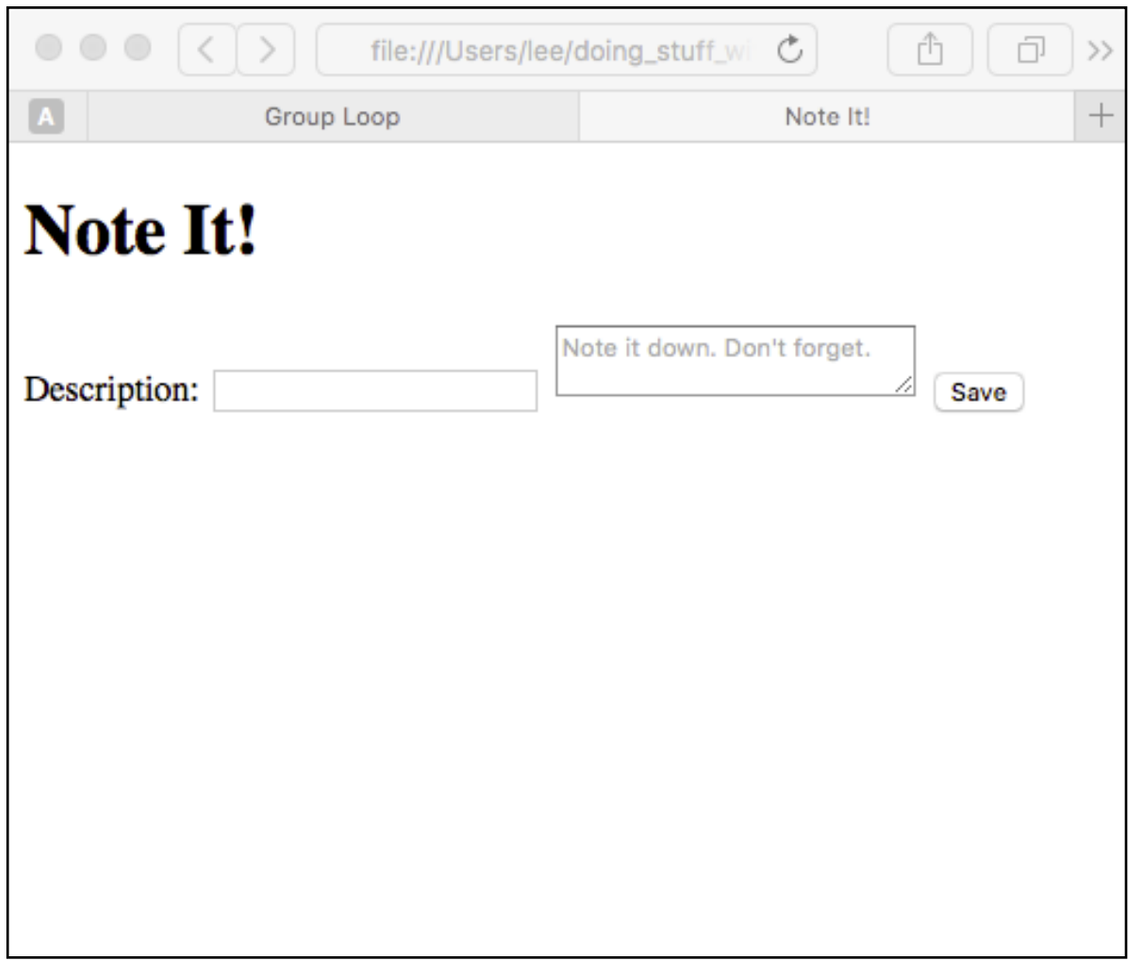


Figure 3. *The note_it.html page without any CSs or JavaScript.*

With what you've learned before, the HTML for this page shouldn't be shocking.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>Group Loop</title>
 5     <script>
 6     </script>
 7
 8     <style>
 9
10      </style>
11   </head>
12
13   <body>
14       <section id="note_entry_area">
15         <h1>Note It!</h1>
16         Description: <input id="description_input"></input>
17         <textarea id="note_editor" placeholder="Note it down. Don't forget."></textarea>
18         <button onclick="saveNote()">Save</button>
19       </section>
20       <section id="all_notes_display"></section>
21   </body>
22 </html>
```

Code Sample - note_it.html

There is one thing on line **17** that is new. You've seen the id property several times before, but the placeholder property is new to you. It's a way to have some text appear in the textarea that goes away as soon as you start typing inside the textarea. Go ahead. Try it. Click on 'Note it down. Don't forget' in the page in your browser. and type See? It went away and you're entering a note. A nice touch in HTML when you want to explain to the user what they need to do.

"Hey Barney, why did you put the input, textarea, and button between a section's starting and ending tags?" I thought you'd see that. I did that because I want to do some CSs for the entire note entry area and the section will make this easier later on. I'll show you that after you're done with adding JavaScript into the file.

So let's start adding in the JavaScript now. Nobody wants to re-enter every note they wrote previously. That means you need to create a function to store all of the notes the user writes. In line with the 'call things what they are and what they do' rule, call this function saveNote. Don't worry, we'll go through this code together just like we did before. Some things and ideas will be new, but that's OK. You've dealt with new things before.

Making It Do Stuff

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Group Loop</title>
5     <script>
6
7     function saveNote(){
8               var currentDateAndTime = new Date()
9               var aNoteDescription = document.getElementById(“description_input”).value
10              var aNoteText = document.getElementById(“note_editor”).value
11              var aCompleteNote = currentDateAndTime.toLocaleString()
                                                        + "--" + aNoteDescription
12              aCompleteNote += “<p>” + aNoteText + “</p>”
13
14              var storedNotesString = localStorage.getItem(“all_notes”)
15              var allNotes = JSON.parse(storedNotesString)
16              if(allNotes == null){
17                      allNotes = []
18              }
19              allNotes.push(aCompleteNote)
20              var allNotesString = JSON.stringify(allNotes)
21              localStorage.setItem(“all_notes”, allNotesString)
22    }
23    </script>
24    <style>
25
26      </style>
27  </head>
28
29    <body>
30        <section id="note_entry_area">
31          <h1>Note It!</h1>
32          Description: <input id="description_input"></input>
33          <textarea id="note_editor" placeholder="Note it down. Don't forget."></textarea>
34          <button onclick="saveNote()">Save</button>
35        </section>
36        <section id="all_notes_display"></section>
37    </body>
38 </html>
```

Details:

All right, let’s get started. I’m going to show you the new parts in different Code Snippets. If you get lost, you should come back here to see how all this fits together.

The **saveNote** function starts on line **7** and ends on line **22**. That’s easy. Something new happens on line **8**. It reads, “Hey page, create a new date and time stamp and put it in the currentDateAndTime variable.” The date and time stamp will remember the date and time when it was created down to the millisecond. You don’t care about that level of accuracy for the Note It! app, but there may come a time in the future when you will. For now, just remember ‘new Date()’ creates a new date and time stamp.

Lines **9** and **10** are nothing new. You’ve seen this kind of code several times before (If you forgot what they are, go take a look at Section 3). Line **11** is where you’ll use the date and time stamp you created on line **8**. In English it is, “Get what is in the currentDateAndTime variable and convert it into a string depending on my current locale. Then append ‘--’ and what is in the aNoteDescription variable. After completing this, put the result into the aCompleteNote variable.” If you are using the Chrome browser and typed ‘stuff’ into description_input, what’s in the aCompleteNote variable is this text with the date and time you click the button instead of the one you see here.

5/9/2016, 9:40:41 AM--stuff

You can tell when I clicked the button, it was just after 9:40 in the morning in my office. “But there are different time zones all over the world Barney. Oh, and what about daylight savings time?”

Don’t worry. The toLocaleString function that converted the date and time stamp to a string already knows where you are and if it is daylight savings time or not. Nice. No need to worry. Where ever you are the time that ends up being part of your note will be the correct, current local time.

If you use a different browser the date and time string may look a little different, but it would contain the same information. For example, if I clicked the button in my Safari browser, on line **11** aCompleteNote would have this in it.

May 9, 2016 at 9:40:41 AM MDT--stuff

See? This is the same information just formatted differently. Safari did add in that I'm in the Mountain time zone of the United States and that it is currently daylight savings time (MDT) but that's a small change.

With line **11** understood you should now move on. Line **12** has the += operator you learned about earlier in this section. If you remember, since you are dealing with text it will append whatever it finds on it's right side to whatever it finds in the variable on its left and put it in the variable it finds on its left side.

Line **12** reads,"Hey browser, make some text that is a starting paragraph tag, append to that little bit of text whatever you find in the aNoteText variable, then also append an ending paragraph tag. Then take that whole piece of text, append it to what is in the aCompleteNote variable and put everything back in the aComplete Note variable."

I know, I know... that's a lot of stuff. The idea is that whatever the user types in for the note should be its own paragraph separate from the date, time, and description. If you typed "I won't forget this." into note_editor before you clicked the Save button, aCompleteNote would now have this in it.

May 9, 2016 at 9:40:41 AM MDT--stuff I won't forget this.

Again, what happened was aCompleteNote started off with **May 9, 2016 at 9:40:41 AM MDT--stuff** in it because of line **11**. Then line **12** added what was in the aNoteText variable surrounded by paragraph starting and ending tags to what was already in aCompleteNote. So at this point, aCompleteNote has this in it.

May 9, 2016 at 9:40:41 AM MDT--stuff I won't forget this.

It's OK if you need to go over lines **11** and **12** a few times. Make sure you understand them. When writing apps you're going to be doing this sort of thing a lot.

Line **14** is where you use the local storage stuff you learned about earlier in this section.

```
14    var storedNotesString = localStorage.getItem("all_notes")
```

Code Snippet 6

In English it is, "Hey, local storage, get the stuff that was stored with the key 'all_notes' and put it in the storedNotesString variable. Got it? Then let's move on to line **15**. That is where you need to learn something new again.

In Section 4 you stored individual pieces of stuff into local storage. Since you're going to need to store an array of messages for this app, you have to have a way to convert an array into a single thing. For a number of years, JSON (Javascript Object Notation) has been the way this is done.

Essentially, JSON is used to covert anything to text. It does the conversion in such a way that the text can then be converted back to the original thing. Let's look at an array as an example.

If you had this array, [5, -3, 'hello', 'there'], it would be converted to this JSON text "[5,-3,'hello','there']". Make sure you notice both of the " characters. They mean the array is now text. It looks a lot like the array doesn't it? That's on purpose. That way you can read it, and it makes it easier for the computer to turn the text back into a JavaScript array. In techie speech, text in a program is called a string of characters or a string. Usually they always use the word 'string' when they mean text. From this point on in the book we will do this too.

Now...line **15** converts the string containing representing the array of stored notes in the storedNotesString variable back into an actual JavaScript array.

```
15      var allNotes = JSON.parse(storedNotesString)
```

Code Snippet 6

How can you know this? Because of the word 'parse'. In techie speech, parse means to examine a string character by character to determine what the string means. When you were learning to read and sounded out words like 'dog', you were parsing. You examined and pronounced each character of the word. Then you made meaning out of the sounds you heard. With this understanding of parse I'll read you what line **15** says.

"Hey JSON parse the string you find in the storedNotesString variable and put the result into the allNotes variable."

