Javascript: JSON and Ajax

Introduction to JSON

Welcome to JavaScript 2! In this course, you'll learn advanced JavaScript, JSON and AJAX and how to use them to suit your professional and creative goals.

Course Objectives

When you complete this course, you will be able to:

- use JSON to serialize data for storage in the browser or on the server.
- store and retrieve data using Ajax and LocalStorage.
- optimize your DOM manipulation code with Document Fragments.
- use Strings and Dates more effectively in your code.
- catch errors with Exceptions.
- add location and maps to your applications with Geolocation and Google Maps.
- modularize your code with Modernizr.
- build a dynamic, interactive, front-end web application.

From beginning to end, you will learn by doing your own JavaScript-based projects using JSON and AJAX, and then handing them in for instructor feedback. These projects will result in an impressive final application which will add to your portfolio and will contribute to certificate completion. Besides a browser and internet connection, all software is provided online by the O'Reilly School of Technology.

Throughout the course, we'll build a To Do application that uses form validation, local storage, and Ajax. Ajax is a term used to describe methods of communicating with resources external to your JavaScript program in order to send and retrieve data—resources like a file, a database, or a web service—as well as responding to user interactions and updating your web application dynamically. One of the key components of Ajax is the data, so after a quick review of objects and arrays, we'll take a look at how we can format data that we want to use in our web applications.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from

misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.

- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove will look like this.

We may also include instructive comments that you don't need to type.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```
The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look lik
e this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

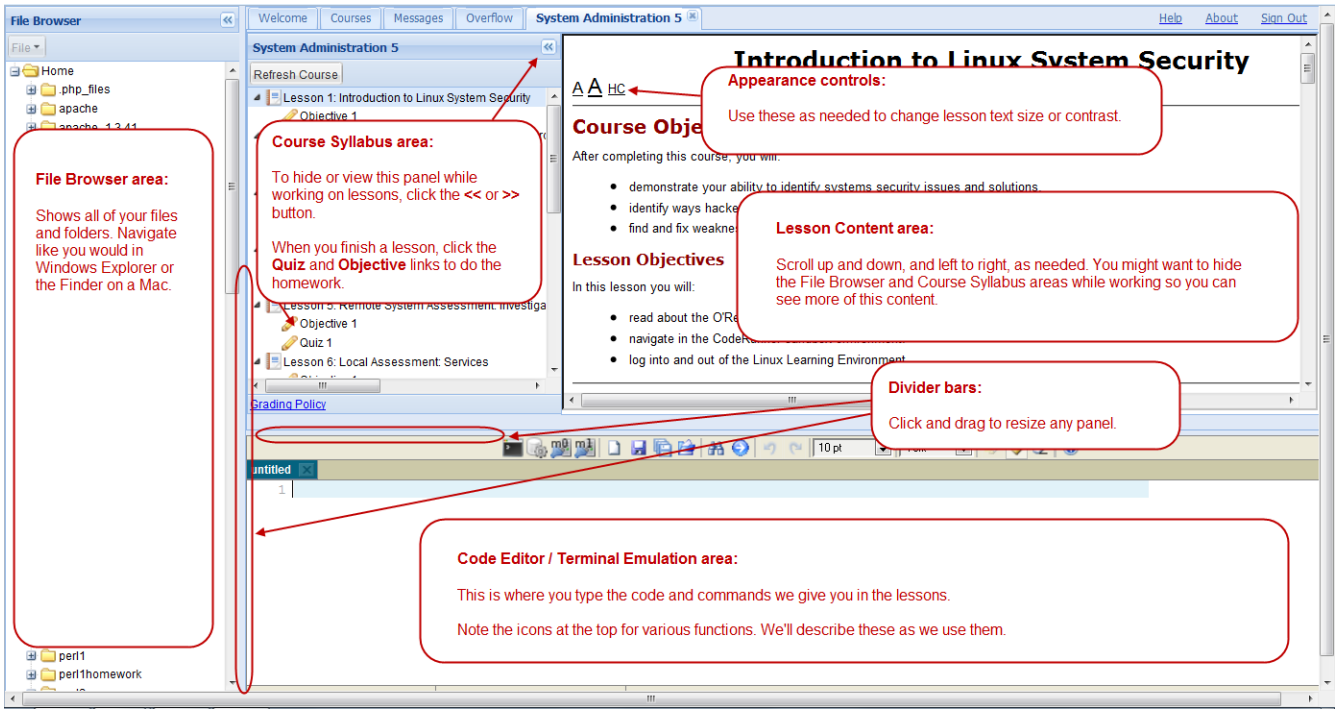
Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's

a quick overview of the various areas of the screen:



These videos explain how to use CodeRunner:

File Management Demo

Code Editor Demo

Coursework Demo

Review of Arrays and Objects

In the JavaScript 1 course, you learned about arrays and objects. You learned that an array is a collection of values, usually related in some way. For instance, you can create an array of colors like this:

OBSERVE:

```
var springColors = [ "AF7575", "EFD8A1", "BCD693", "AFD7DB", "3D9CA8" ];
```

...or an array of temperatures like this:

OBSERVE:

```
var temperatures = [ 47.5, 63.2, 56.7, 53, 51.2 ];
```

You also learned about objects. You learned that an object usually describes a thing, for instance a pet cat, and consists of properties of that thing, each of which has a name and a value:

OBSERVE:

```
var pickles = {
    type: "cat",
    name: "Pickles",
    weight: 7
};
```

Let's make a simple HTML page with a JavaScript program to create an array and an object, and display their values in the console. Create a new file in the Code Editor as shown:

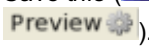
CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>Introduction to JSON</title>
  <meta charset="utf-8">
  <script src="intro.js"></script>
</head>
<body>

</body>
</html>
```

Save this () as **intro.html** in your **javascript2/** folder (to create the folder, right-click the **Home** folder in the File Browser, select **New folder...** or press **Ctrl+n**, type the new folder name **javascript2**, and press **Enter**). As you can see, this web page has no content, but it does have a link to a JavaScript file, **intro.js**. Let's make that next. Click the **New File** () icon in the Code Editor toolbar and type the code shown:

CODE TO TYPE:

```
var springColors = [ "AF7575", "EFD8A1", "BCD693", "AFD7DB", "3D9CA8" ];
var temperatures = [ 47.5, 63.2, 56.7, 53, 51.2 ];
var pickles = {
    type: "cat",
    name: "Pickles",
    weight: 7
};
window.onload = init;