"That's fine and dandy Barney, but can parse make sense of something like 'Go to the store and buy a loaf of bread?' If it can, then why don't we all write code that looks like that?" Well...because it can't make sense of 'Go to the store and buy a loaf of bread'. It can only make sense of strings that match an international standardized pattern called JSON (Javascript Object Notation). Basically, if the string wasn't created by some pre-written JSON code, a JSON library in techie speak, or by a person who knows the standard very well, JSON.parse can't understand the string and will fail. This means you have to be very careful if you are typing in the strings yourself. Thankfully, in JavaScript the JSON library has a way to create those strings for you. You'll see how to do that when you get to line **21**.

Let's imagine what is going to happen if the user has never stored any messages. This is going to happen the first time they click Save, after all. "If they've never stored anything, then what will localStorage.getItem give me back?" you ask. It will give you back nothing but it is a special kind of nothing. It is called 'null'.

If you haven't been exposed to null before it can be kind of tricky to grasp. Imagine you had an egg basket with a bunch of eggs in it. If someone asked you what you had, you'd answer, "A basket of eggs." If you had the same basket without any eggs in it you'd answer, "An egg basket". If you didn't even have the egg basket, you'd answer, "Nothing." There is a difference between having an egg basket with no eggs in it and not having an egg basket. In te-

chie speak you egg basket would be null (nothing). Got it?

```
14    var storedNotesString = localStorage.getItem("all_notes")
```

Code Snippet 7

When line **14** of your code used local storage's getItem function your code expected to get back a string representing an array of messages. If no array of messages was stored, then getItem can't find anything to give back to your JavaScript. It is forced to give back nothing so it gives back null, the special indicator of nothing in JavaScript. As part of line **14**, null is put into the storedNotes-String variable in this situation.

That means on line **15**, JSON.parse was told to try to understand null (nothing). It turns out when your code asks parse to make sense of null, it says, "Hey, I can't do anything with this. I'll just give back nothing (null) in return." So allNotes, which your code expects to be holding an array, holds null instead.

Lines **16** - **18** fix this problem.

```
16      if(allNotes == null){
17            allNotes = []
18      }
```

Code Snippet 8

Line **16** says,"If what's in the allNotes variable is equal to null, put an empty array into the allNotes variable."

"Why does the JavaScript need to have this in it?" you ask. One way or another, you're going to need an array to hold notes. Line **14** and **15** end up creating an array if notes were stored. If they weren't, then lines **16** - **18** create an empty array that you'll store later on. Think about this for a while.

Here is a real world example of the same type of situation. You are at a store and ready to pay for your purchases. The store owner says, "Did you bring your reusable bag?" If you did bring it, your purchases go in your bag. If you didn't bring it, you need a bag to put your purchases in. A new bag is provided by the store owner and your groceries are put in the bag. Whether or not you brought a bag with you, you end up with your purchases in a bag as you leave the store. In the code you are working on, line **19** is where 'your purchases are put in the bag.'

```
19      allNotes.push(aCompleteNote)
```

Code Snippet 9

Line **19** says,"Push aCompleteNote onto the end of the allNotes array." With the note added to the array, now you need to store the

allNotes array so the new note isn't lost. JSON and local storage come to your rescue.

```
20        var allNotesString = JSON.stringify(allNotes)
21        localStorage.setItem("all_notes", allNotesString)
```

Code Snippet 10

JSON has a function that converts stuff into a string so localStorage can store the string. The JSON function is called *stringify*. Line **20** uses stringify. It says,"Hey JSON, convert everything in the array in the allNotes variable into one string and put the string you create into the allNotesString variable."

Line **21** stores the string. You've seen this kind of JavaScript before but lets read through it again. It says, "Hey localStorage, store the allNotesString using the key 'all_notes' so I can get it back later."

Getting Stuff Back Out And Showing It

Wow, that was a lot of new ideas and JavaScript. It works great but your user still can't see the stored notes. Let's fix that problem. For now, let's add another button, create another function, and when the button is clicked run the function. The function will get all of the stored notes, clean out any currently displayed notes, and then add in each stored note.

The code is getting too long to display it all at once, so I've hidden lines **8** - **21**.

```
1  <!doctype html>
2  <html>
3    <head>
4      <title>Group Loop</title>
5      <script>
6
7      function saveNote(){
8 - 21          .....
22     }
23     function showAllNotes(){
24             var storedNotesString = localStorage.getItem("all_notes")
25             var allNotes = JSON.parse(storedNotesString)
26             if(allNotes != null){
27                     var noteDisplayer = document.getElementById("all_notes_display")
28                     noteDisplayer.innerHTML = null
29                     var numberOfNotes = allNotes.length
30                     for (var i = 0; i < numberOfNotes; i++) {
31                             var aNote = allNotes[i]
32                             noteDisplayer..innerHTML += "<hr><p>" + aNote + "</p>"
33                     }
34             }
35
36     }
37     </script>
38     <style>
39
40      </style>
41   </head>
42
43    <body>
44        <section id="note_entry_area">
45          <h1>Note It!</h1>
46          Description: <input id="description_input"></input>
47          <textarea id="note_editor" placeholder="Note it down. Don't forget."></textarea>
48          <button onclick="saveNote()">Save</button>
49          <button onclick="showAllNotes()">Show Notes</button>
50        </section>
51        <section id="all_notes_display"></section>
52    </body>
53 </html>
```

Code Sample - note_it.html

Details:

The new button is on line **49**. There you see the new function *showAllNotes* set to be run when the button is clicked. After you add the

button into your code, in your browser the page will look like Figure 1.

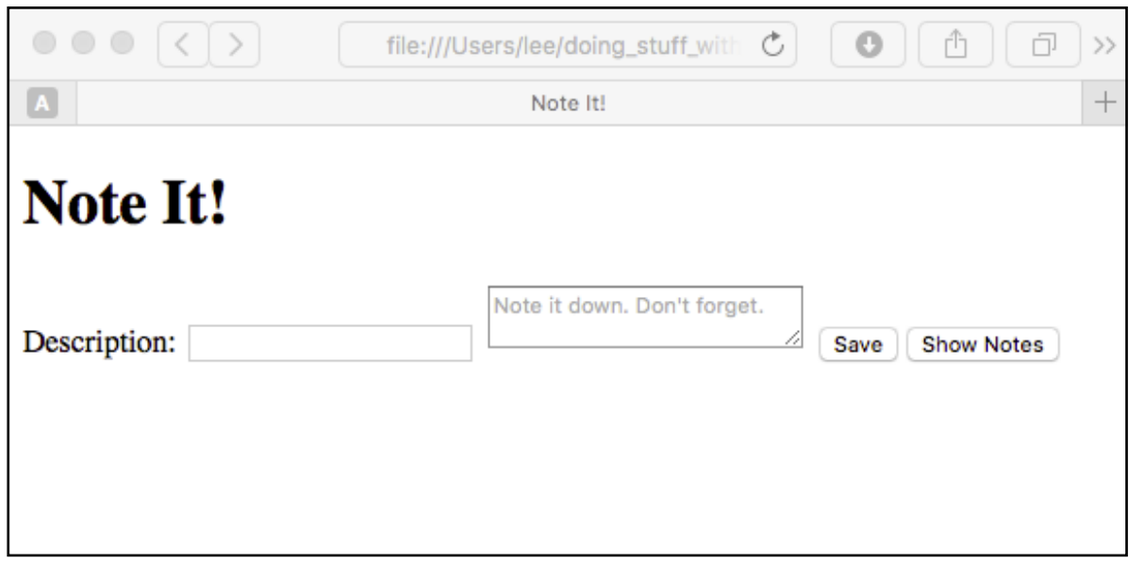


Figure 1. *The note_it.html page without any CSS.*

Nope...it's not pretty. Don't worry. You'll have a chance to make the user experience better after you get done with this last major part of the JavaScript.

Here we go. Most of what you are going to see here you've seen before; if, JSON, local storage, getElementById, a loop, innerHTML, and arrays. All you're going to do is use these things in a different situation. The principles all remain the same as the last time you used them.

Let's start with line **24**. It will get any stored messages and reads, "Hey localStorage, get the item that was stored with the key 'all_notes' and put what you find in the storedNotesString variable." That's identical to the last time you used the getItem function. Line 25 uses JSON.parse again. And it's exactly the same as when you used it in the *saveNote* function. In English it is, "Hey JSON. parse the string you find in the storedNotesString and put the result into the allNotes variable." So, if storedNotesString has all the notes in it, allNotes will end up being an array of the notes. And, just like in the *saveNote* function, if no notes have been stored then allNotes ends up holding that special 'nothing' called null.

Look carefully at the next line of JavaScript, line **26**. Compare it to line **16** in the *saveNote* function from before. I've copied both here for you so you don't have to go back and look.

```
16        if(allNotes == null){
.
.
.
26        if(allNotes != null){
```

Code Snippet 12

There is one small change that makes a major difference. See it? Line **16** reads,"If what is in the allNotes variable is null...." Line **26** says, "If what is in the allNotes variable is NOT null..." Ahhh...big difference. Maybe you remember from before that the ! character means 'not'.

Think about this. It is bad to use up any resource, air, water, electricity, food, etc., when it isn't needed. It's also bad to use up computer resources, electricity, CPU time, memory, storage, etc., when they aren't needed. So your code should try to display notes only if there are notes to display. That's what line **26** is there for. If allNotes is not equal to null, then there must be notes to display so dis-

play them. If there are no notes to display, lines 27 - 33 won't run. That saves electricity and other computer resources.

Line 27 says, "Hey document, go get the element who's id is 'all_notes_display' and put the section you find in the noteDisplayer variable." You've used document.getElementById several times now. This time, instead of changing what's between the beginning and ending tags, you'll store the section so you can use it later. "Why?" you ask? Good question. You are going to need to use it more than once. It uses less computer resources to look up a tag once and use it multiple times than it does to look it up over-and-over again. That's why you'll put the section in a variable. You're going to use the section more than once.

Line 28's a little strange. Let's take a look at it. It uses that special nothing indicator, null, in a different way.

```
28              noteDisplayer.innerHTML = null
```

Code Snippet 13

You remember from the many times you've used innerHTML before that whatever is on the right hand side of the equals character will end up between the noteDisplayer's starting and ending tags. "But null means 'nothing', Barney." Yep. You're right it does. That makes line 28 read like this. "Take nothing and put it between the beginning and ending tags of the noteDisplayer section." This is a clever little trick. If there were any notes between the noteDisplayer's starting and ending tags, they are gone! This is how you 'clear the screen' before you start showing the tags again. It would be a good idea if you played with this. I'd suggest removing line 28, and then trying to use the app. Add in a new note and then click the Show Notes button. Seriously...stop here and do this.

STOP HERE AND GO PLAY. PLEASE.

Did you see what went wrong???

You ended up with duplicate notes showing on the page. The same notes keep getting shown over-and-over again. Each time you click the 'Show Notes' button more duplicates get show up.

Now put line 28 back in again.

Double click the note_it.html file and then click the 'Show Notes' button. The duplicates are all gone! The key is line 28. Do some thinking here. Why does line 28 make the duplicates go away?

Figured it out??? OK. Then let's move on to the next line, 29.

Back when you were creating the group_loop.html page you learned about *for* loops. If you remember, the third part of a for loop checks to see if the loop should continue. Line 29 is finding

out what number should be used in the for loop in line **30** for this 'should the loop continue' check.

There are only a certain number of notes in the allNotes array. It would be bad to keep going around the for loop after all the notes in the array have been added to the page. Believe me. It would be REALLY bad. So line **29** in Code Snippet 1 finds out how many messages there are. In English it says, "Hey array in the allNotes variable, tell me how many notes you have in you and put that number in the numberOfNotes variable."

```
23    function showAllNotes(){
24            var storedNotesString = localStorage.getItem("all_notes")
25            var allNotes = JSON.parse(storedNotesString)
26            if(allNotes != null){
27                    var noteDisplayer = document.getElementById("all_notes_display")
28                    noteDisplayer.innerHTML = null
29                    var numberOfNotes = allNotes.length
30                    for (var i = 0; i < numberOfNotes; i++) {
31                            var aNote = allNotes[i]
32                            noteDisplayer..innerHTML += "<hr><p>" + aNote + "</p>"
33                    }
34            }
35
36    }
```

Code Snippet 1

Now that you know how many notes are in the allNotes array, you can create a *for* loop that will work well. The only difference between this for loop and the one you created for the group_loop.html page is the check. Line **30**'s check uses a variable and the for loop in the group_loop.html page used the number '7'. Back then, there were always seven things in the array. A note app that could only ever show the same seven notes wouldn't be any good. That's why line **30** includes the check that says, "keep going while the i variable is less than the number in the numberOfNotes variable." (Hint: If the less than part of the quote is confusing you, go back to the beginning of this section. It is explained there.)

Lines **31** and **32** are what is going to happen each time your code goes around the loop. Remember, the i variable starts at zero and goes up 1 each time your code goes around the loop. So it will change like this; 0,1,2,3,4,5... and will continue until is reaches the number in the numberOfNotes variable.

31 reads,"Hey allNotes array, when i is zero get me the zeroth note from the array. When i is 1 get me the oneth note, etc., etc. In other words, give me the ith note and put it in the aNote variable." then line **32** appends the note to whatever is already in between noteDisplayer's starting and ending tags. You've seen code much like line **32** before so I won't re-explain it here but there is one new thing.

The horizontal rule tag, <hr>, adds a line all the way across the page. In what you're doing here, it will put a line across the page before each note is shown. (Notice the hr tag doesn't have a closing tag. Don't try to insert one.)

So that's the code so far. Stop. Ponder what you've seen. Truly strive to understand it. Don't just go on thinking, "If I keep on, sooner or later I'll get it." It won't happen. If you don't understand, stop, play with the code you've written, ponder, and think. You should also talk with someone about what you are trying to learn. They don't have to be an expert in JavaScript. They don't even need to be a programmer or know anything about programming. If you have a teacher, you should also ask them questions. Good, specific questions about what you've read and don't understand.

STOP HERE. PONDER AND THINK. PLEASE.

If you take a look at what your page looks like, Figure 2, it's pretty ugly. Your user won't like it. Let's fix it up.

Making It Better

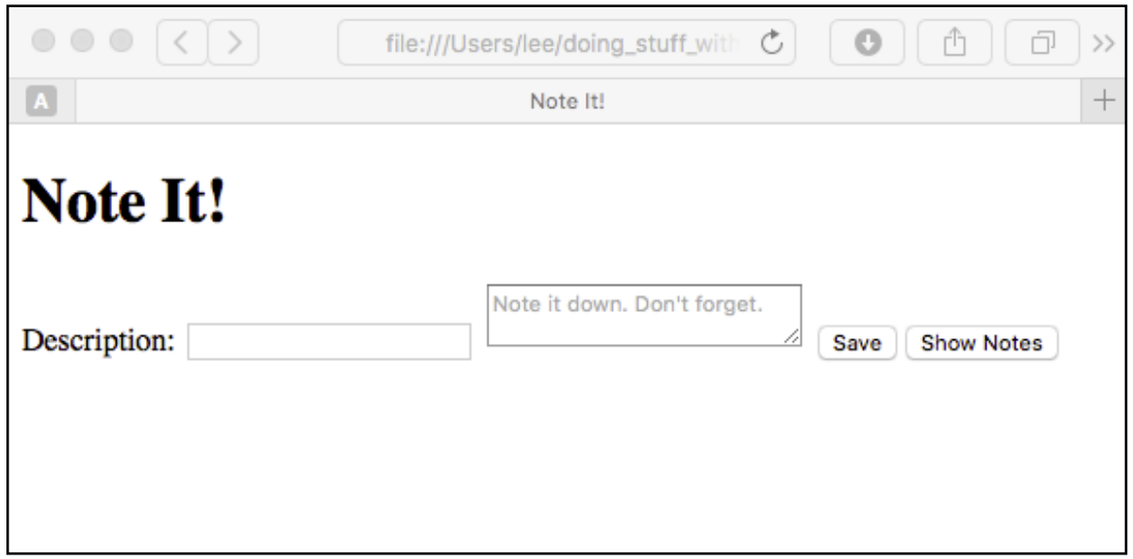


Figure 2. *The note_it.html page without any CSS.*

The first thing that really bothers me is the 'Show Notes' button. One thing is certain. Users hate clicking when they don't need to. Why should you make your user click Save and then have to click 'Show Notes' to see if the note was saved? Why can't the page just update the list of notes each time your user clicks Save? Well...it can. You're going to use a function trick that you didn't learn about in the Functions and Events section. You are going to remove the ShowNotes button and have your *saveNote* function run, in techie speak - call, your *showAllNotes* function. Kind of like the embedded ifs from the Section 4, but instead of embedded ifs there'll be embedded functions. After you make this change, your code will look like this.

```
 1 <!doctype html>
 2 <html>
 3   <head>
 4     <title>Group Loop</title>
 5     <script>
 6
 7     function saveNote(){
 8               var currentDateAndTime = new Date()
 9               var aNoteDescription = document.getElementById(“description_input”).value
10               var aNoteText = document.getElementById(“note_editor”).value
11               var aCompleteNote = currentDateAndTime.toLocaleString()
                                                          + "--" + aNoteDescription
12               aCompleteNote += “<p>” + aNoteText + “</p>”
13
14               var storedNotesString = localStorage.getItem(“all_notes”)
15               var allNotes = JSON.parse(storedNotesString)
16               if(allNotes == null){
17                         allNotes = []
18               }
19               allNotes.push(aCompleteNote)
20               var allNotesString = JSON.stringify(allNotes)
21               localStorage.setItem(“all_notes”, allNotesString)
22               showAllNotes()
23     }
24     function showAllNotes(){
25 - 36          .....
37     }
38     </script>
39     <style>
40
41      </style>
42   </head>
43
44    <body>
45        <section id="note_entry_area">
46          <h1>Note It!</h1>
47          Description: <input id="description_input"></input>
48          <textarea id="note_editor" placeholder="Note it down. Don't forget."></textarea>
49          <button onclick="saveNote()">Save</button>
50        </section>
51        <section id="all_notes_display"></section>
52    </body>
53 </html>
```

Code Sample - note_it.html

“So one of my functions can use another one of my functions?!” You bet. It is one of the great things about functions. If you make this change, double click your note_it.html file, and then add another note, all of the stored notes you’ve saved before show up.

“That’s fine,” you say, “but my user won’t want to have to click the Save button just to see what they’ve already saved. You said they hate clicking if they don’t have to.” You’re absolutely right. They won’t want to do that extra, worthless click. They’ll want to see all of their saved messages as soon as they double click the note_it.html file. That would be a MUCH better experience. Let’s make that happen in your code.

Another Type Of Event

To understand this next change, you need to learn about a new type of event. You are already familiar with the click event. It’s the event your button is listening for. Remember *onclick*? The function you assign to *onclick* for the button is the function that runs when the button is clicked. Right?

Well the new event is named ‘load’. Guess what? There is a way to assign a function to execute when the load event happens. If the click/onclick pattern holds, and it does, then since there is a load event there should be something called *onload*. There is but it isn’t part of button. It’s part of the body tag. Line **44** in Code Snippet 2 shows you how to add *onload* to your page’s body tag and run showAllNotes when the load event happens.

```
<body onload="showAllNotes()">
    <section id="note_entry_area">
      <h1>Note It!</h1>
      Description: <input id="description_input"></input>
      <textarea id="note_editor" placeholder="Note it down. Don't forget."></textarea>
      <button onclick="saveNote()">Save</button>
    </section>
    <section id="all_notes_display"></section>
</body>
```

Code Snippet 2

"But when does the load event happen?" you ask. Another excellent question! The load event happens when all of your page's JavaScript, CSS, and HTML has been loaded and is ready to go, but before anything is shown to the user. So the load event happens right before the user sees anything.

That time between when everything is ready to go and before the user is shown anything is a very handy time to make changes to your page. That way the user won't see the changes happen. It would be bad to show your user the note_it.html page and then show them all the saved notes. That would irritate them. Instead, by using *onload*, you can update the page with all the notes BEFORE your user sees anything on the page.

Figure 3 shows what happened after I made this change and double clicked this version of note_it.html. I had previously stored a note who's description was 'Stuff' and the note was 'I don't want to forget'.

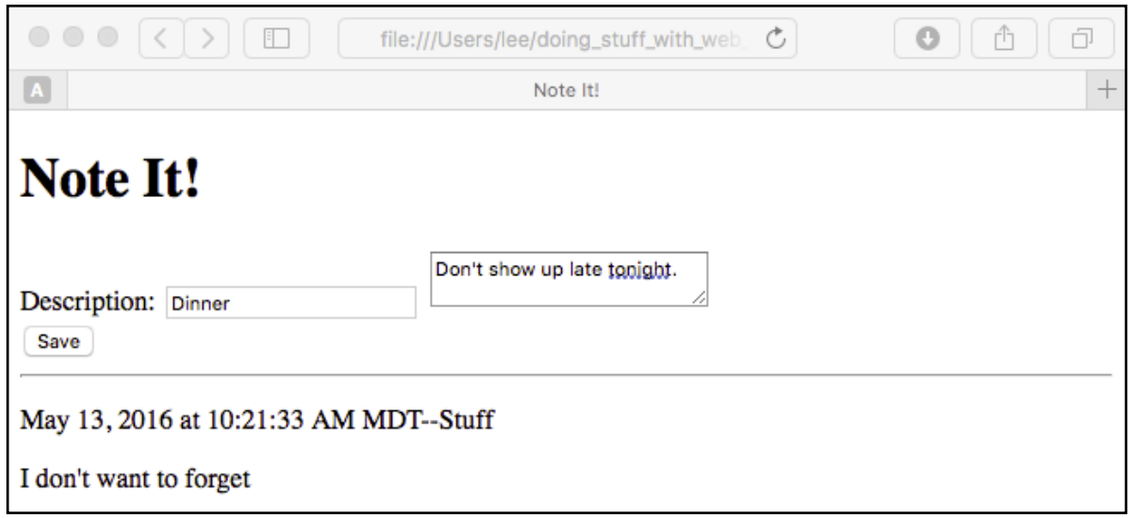


Figure 3. *The note_it.html page after adding onload to the body tag.*

All right...this is getting MUCH better. But there are still some things that are bugging me. Take a look at Figure 4.

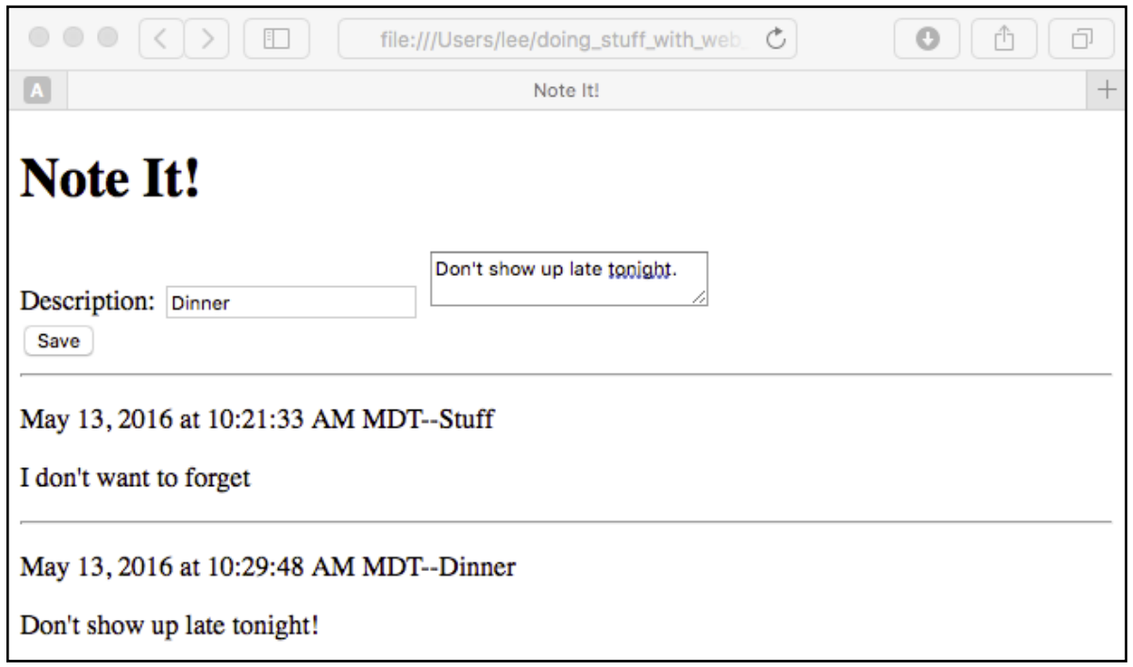


Figure 4. *The note_it.html page after adding another note.*

After the Save button was clicked, my new reminder to be home for dinner on time was saved and shown to me. That is working right. But now, if I need to add another note, the old description and note are in the way. I'm going to have to delete them from the input box and the text field.

Now, I'll grant you that that isn't too hard. But it is irritating because I know the page could have cleared them out for me. It doesn't take much to make this happen and it will reinforce something you just learned. Code Snippet 3 shows you some code that clears out the description and the note. Take a look at lines **23** and **24**. They should look a little familiar. Based on what you just learned about getting rid of the duplicate messages, can you figure out what these two new lines of JavaScript are doing? Spend some time with this. Don't rush.

```
7   function saveNote(){
8           var currentDateAndTime = new Date()
9           var aNoteDescription = document.getElementById("description_input").value
10          var aNoteText = document.getElementById("note_editor").value
11          var aCompleteNote = currentDateAndTime.toLocaleString()
                                                    + "--" + aNoteDescription
12          aCompleteNote += "<p>" + aNoteText + "</p>"
13
14          var storedNotesString = localStorage.getItem("all_notes")
15          var allNotes = JSON.parse(storedNotesString)
16          if(allNotes == null){
17                  allNotes = []
18          }
19          allNotes.push(aCompleteNote)
20          var allNotesString = JSON.stringify(allNotes)
21          localStorage.setItem("all_notes", allNotesString)
22          showAllNotes()
23          document.getElementById("description_input").value = null
24          document.getElementById("note_editor").value = null
25  }
```

Code Snippet 3

Have you got it figured out?

Remember how you set what would be between the all_notes_display section's starting and ending tags to be nothing?

Line **23** does something similar but to the description_input box instead of a section. It says, "Hey document, go get the element who's id is 'description_input' and set it's value, what the user typed in, to be nothing (null)." Got it?? It makes it as if the user had typed in nothing! Line **24** does the same thing for the note_editor text area and the end result looks like Figure 5 when a new note is entered.

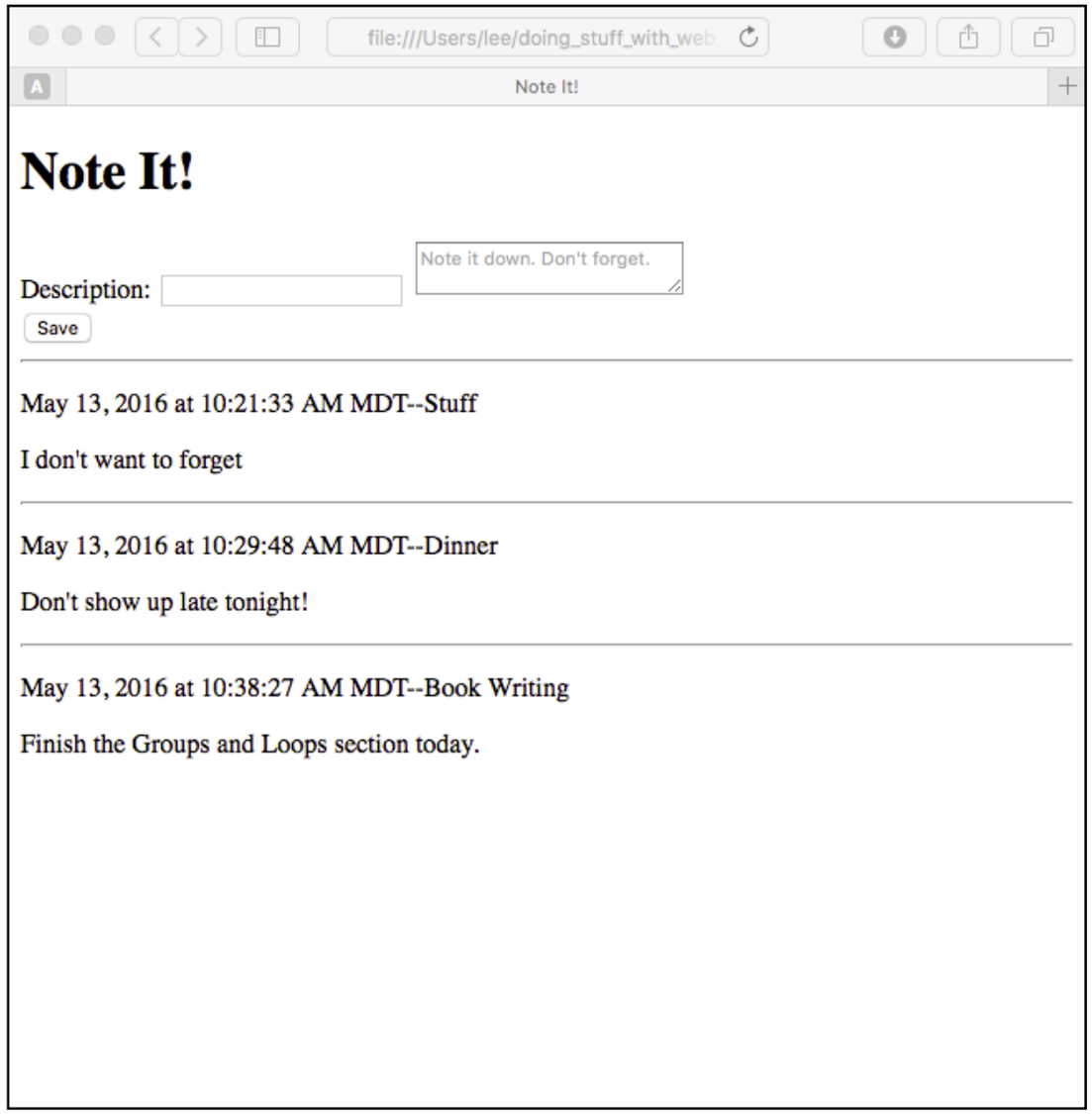


Figure 5. *The note_it.html page clearing input when a note is saved.*

"But wait a minute Barney," you say. "You said good code shouldn't look up something twice; that would waste resources. Didn't you just break your own rule?" You are right. I did break my own rule. I did it on purpose. I could have made some more minor changes and told you EXACTLY how to do those changes. I didn't because you need to figure out what changes to make to saveNote so description_input and note_editor are only looked up once instead of twice. This will be good practice for you. It will help you learn to write your own code instead of just typing in what you are told. After all, programming isn't about being told what to type, it's figuring out what should be typed to create an application. It is a creative process and I didn't want to deprive you of this opportunity.

You make the change. Do it now. I'm not going to show you how to do it. You know and understand enough to do this if you have been understanding stuff as you've gone through this book. Take a chance. Strike out on your own.

STOP HERE. GO FIX THE FUNCTION.

Now that you've fixed that mistake, what techies call 'a bug', let's move on. There are still some things that are bugging me that you could fix with JavaScript, but I think it's time to move on and make this app look nicer. Let's do some CSS.

What you're going to add next is similar to the CSS you used in the even_or_odd.html page. That should make it easier to see the new stuff.

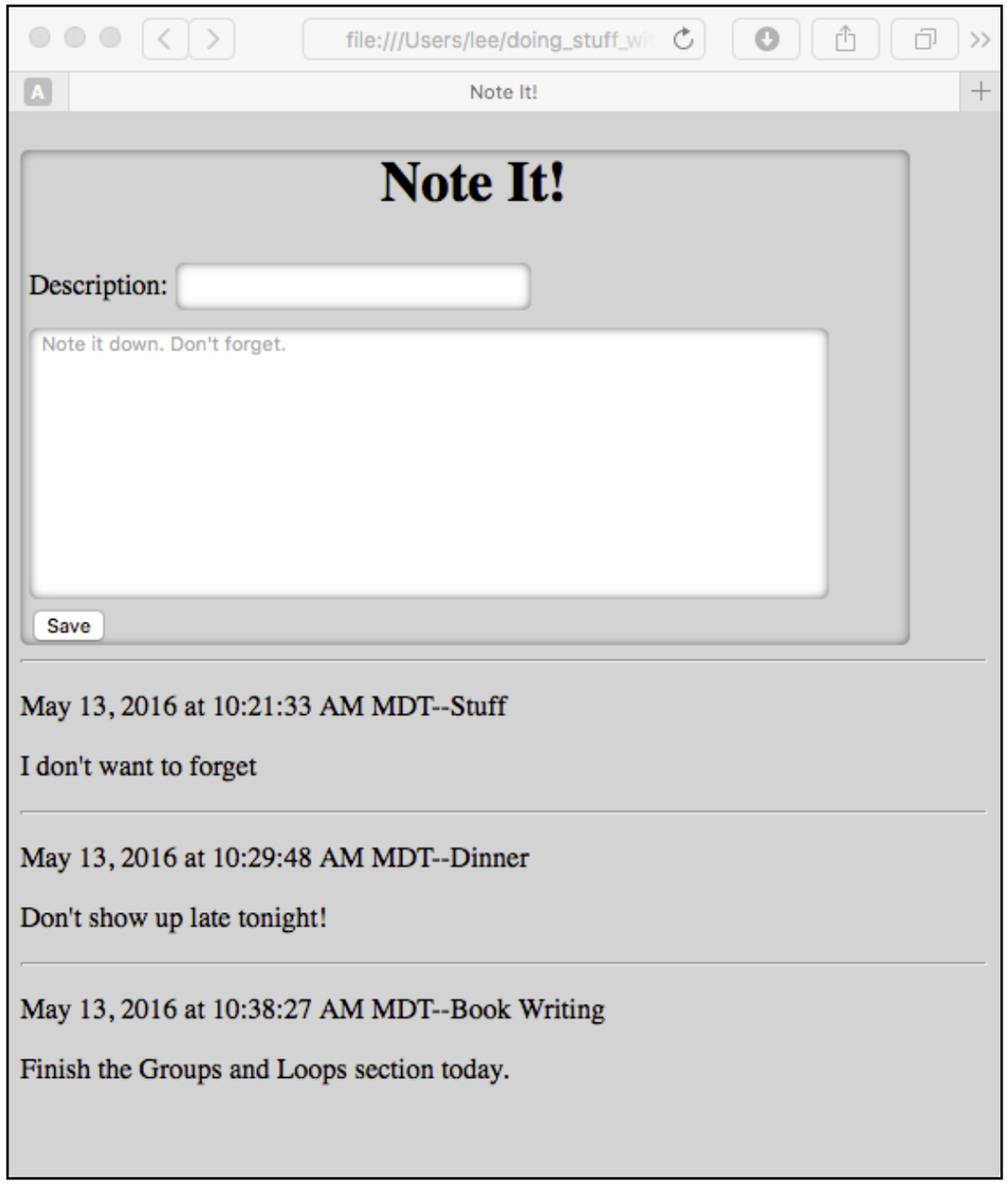


Figure 6. *The note_it.html page with all the CSS added.*

The page in Figure 6 does look better than Figure 5. It isn't perfect. Feel free to play with changes after you understand the new stuff you're going to see. That way the note_it.html app will reflect your taste.

OK. Here we go.

Shifting the background to grey and making the text input box look nice are easy. You've seen them before. Let's start there.

```
<style>
        body{
            background-color: LightGrey;
        }
        input{
            border: none;
            border-radius: 5px;
            box-shadow: 0px 0px 5px #666666 inset;
            height: 25px;
            width: 200px;
            text-indent: 5px;
        }
        input:focus{
            outline:none;
        }
</style>
```

Code Snippet 4

No changes since you did this to the other page, so let's add in something new.

If you remember, there is a section you gave the id of 'note_entry_area'. Now your going to give that section, and only that section, some CSS. Code Snippet 5 shows how to pull this off.

```
<style>
        body{
            background-color: LightGrey;
        }
        input{
            border: none;
            border-radius: 5px;
            box-shadow: 0px 0px 5px #666666 inset;
            height: 25px;
            width: 200px;
            text-indent: 5px;
        }
        input:focus{
            outline:none;
        }
        #note_entry_area{
            width: 500px;
            padding-left: 5px;
            border:none;
            border-radius:5px;
            box-shadow:0px 0px 5px #666666 inset;
            text-indent:5px;
        }
</style>
```

Code Snippet 5

When you want to assign some CSS to an element of your page you can use the id of the element to identify it specifically. You need to use the # character in front of the element's id so your computer knows you want it to only change the element with that specific id. The CSS stuff you just added between the style tags reads like this. "Hey computer, go find an HTML tag that has the id 'note_entry_area' and apply the following CSS to what you find."

So, if you want to apply CSS to all sections, you would have used 'section'. Since you only wanted to apply the CSS to the note_entry_area section and NOT the 'all_notes_disply' section, you used '#note_entry_display' instead. The CSS you're applying to the section is almost the same as that for the input box. Again, you've

seen the effects of adding this before so I won't go over it line-by-line except for one thing.

On the second line of the CSS for the note_entry_section, there is a new thing. It reads, "Hey computer, add 5 pixels of padding to the inside of the section on the left. Don't allow anything to be in those 5 pixels."

You can use padding on any or all of the insides of sections, bodies, or other tags. It's a way to make some space so your text and other things in the page don't bump right up against the edge of the section, body, or other tags. It's handy. You'll use it a lot.

The changes you've made are making a big difference. Now the page looks like Figure 7.

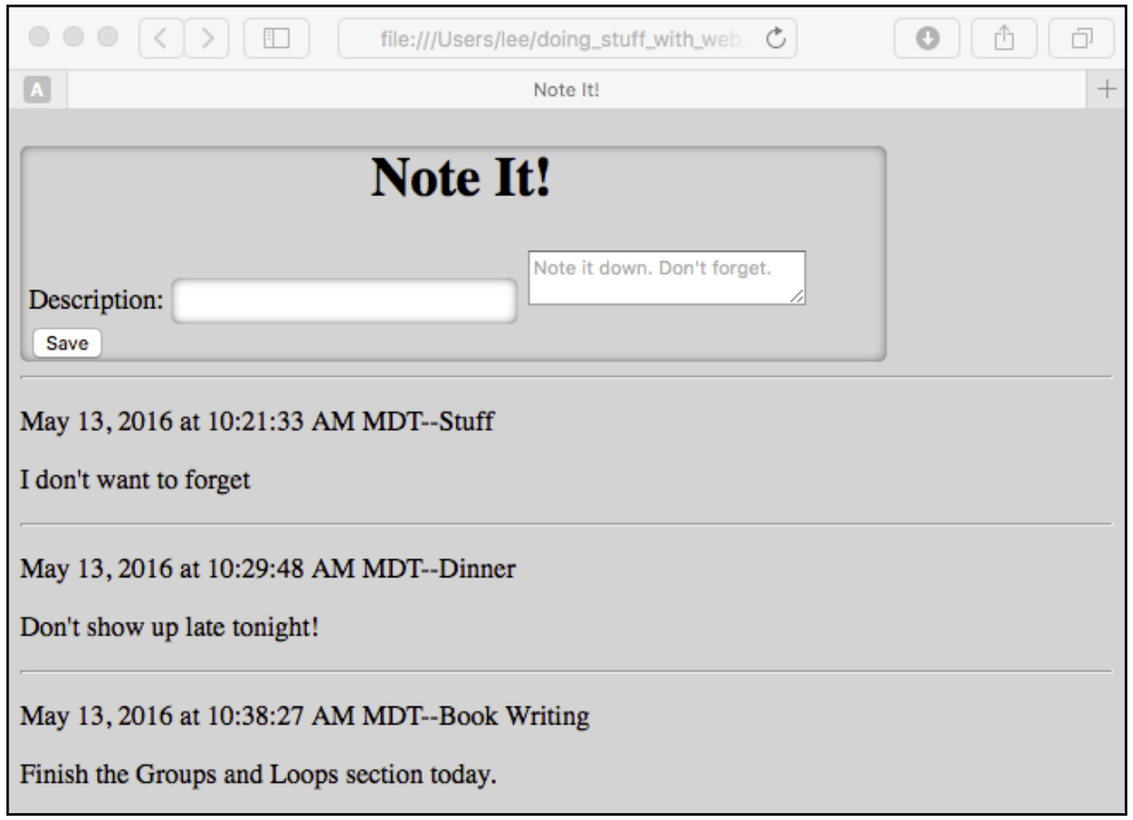


Figure 7. *The note_it.html page with some CSS added.*

That textarea is still awful. It is too small. It needs to be larger so your user can see what they're typing. It would be nice to change the border so it matches the borders for the section and text input box too. Do that next. Since there's only one textarea you can apply the CSS to all text areas without hurting anything.

```
<style>
        body{
            background-color: LightGrey;
        }
        input{
            border: none;
            border-radius: 5px;
            box-shadow: 0px 0px 5px #666666 inset;
            height: 25px;
            width: 200px;
            text-indent: 5px;
        }
        input:focus{
            outline:none;
        }
        #note_entry_area{
            width: 500px;
            padding-left: 5px;
            border:none;
            border-radius:5px;
            box-shadow:0px 0px 5px #666666 inset;
            text-indent:5px;
        }
        textarea{
            margin-top: 10px;
            border:none;
            border-radius:5px;
            box-shadow:0px 0px 5px #666666 inset;
            width:90%;
            height: 150px;
            text-indent:5px;
        }
</style>
```

Code Snippet 6

The height and width are set to be different from what they are for the input box, but they should be. You wouldn't want them to be the same. That would be way to small.

After you add in the changes at the end of Code Snippet 6, your page looks like Figure 8.

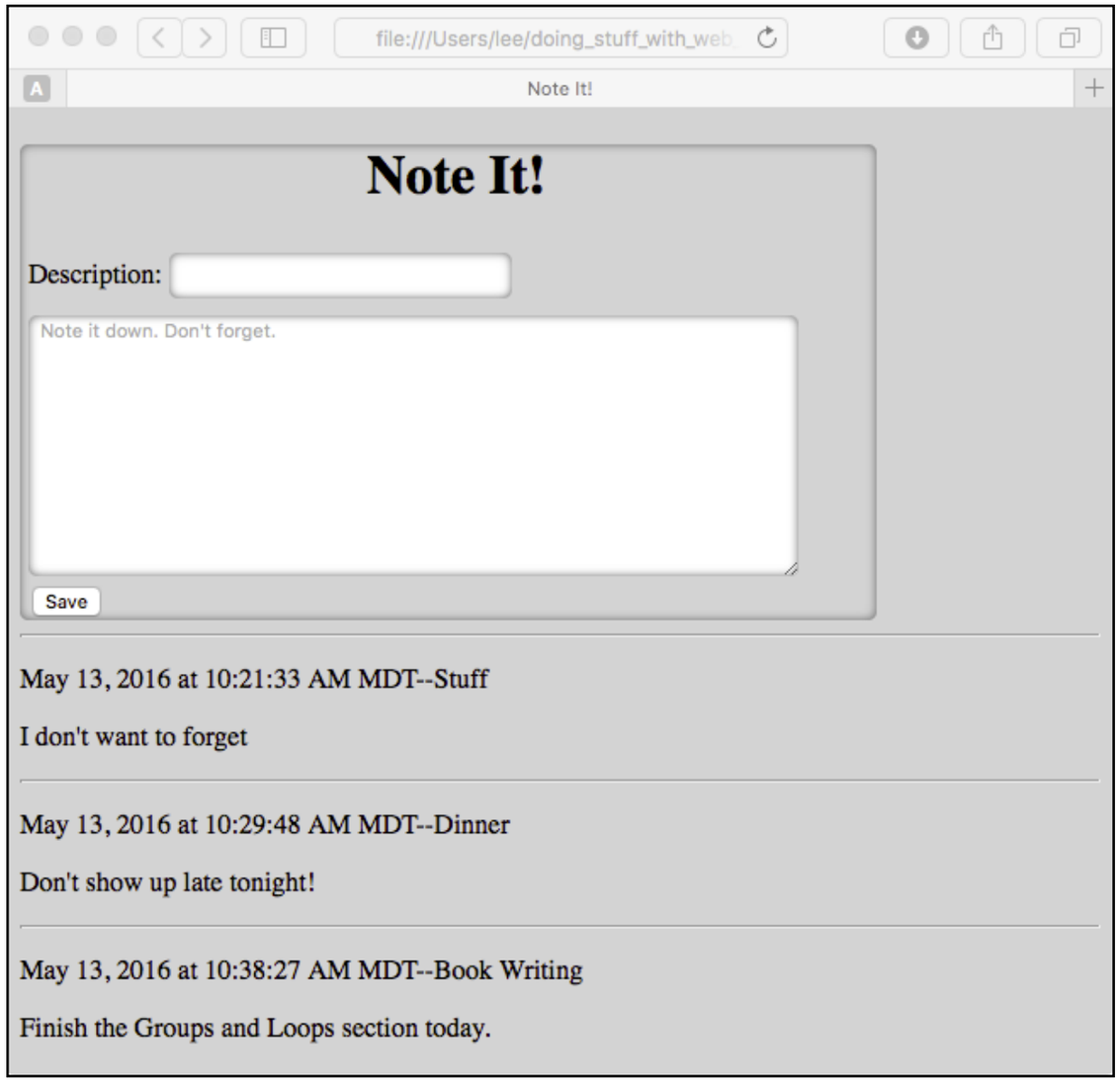


Figure 8. *The note_it.html page with the text area resized.*

There is just one more thing. In the bottom-right corner of the text area, you can see a resizing indicator. If you let the user resize the text area it's going to mess up the work you've done. But if you turn off your user's ability to resize the text area they won't be able to see really long notes they enter. This is a problem. Let's fix it.

Not only is it possible to turn off the resizing of text areas using CSS, but you can also turn on scrolling! You always wondered how they got parts of the pages you've used to scroll, haven't you. Now that piece of magic is going to go away.

You'd think these two things would be hard. After all, scrolling involves a lot of stuff. You've got to have scroll bars, you have to make the text move when your user scrolls, and all kinds of stuff. Guess what? In CSS these things are easy. So easy that to make both changes requires only two lines of CSS.

```
<style>
        body{
            background-color: LightGrey;
        }
        input{
            border: none;
            border-radius: 5px;
            box-shadow: 0px 0px 5px #666666 inset;
            height: 25px;
            width: 200px;
            text-indent: 5px;
        }
        input:focus{
            outline:none;
        }
        #note_entry_area{
            width: 500px;
            padding-left: 5px;
            border:none;
            border-radius:5px;
            box-shadow:0px 0px 5px #666666 inset;
            text-indent:5px;
        }
        textarea{
            margin-top: 10px;
            border:none;
            border-radius:5px;
            box-shadow:0px 0px 5px #666666 inset;
            width:90%;
            height: 150px;
            text-indent:5px;
            resize:none;
            overflow:scroll;
        }
</style>
```

Code Snippet 7

Code Snippet 7 has two new lines of CSS. You can see them at the end of the CSS for all the text areas. The first line is 'resize:none'. It means, "Hey computer, turn off resizing for any textarea you find and take away the resizing indicator." That's it. No more resizing to mess up the work you've done!

The last line of the textareas' CSS, overflow:scroll, is all that's required to get stuff to scroll inside a section, textarea, etc. It says, "Hey computer, if the stuff inside the textarea is too big to fit inside it, add scrollbars and let the stuff inside scroll around." That's it!

Now, there are other options for resize and overflow but none and scroll are all you need right now. Feel free to try out some of the other options. You can find out what they are by googling them.

Representing Things

Lorem Ipsum

1. How can JavaScript represent things we see in the real world and complex ideas?
2. What is an associative array?
3. How is an associative array made and used?

The world around us is made up of things. We make up groups of things all the time. This thing fits in the bird, goose, and snow goose groups and many others. It could also fit in a 'things with white coloration' group.

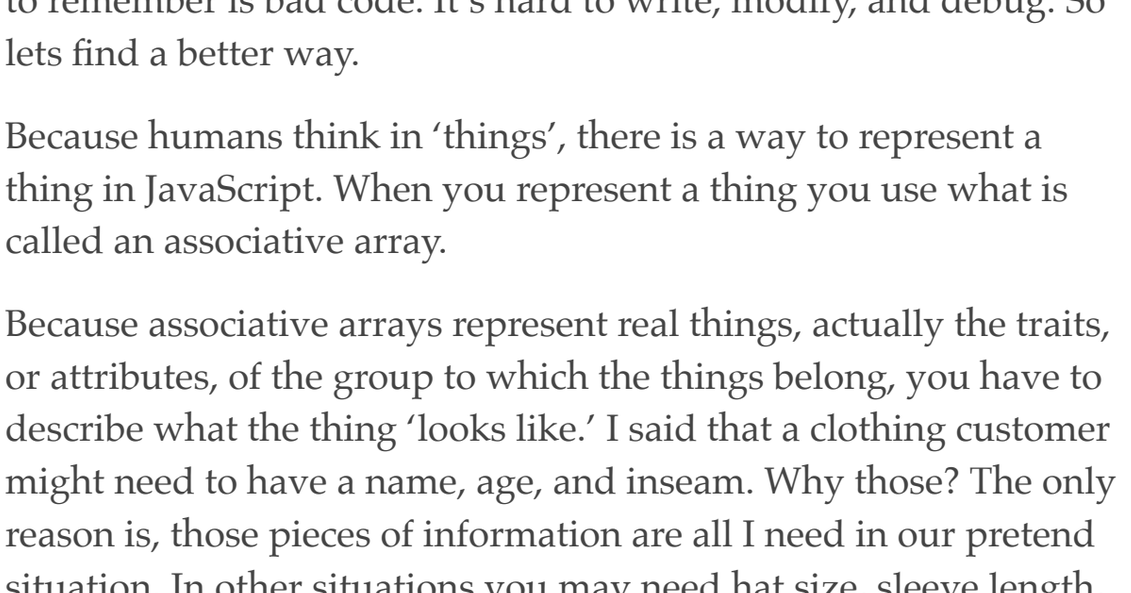
Arrays are nice, they let you keep stuff in order, let you find stuff quickly using an index number, and can be easy to use, but how in the world can you represent something like a customer using arrays? A clothing customer might have a name, age, and inseam. That means each customer would need to consist of a string, int, and double. You could put those three things in an array, but then you'd have to make sure to remember what order they were in. That sounds easy right now, but what if there were 15 instead of 3 things for each customer? Now it's really hard to remember what order they are in. Stuff that's hard

to remember is bad code. It's hard to write, modify, and debug. So lets find a better way.

Because humans think in 'things', there is a way to represent a thing in JavaScript. When you represent a thing you use what is called an associative array.

Because associative arrays represent real things, actually the traits, or attributes, of the group to which the things belong, you have to describe what the thing 'looks like.' I said that a clothing customer might need to have a name, age, and inseam. Why those? The only reason is, those pieces of information are all I need in our pretend situation. In other situations you may need hat size, sleeve length, or any number of other measurements. For our example let's keep it simple.

The pattern for describing something's group using an associative array is always the same. A variable is used to hold the associative array.

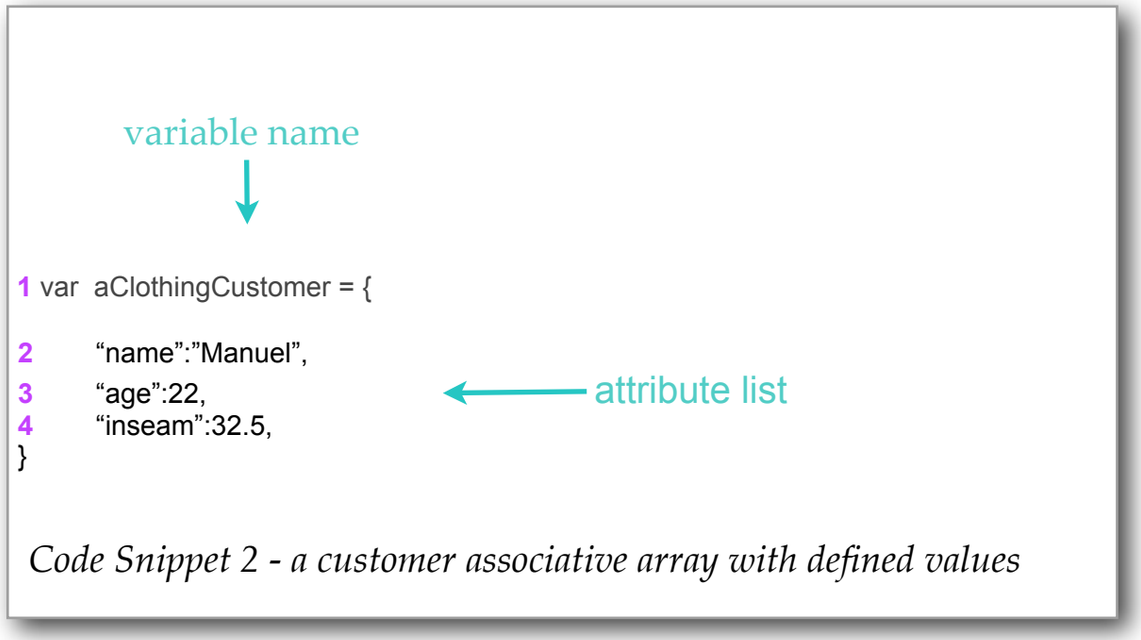


Code Snippet 1 - describing a group using an associative array

This is followed by using the = character to assign the associative array to the variable. In other words, the associative array will be put in the variable.

You use the { and } characters to create an associative array. Between them, add a bunch of attribute names and each attribute's associated value. This is how you describe any kind of thing.

So lets stop talking about patterns and how to do this in general. Let's look at an example of a customer. The customer's name is Manuel. He is 22, and has an inseam of 32.5 inches.



Code Snippet 2 - a customer associative array with defined values

Details:

Line **1** starts you off. It has the name of the variable, *aClothingCustomer*, and says,"Create a variable called aClothingCustomer and put into the variable an associative array with the properties...." The { character on the right hand side of the = character tells the computer that what comes next is an associative array.

Line 2 says"Give the associative array an attribute called 'name' and set the value of the attribute to 'Manuel'. Line 3 does the same thing for the age attribute. "Also give the associative array an attribute called 'age' and set its value to 22." Line 4 is the last attribute. "Finally, give the associative array an attributed called 'inseam' and set its value to be 32.5." You know inseam is the last attribute because the } character comes next. It tells the computer that the associative array is now complete.

One more thing. Techies call the name of an associative array's attribute its 'key'. So in this case 'name' is a key and its associated value is 'Manuel'. Got it? Let's try another. There is an 'age' key and its associated value is 22. One more time. There is an 'inseam' key and its associated value is 32.5.

Keys and values. That's what make up associative arrays. You use them to describe complicated things, like customers, that have properties. Got it now? Great! Let's go on.

You're going to make an app that lets the user enter names, ages, and inseams for customers, stores the customers, and displays all the customers that have been stored. In a lot of ways, this app is nearly identical to the one from Section 5, Groups and Loops. There will be some small changes to what you learned there but don't let the changes mess you up. Keep focused on what you already know. Don't loose the forest, your in the trees. :)

So here is what the app is going to look like.

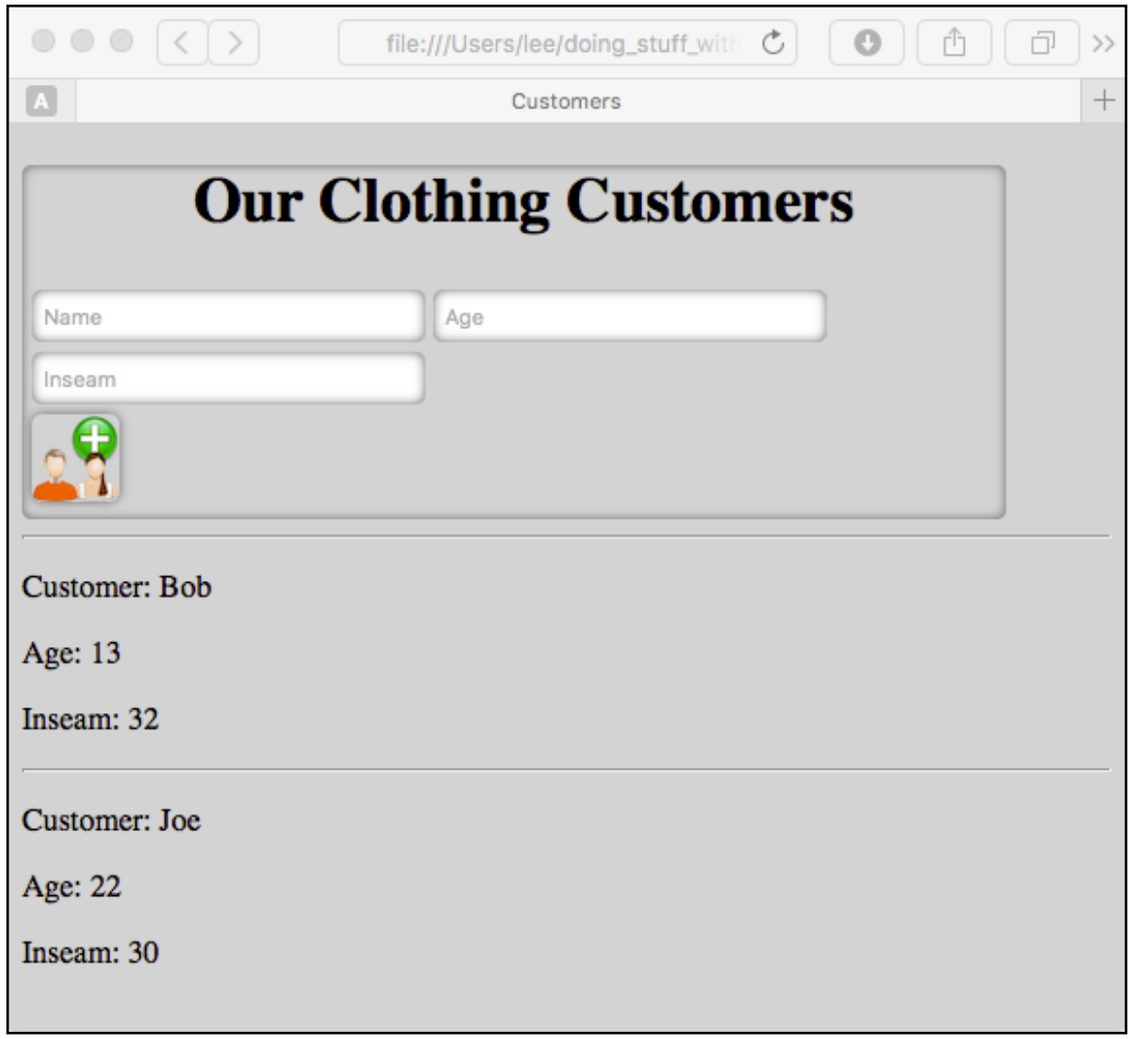


Figure 1. *The completed application with two customers added.*

You'll see some strong similarities between this one and the one from Section 5. There is a part of the page used to enter customer information. It's going to be a section, right? That section will need to have an h1 header tag and 3 input tags, one for each of the pieces of data. "But wait a minute Barney. Why'd you put Name, Age, and Inseam in the input tags? Won't the user have to clear them out before they can type?!" Good catch! Glad you brought it up.

Like the text area used in the last section's final example, input tags can also have place holders. That means when your user starts typing a name, age, or inseam the place holder will disappear. Nice. Now your app doesn't have to have those ugly pieces of text out in front of its input boxes. That is the old way to tell your user what to type. Oh...and by the way...it's the way they did it in the 1990's. Us techies have learned a lot about programming since then. Go with the new way.

"But there isn't a Button!" you say. Oh...another good catch. You're right. The app could have a button that said something like "Add a Customer". It could, but that's so old school. Instead, this app has an icon your user can click to add a customer. it's a much more modern approach.

So those are the similarities and differences that you can see. You'll need to make some minor adjustments to the HTML, CSS, and JavaScript from Section 5's note_it.html application to make this happen. But don't worry. It isn't a lot of changes and I'll step you through them like always.

If you're ready, we'll start with the HTML.

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>Group Loop</title>
5     <script>
6     </script>
7
8     <style>
9
10     </style>
11  </head>
12
13   <body onload=”showAllCustomers()”>
14       <section id="customer_entry_area">
15         <h1>Our Clothing Customers</h1>
16         <input id="name_input" placeholder="Name”></input>
17         <input id="age_input" placeholder="Age”></input>
18         <input id="inseam_input" placeholder="Inseam”></input>
19         <br>
20         <img src=”add_customer.png” onclick="addAndSaveCustomer()"></img>
21       </section>
22       <section id="all_customers_display"></section>
23   </body>
24 </html>
```

Code Sample - customer.html

So there it is. It's almost the same as what you saw before. Lines 16 - 18 are the three input tags and their place holders. The onload function for the body has a different name but is almost the same thing you saw for the Note It! app. The section on line 22 is the same except the id has changed so it will describe the section's new contents. All-in-all, the biggest change is line 20, the new icon the user can click.

One nice thing about HTML is you can add onclick to anything you want. It doesn't have to be buttons. Buttons themselves can be too ugly for designers. Instead you can use an image of almost any type, but first you have to learn about the image tag.

The image tag is shorthanded to <img></img> for its starting and tending tags. “That’s not shocking,” you say. “I’ve already figured that out. What’s that src thing?” It is where you put the name of the image file, its source. Get it?? source - src?? Good.

There is just one thing to be careful of. Officially, the src you put in your img tag has to be the path to the image file. The easiest, but not always the best, thing to do is put the image in the same directory on you computer as your html file. For this app, I’d suggest putting the add_customer.png file in the same spot as your customer.html file. Then all you’ll need to do is set the src to ‘add_customer.png’, the name of the file. If you double click your customer.html file and can’t see the image, its because you don’t have the html and png files in the same directory. Move the image file around until you get it in the right spot. It will already be in the right spot in the gitHub zip file download for this book.

So far, when you double click the customer.html file it looks like Figure 2.

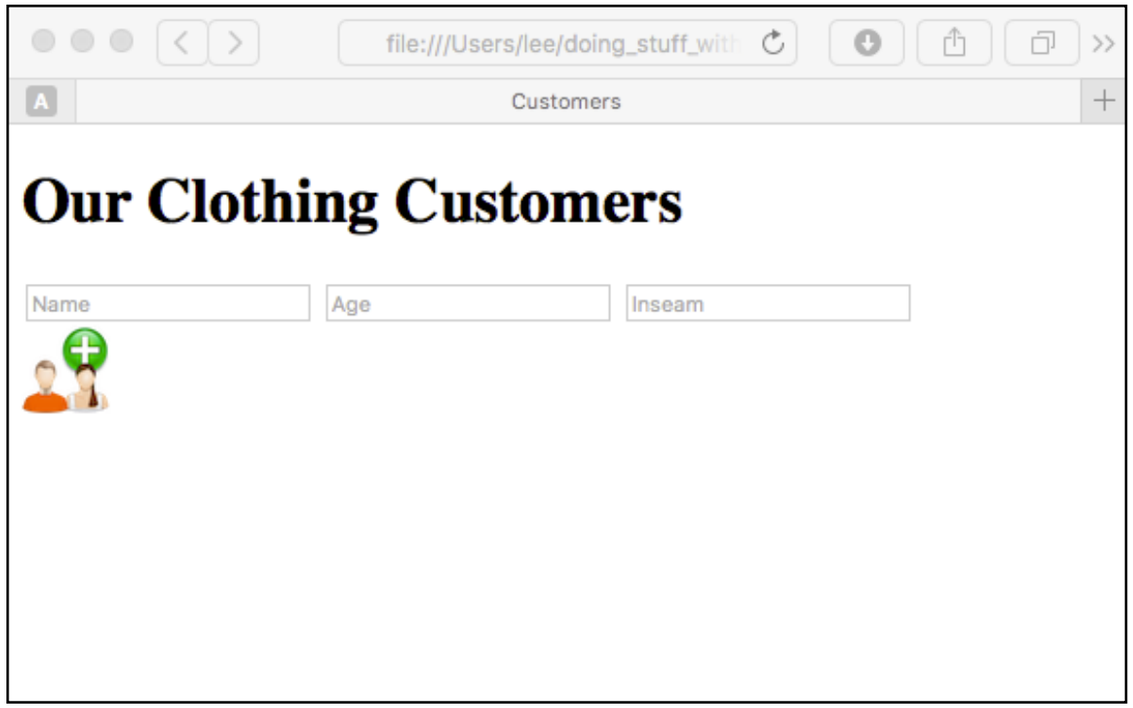


Figure 2. *The customer.html file with only the HTML in place.*

The JavaScript for this app is almost the same as the code in the note_it.html app. Let’s go over the similarities and differences.

The customer.html app has two functions, the first adds and saves new customers, the second displays all of the customers that have already been stored.

The flow of the first function goes like this;

1. get the customer stuff to store,

2. get the stored customers,

    a. If there aren’t any stored customers, create an empty array to put customers in.

b. If there are stored customers, convert the stored customer string back into an array.

3. add the new customer to the array,

4. store all the customers,

5. update what the user can see with all the new customers, and

6. clear out the input boxes.

Let's look at the code for the first function, addAndSaveCustomer.

```
6   function addAndSaveCustomer(){
7           var aName = document.getElementById("name_input").value
8           var anAge = document.getElementById("age_input").value * 1
9           var anInseam = document.getElementById("inseam_input").value * 1
10          var aClothingCustomer = {
11                  "name":aName,
12                  "age":anAge,
13                  "inseam":anInseam
14          }
15          var allCustomers = null
16          var storedCustomersString = localStorage["all_customers"]
17          if(storedCustomersString == null){
18                  allCustomers = []
19          }
20          else{
21                  allCustomers = JSON.parse(storedCustomersString)
22          }
34          allCustomers.push(aClothingCustomer)
24          var allCustomersString = JSON.stringify(allCustomers)
25          localStorage["all_customers"] = allCustomersString
26          showAllCustomers()
27
28          document.getElementById("name_input").value = null
28          document.getElementById("age_input").value  = null
30          document.getElementById("inseam_input").value = null
31  }
```

Code Snippet 3

Details:

Lines 7, 8, and 9 you've seen before, but with a minor change. You'll want the stored ages and inseams to be numbers instead of text. That will let you see patterns in your customers as you try to grow your business. That's the plan, right? Grow the business? If it isn't then there is no sense in creating the app.

If you look at the ends of lines 8 and 9 you'll see ' * 1'. If you multiply the text the user enters by the number 1, JavaScript will convert the text to a number! That's nice. I've seen a lot of other languages where it was a lot harder than that. Let's just do a happy dance and move on.

Lines 10 - 14 is where you put together an associative array that represents a clothing customer. In Code Snippet 2 the values associated with the keys were typed directly into the code, in techie speak they would be 'hard coded'. Hard coding is fine if every customer is exactly the same. That's never going to be true in a real business. Instead, these lines of JavaScript associate the keys with what was put in the aName, anAge, and anInseam variables. Pause and think about that for a minute.

In English, Lines 10 - 14 read, "Create an associative array called aClothingCustomer with a key called 'name' who's associated value is what is in the aName variable. Add another key called 'age' who's associated value is what is in the anAge variable. Last of all, add a final key called 'inseam' who's associated value is what is in the anInseam variable."

Read this over a few times. make sure you understand it. Work it out in your head, write it out step-by-step on paper, draw a picture, write some other playground code that does something simi-

lar, or whatever you need to do to get to where you understand what's going on.

Ready? Then let's go.

Lines 15 - 22 are the ones where your JavaScript gets the stored customers (Step 2 of the flow). It looks different than the JavaScript where you got stuff from localStorage before but the basics are the same. If customers have already been stored, then you want to add the new customer to that array. But in this app, if there aren't any other customers, you should create a new array and add the customer to that.

That new array? You first see it on line 15. That line says, "Create a variable called allCustomers and set it, temporarily, to be null (nothing)." Don't worry. The if-else statement, lines 17 - 22, will take care of making sure allCustomers ends up with the right stuff in it.

Line 16 is a different way you can get stuff out of localStorage. Guess what??...LocalStorage is an associative array!!! That means you don't need to use a method to to get stuff back out. Let's read line 16. "Hey localStorage, get me the string associated with the 'all_customers' key." No more using 'getItem'!

If nothing is associated with the 'all_customers' key, then null ends up in the storedCustomersString. That sets you up for lines 17 - 22. In them, if storedCustomersString is null, then no customers were stored and line 18 puts an empty array, kind of like an empty bucket, in the allCustomers variable.

If storedCustomersString is not null, then your JavaScript found some stored customers. Line 18 parses the storedCustomersString, using JSON.parse, so you have the array of existing customers in the allCustomers variable. By line 23, allCustomers has an array in it. It might be an empty array or it might be an array with stored customers in it. Either way, you're ready to add in the new customer.

Line 23 reads,"Hey, array in the allCustomers variable, add the associative array in aClothingCustomer to your end spot." Now, if the array had no customers in it, it now has one. If it had 15 it now has 16. Regardless of how many customers were already in the array line 23 puts the new customer on the end of the array.

The rest of this function is so similar to the storeNote function in the previous section that I won't go over all the lines. Instead, let's focus on line 25.

You remember how localStorage is an associative array? And you remember how to use a key to get a value out of localStorage (line 16) or any other associative array? Well, line 25 is how you associate a key with a value...and you don't have to use setItem to do it! Let's read it. "Hey localStorage, create a key called 'all_customers' and associate the string in the allCustomersString variable with that key."

I know...you're worried. "What if the key is already there?" Well...if it is, that's fine. The string in allCustomersString will be associated with 'all_customers' and any previous associations will be gone. That's why we got the string, parsed it to get the array, and then added to the array. That way no customers get deleted.

Here is an analogy that might help. Associative array keys are like grade school kids. Young children think they can only have one

'best friend.' If they get another best friend then the old best friend isn't a best friend any more. Associative array keys only associate with one value at a time. In this app, that value is a string representing all of the customers in the array.

Did you catch that? Since a key can only be associated with one thing at a time, you have to convert the array of customers into one thing. That one thing is a JSON string. Do this, and now you've fooled localStorage into associating a key with multiple things. Kind of like getting a young child to have more than one best friend like adults do.

The second function, showAllCustomers, is so much like showAllNotes that you hardly need my help.

```
32  function showAllCustomers(){
33          var storedCustomersString = localStorage["all_customers"]
34          if(storedCustomersString != null){
35                  var allCustomers = JSON.parse(storedCustomersString)
36                  var customerDisplayer =
                                document.getElementById("all_customers_display")
37                  customerDisplayer.innerHTML = null
38                  var numberOfCustomers = allCustomers.length
39                  for (var i = 0; i < allCustomers.length; i++) {
40                          var aClothingCustomer = allCustomers[i]
41                          customerDisplayer.innerHTML += "<hr><p>Customer: "+
                                        aClothingCustomer["name"] +"</p>"+
                                        "<p>Age: "+ aClothingCustomer["age"] +
                                        "</p>"+"<p>Inseam: "+
                                        aClothingCustomer["inseam"] +"</p>"
42                  }
43          }
44  }
```

Code Snippet 3

Details:

The only thing new here is line 41....and that isn't really new. In it you see, again, how to get a value associated with a key out of an associative array. Looks familiar doesn't it. Yep. It works just like getting something out of localStorage. Why?? Because localStorage is an associative array. Remember? So that's it for the JavaScript. It must be time to make this app look better.

The CSS for this app is identical to that for the note it app in the previous section EXCEPT for the stuff right before the style's clos-

ing tag. Take a look in Code Snippet 4 and find the CSS there for all images. Remember, img means image.

```
<style>
        body{
            background-color: LightGrey;
        }
        input{
            border: none;
            border-radius: 5px;
            box-shadow: 0px 0px 5px #666666 inset;
            height: 25px;
            width: 200px;
            text-indent: 5px;
        }
        input:focus{
            outline:none;
        }
        #note_entry_area{
            width: 500px;
            padding-left: 5px;
            border:none;
            border-radius:5px;
            box-shadow:0px 0px 5px #666666 inset;
            text-indent:5px;
        }
        textarea{
            margin-top: 10px;
            border:none;
            border-radius:5px;
            box-shadow:0px 0px 5px #666666 inset;
            width:90%;
            height: 150px;
            text-indent:5px;
        }
        img{
            margin-top: 5px;
            margin-bottom: 5px;
            border:none;
            border-radius:5px;
            box-shadow:0px 0px 5px #666666;
        }
</style>
```

Code Snippet 4

Details:

The CSS for the images in this app, and there is only the add_customer.png image, is very much like that for the input tags that you did in the last app and for this one. The difference is on the last line of the images' CSS. On the input tags the box shadow is inset. That means the shadow ends up on the INSIDE of the input box when it is displayed. That make the input box look like it is lower than the rest of the page.

You want your button to look like it sticks out of the page a little bit. All you have to do is make the shadow be on the outside of the image. To do this, just don't tell the shadow to be inset. Got it? By default the shadow will be outside of the image.

With this last little bit of web stuff, the app looks like this.

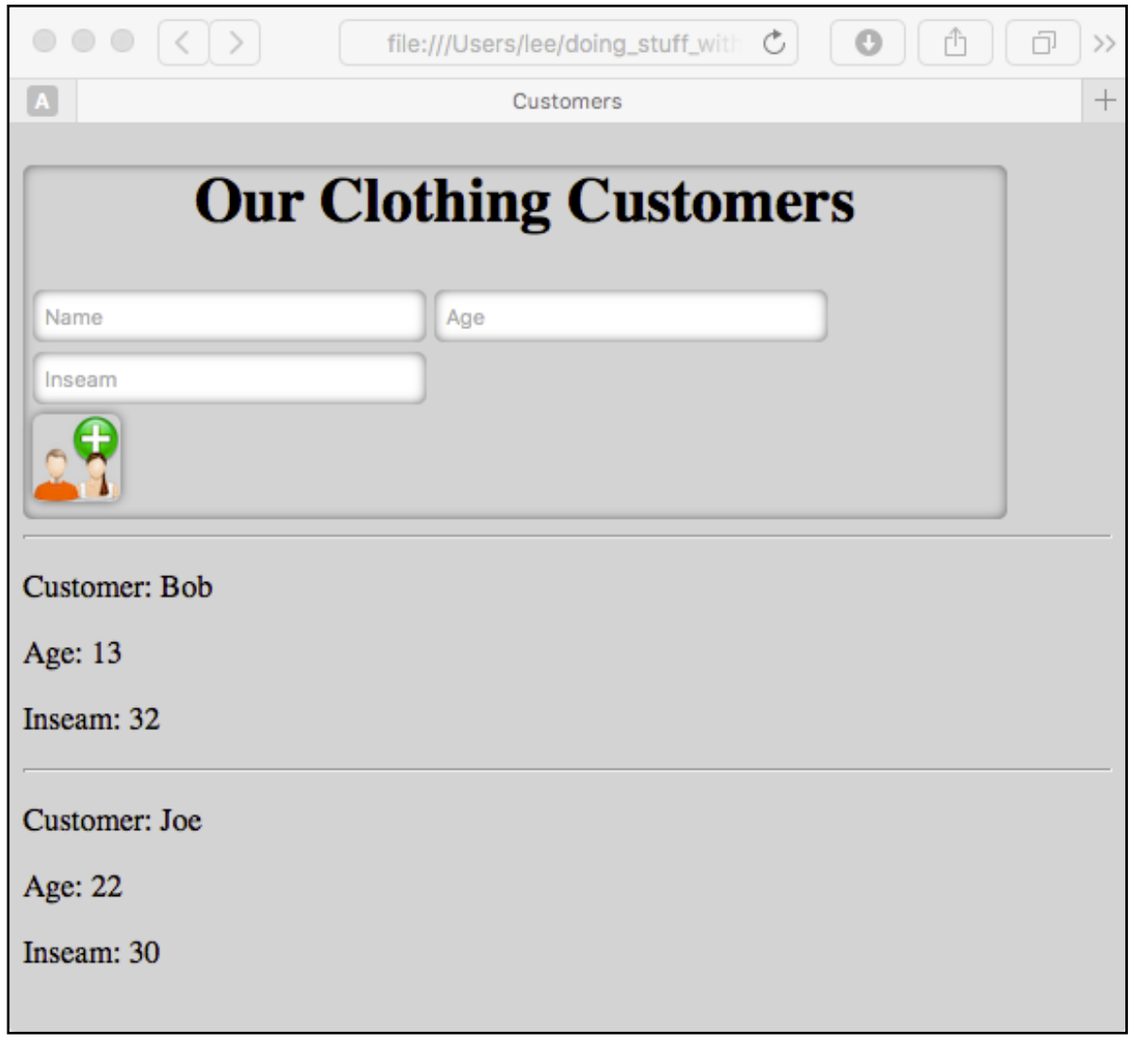


Figure 3. *The completed customer.html page with customers added.*

Where Next?

A good start is only a start. If you don't keep going did you really start or are you right where you began?

There are still many things about JavaScript, HTML, and CSS you don't know. A few of these are listed here with links to help in your discovery process.

Putting your JavaScript in its own file.

Putting your CSS in its own file.

Using CSS to move around stuff on your pages.

Also, a good tutorial or additional classes would be a good idea. Browse through any complete JavaScript, HTML, and CSS Tutorial, to see what else is possible. Then talk with others new to web stuff about it. And most important of all....PLAY with what you find. Go have some fun!!