function init() {
    console.log(springColors);
    console.log(temperatures);
    console.log(pickles);
}
```

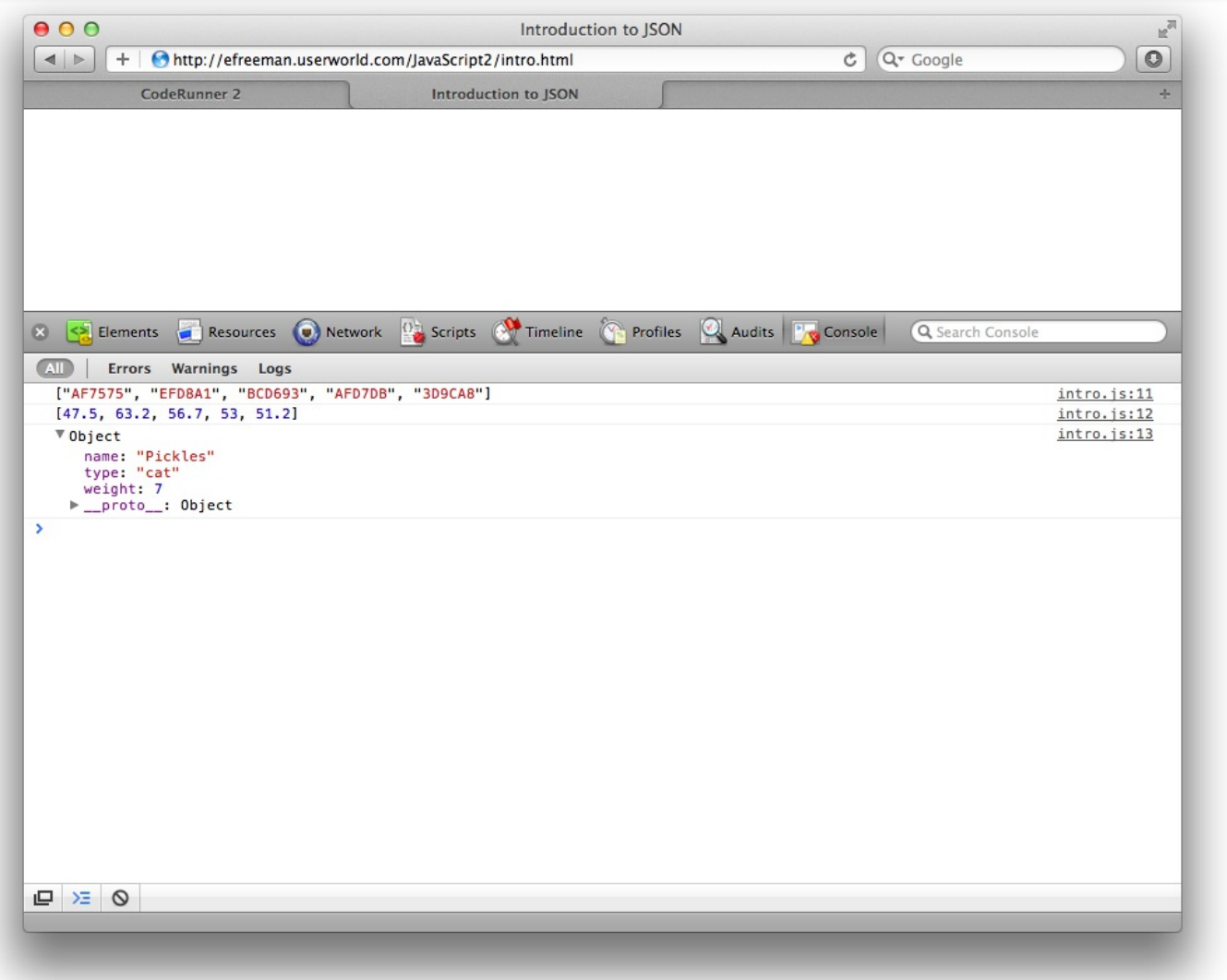
Save this () as **intro.js** in your **javascript2/** folder. Make sure you have **intro.html** open, and click Preview (

). A new empty browser window (or tab) appears.

To see the results of the JavaScript code, you need to open a console window. If you need to, you can review the video and the guide on how to access the console in your browser:

The developer tools are a little different in every browser, and they changing frequently as new versions of browsers are released, so you may have to be a little industrious and do some experimenting on your own. We've created a basic guide to get you started, and a short video that explains how to access the developer tools, for Safari, Chrome, and Firefox, and a video for Microsoft Internet Explorer 9.

When you open the console, if you don't see any results, just reload the page (intro.html). You'll see the values of the two arrays and the object.



Take a careful look at how these values are displayed; we'll come back to compare these values with other values shortly.

JavaScript Object Notation (JSON)

In this course, you'll learn how to communicate with web services from your JavaScript programs. A web service is a file or program on the web that often provides data you can use in your web application. But what kind of data, you might be asking? All kinds! There are many types of web services available on the internet that offer their data for use (Twitter, for example).

When you get data from a web service—even if it's just a local file on your own computer—you need to know what format the data is in. It could be just a plain text file containing words, or a comma-separated values (CSV) file, or even an XML file (eXtensible Markup Language, a common data interchange format).

The format we'll use in this course is called *JavaScript Object Notation* or *JSON*. Why are we using this format? Because the data format is almost identical to the way you write arrays and objects in JavaScript, so it's easy for you (a human) to read, and easy for JavaScript to understand! It's also a common format used to represent data in web applications.

Let's start by looking at an example of an array and an object written in JSON format. First, here's the **springColors** array written in JSON, and also as we wrote it earlier in our Javascript program:

OBSERVE:

```
JSON: ["AF7575","EFD8A1","BCD693","AFD7DB","3D9CA8"]
JavaScript: var springColors = [ "AF7575", "EFD8A1", "BCD693", "AFD7DB", "3D9CA8" ];
```

Compare the two formats, here and where it printed to the console. Pretty similar, right?

What about the **temperatures** array?

OBSERVE:

```
JSON: [47.5,63.2,56.7,53,51.2]
JavaScript: var temperatures = [ 47.5, 63.2, 56.7, 53, 51.2 ];
```

Again, compare these to how the array looks when it's printed in the console. It looks the same, right?

Finally, let's take a look at the **pickles** object:

OBSERVE:

```
JSON: {"type":"cat","name":"Pickles","weight":7}
JavaScript: var pickles = {
    type: "cat",
    name: "Pickles",
    weight: 7
};
```

This time, there's a slight difference: each of the property names ("type," "name," and "weight") is enclosed in double quotation marks. Otherwise, the JSON object is written the same way as the object in JavaScript. Note that the JSON object is different from objects displayed in the console, but that doesn't affect how you write the code to create the objects.

JSON can represent all of the core values you typically write in JavaScript: strings, numbers, and Boolean values, as well as arrays and objects that use these kinds of values in them. JSON can even represent objects that contain other objects and arrays. For instance, you can add the "likes" property to the **pickles** object, like this:

CODE TO TYPE:

```
var springColors = [ "AF7575", "EFD8A1", "BCD693", "AFD7DB", "3D9CA8" ];
var temperatures = [ 47.5, 63.2, 56.7, 53, 51.2 ];
var pickles = {
    type: "cat",
    name: "Pickles",
    weight: 7,
    likes: ["sleeping", "purring", "eating butter"]
};
window.onload = init;

function init() {
    console.log(springColors);
    console.log(temperatures);
    console.log(pickles);
}
```

 and 

, and the **pickles** object in the console now displays the **likes** property. If we represent the **pickles** object in JSON, it looks like this:

OBSERVE:

```
{"type":"cat","name":"Pickles","weight":7,"likes":["sleeping", "purring", "eating butte
r"]}
```

The only difference in the JSON representation of the **likes** property of the object and the Javascript is that in JSON, the property name is enclosed in double quotation marks ("likes").

So, *why* would you want to represent an object or array in JSON format, and *how* do you do it? I'm glad you asked...

Why Use JSON?

Imagine you run a pet adoption center, and you need a web application that does two things: allows you to enter new pets up for adoption, and shows you all the current pets available for adoption.

Instead of editing the whole HTML page each time you make a change to the list of pets for adoption, you want to be able to load the images of individual pets dynamically into your page whenever the page is loaded. If you store the data separate from the HTML and the JavaScript, then you won't have to edit the page each time (and risk making a mistake) and it will be a lot easier to update.

In addition, you have to represent the data about the pets somehow. You need to store the data in a format that your web application can understand. You could invent a format, or you could use an existing format like CSV, XML, or...JSON! Using JSON makes getting the data into your application really convenient. Let's look at how you might represent a list of pets available for adoption using JSON:

OBSERVE:

```
["Pickles", "Tilla"]
```

This is an array of all the pets available for adoption. It includes only their names; you could have much more information about each pet if you use objects. Create a new file that looks like this:

CODE TO TYPE:

```
[ { "type":"cat",
    "name":"Pickles",
    "weight":7,
    "likes":["sleeping","purring","eating butter"]
  },
  { "type":"dog",
    "name":"Tilla",
    "weight":25,
    "likes":["sleeping","eating","walking"]
  }
]
```

Save the file in your **javascript2/** folder as **pets.json**. Make sure you're in HTML mode, and you can click Preview (Preview) to see the contents of the file. Of course, it's just data, so it won't actually do anything.

Note Make sure to type the code above *just as you see it*! You can copy and paste the JSON into your file if you want, just to be certain.

We're not going to do anything with this file just yet; we'll get to that in a later lesson. For now, we'll learn how to turn an array of objects, just like the one above, into a JSON string in your JavaScript.

How to Use JSON in JavaScript

(Or: Serializing a JavaScript Object)

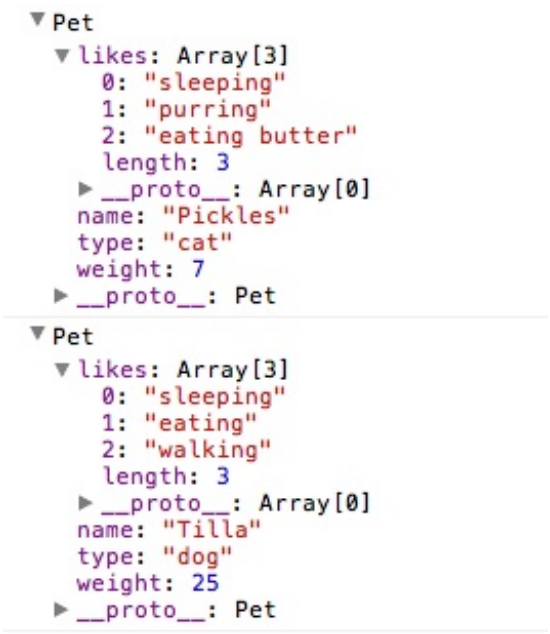
Let's say you have a JavaScript program with two pet objects. Update **intro.js** as shown:

CODE TO TYPE:

```
var springColors = [ "AF7575", "EFD8A1", "BCD693", "AFD7DB", "3D9CA8" ];
var temperatures = [ 47.5, 63.2, 56.7, 53, 51.2 ];
var pickles = {
    type: "cat",
    name: "Pickles",
    weight: 7,
    likes: ["sleeping", "purring", "eating butter"]
};

function Pet(type, name, weight, likes) {
    this.type = type;
    this.name = name;
    this.weight = weight;
    this.likes = likes;
}

window.onload = init;

function init() {
    console.log(springColors);
    console.log(temperatures);
    console.log(pickles);

    var pickles = new Pet("cat", "Pickles", 7, ["sleeping", "purring", "eating butter"]
);
    console.log(pickles);

    var tilla = new Pet("dog", "Tilla", 25, ["sleeping","eating","walking"]);
    console.log(tilla);
}
```

 Save it, open **intro.html**, and 

. In the console, you see the two Pet objects you created:



Now, let's turn these objects into JSON. Update **intro.js** as shown:

CODE TO TYPE:

```
function Pet(type, name, weight, likes) {
    this.type = type;
    this.name = name;
    this.weight = weight;
    this.likes = likes;
}

window.onload = init;

function init() {
    var pickles = new Pet("cat", "Pickles", 7, ["sleeping", "purring", "eating butter"]
);
    console.log(pickles);
    var picklesJSON = JSON.stringify(pickles);
    console.log(picklesJSON);

    var tilla = new Pet("dog", "Tilla", 25, ["sleeping","eating","walking"]);
    console.log(tilla);
    var tillaJSON = JSON.stringify(tilla);
    console.log(tillaJSON);
}
```

Save it, open **intro.html**, and Preview. In the console, you see the two Pet objects you created as before, and under each one, their JSON representations:

```
▼ Pet
  ▼ likes: Array[3]
      0: "sleeping"
      1: "purring"
      2: "eating butter"
      length: 3
    ▶ __proto__: Array[0]
    name: "Pickles"
    type: "cat"
    weight: 7
  ▶ __proto__: Pet
{"type":"cat","name":"Pickles","weight":7,"likes":["sleeping","purring","eating butter"]}
▼ Pet
  ▼ likes: Array[3]
      0: "sleeping"
      1: "eating"
      2: "walking"
      length: 3
    ▶ __proto__: Array[0]
    name: "Tilla"
    type: "dog"
    weight: 25
  ▶ __proto__: Pet
{"type":"dog","name":"Tilla","weight":25,"likes":["sleeping","eating","walking"]}
```

Compare the JavaScript objects with the JSON versions. They are very similar.

JSON.stringify()

To convert a JavaScript object to JSON, use the built-in **JSON** object and its method, **stringify()**:

OBSERVE:

```
var pickles = new Pet("cat", "Pickles", 7, ["sleeping", "purring", "eating butte
r"]);
var picklesJSON = JSON.stringify(pickles);
```

First, we create a Pet object, **pickles**. You've seen this kind of JavaScript before. Then, we use the

stringify() method of the **JSON** object, and pass in the **pickles** object as the argument. We get back a JSON version of the object, which we store in the variable **picklesJSON**. We did the same thing with **tilla** to turn the **tilla** object into JSON.

> **Note** The **JSON** object is built-in to JavaScript in all *modern* browsers, just like the **document** object, but you need to make sure you're using a fairly recent version of your favorite browser. That means IE9+, Safari 5+, Chrome 18+, Firefox 11+, or Opera 11+. Some older versions of some of these browsers have the JSON object, but for this course, use one of these versions (or later).

So now let's create an array of the two objects, and convert the array to JSON. Update **intro.js** as shown:

CODE TO TYPE:

```
function Pet(type, name, weight, likes) {
    this.type = type;
    this.name = name;
    this.weight = weight;
    this.likes = likes;
}

window.onload = init;

function init() {
    var pickles = new Pet("cat", "Pickles", 7, ["sleeping", "purring", "eating b
utter"]);
    console.log(pickles);
    var picklesJSON = JSON.stringify(pickles)
    console.log(picklesJSON);

    var tilla = new Pet("dog", "Tilla", 25, ["sleeping","eating","walking"]);
    console.log(tilla);
    var tillaJSON = JSON.stringify(tilla);
    console.log(tillaJSON);

    var petsArray = [ pickles, tilla ];
    var petsArrayJSON = JSON.stringify(petsArray);
    console.log(petsArrayJSON);
}
```

We are passing an array to JSON.stringify(). This is not a problem. As with objects, you can turn an array into JSON.

Save it, open **intro.html**, and Preview. In the console, you see the two Pet objects as before. Underneath the two objects, you see the JSON string representation of the array containing the two objects:

```
▼Pet
  ▶likes: Array[3]
   name: "Pickles"
   type: "cat"
   weight: 7
  ▶__proto__: Pet
▼Pet
  ▶likes: Array[3]
   name: "Tilla"
   type: "dog"
   weight: 25
  ▶__proto__: Pet
[{"type":"cat","name":"Pickles","weight":7,"likes":["sleeping","purring","eating butter"]},
{"type":"dog","name":"Tilla","weight":25,"likes":["sleeping","eating","walking"]}]
```

Compare the JavaScript objects' appearance with the appearance of the JSON-formatted array of objects. JSON is called "JavaScript Object Notation" because it looks just like JavaScript objects. The advantage is that a JSON-formatted object is a *string*. That means you can store it in a file, just like you did before when you created the **pets.json** file.

If you compare the output in the console that shows the JSON formatted array to what you typed into **pets.json**, you'll see they are *exactly the same* (ignore the extra white space you added in the file).

Turning an object into a string like this is known as *serializing an object*. It is incredibly useful because it allows you to store objects in plain text files, or transfer them between web applications, even if those applications are written in different languages. Let's say you have a Pet object in JavaScript, how would you give that object to, say, a PHP program? The only way you can do that is to serialize the object. Once the object is serialized, you can give it to the PHP program, and if the PHP program knows that the format is JSON, the program can *deserialize* the object (turn the string back into an object again), and voila! You've just transferred an object between two programs written in entirely different languages. That's cool.

Deserializing an Object

What if you've got an object represented in JSON and you want to get the object back? Update **intro.js** as shown:

CODE TO TYPE:

```
function Pet(type, name, weight, likes) {
    this.type = type;
    this.name = name;
    this.weight = weight;
    this.likes = likes;
}

window.onload = init;

function init() {
    var pickles = new Pet("cat", "Pickles", 7, ["sleeping", "purring", "eating b
utter"]);
    console.log(pickles);
    var picklesJSON = JSON.stringify(pickles);
    console.log(picklesJSON);

    var anotherPickles = JSON.parse(picklesJSON);
    console.log(anotherPickles);

    var tilla = new Pet("dog", "Tilla", 25, ["sleeping","eating","walking"]);
    console.log(tilla);

    var petsArray = [ pickles, tilla ];
    var petsArrayJSON = JSON.stringify(petsArray);
    console.log(petsArrayJSON);
}
```

Save it, open **intro.html**, and Preview. In the console, you see the JSON representation of the **pickles** object, and below that you see a new Pet object that was converted back into an object from JSON:

```
{"type":"cat","name":"Pickles","weight":7,"likes":["sleeping","purring","eating butter"]}
▼Object
  ▶likes: Array[3]
   name: "Pickles"
   type: "cat"
   weight: 7
  ▶__proto__: Object
```

So what did we just do?

OBSERVE:

```
var pickles = new Pet("cat", "Pickles", 7, ["sleeping", "purring", "eating butte
r"]);
var picklesJSON = JSON.stringify(pickles);
console.log(picklesJSON);
```

First, we created a new **pickles** object. Then we converted that object to JSON using **JSON.stringify()**, and stored the result in the variable **picklesJSON**. Next, we printed **picklesJSON** to the console.

OBSERVE:

```
var anotherPickles = JSON.parse(picklesJSON);
console.log(anotherPickles);
```

We took **picklesJSON** and we passed it to the **JSON.parse()** method. **JSON.parse()** does the opposite of what **JSON.stringify()** does: it takes a piece of data formatted as JSON, and converts it back into a JavaScript object (or array). This returns an object, **anotherPickles**, which we print to the console.

It's important to recognize that you get back another object, *not* the same object as our original **pickles** object. It's like **pickles**, in the sense that it has the same properties and property values, but now we have two *different* objects, **pickles** and **anotherPickles**.

So, **JSON.stringify()** takes an object and **serializes** it into JSON format; and **JSON.parse()** takes a piece of data in JSON format and **deserializes** it into a JavaScript object.

This allows you to transfer objects (or arrays) between JavaScript programs, between web services and web applications, and beyond. In the next couple of lessons, you'll see how you can use JSON with **Ajax** (a way of communicating with external resources from your JavaScript program) to load and save data from your web application.

JSON or XML?

You may have heard of XML or even used it in web applications before, as a data exchange format. You might be wondering whether we could also use XML to represent objects and arrays, and exchange data with web services. Yes, we could. In general, you may choose to use XML or JSON to represent your data; it depends on the context, the type of data, the web application, or the other programs or web services with which you might be exchanging data.

The good news is that anything you can represent in XML, you can also represent in JSON. While XML has additional features that execute tasks like validating your data to make sure it's correct, the kinds of data that can be represented are the same. So why choose JSON?

JSON is simpler to parse (that is, it's simpler to convert data in JSON format to a JavaScript object or array). A piece of data in JSON format is typically smaller in size than the same data in XML, and it's somewhat easier for us humans to read. JSON is well supported in JavaScript and web application tools, but it is not so well supported in other areas (for example, library systems, content management systems, and so on), where XML has broad support. For instance, you can get text editors that are specifically designed to help you write XML.

If you want to use your data in a variety of different contexts, you might choose XML. However, for web applications specifically, JSON is easier to work with; that's what we'll be using in this course.

Take some time to practice JSON and do the project before moving on to the next lesson. We'll use JSON to store and retrieve values throughout the rest of the course.

Introduction to Ajax

Lesson Objectives

When you complete this lesson, you will be able to:

- to create web pages that behave more like applications than pages.
- use Ajax to add new data to your web page without having to reload the page or load an entirely new page.
- send and receive data while you type, using asynchronous communication.
- update a page dynamically with new information, often with the data received from a server.

What Is Ajax?

We've mentioned Ajax a couple of times now. So, what is Ajax and how are we going to use it?

A few years ago, before any websites used Ajax, if you wanted to get information on the web, you clicked a link, waited for the current page to go away, and then waited for a new page to load with the information you wanted. For instance, if you were using a website like Amazon and searching a database of books, you entered a book title in the search bar, and waited for a new page to load with the results. Or if you were browsing Google for search results, each time you clicked "next," you waited for a new page to load with the results. This is the way we all expect the web to behave most of the time.

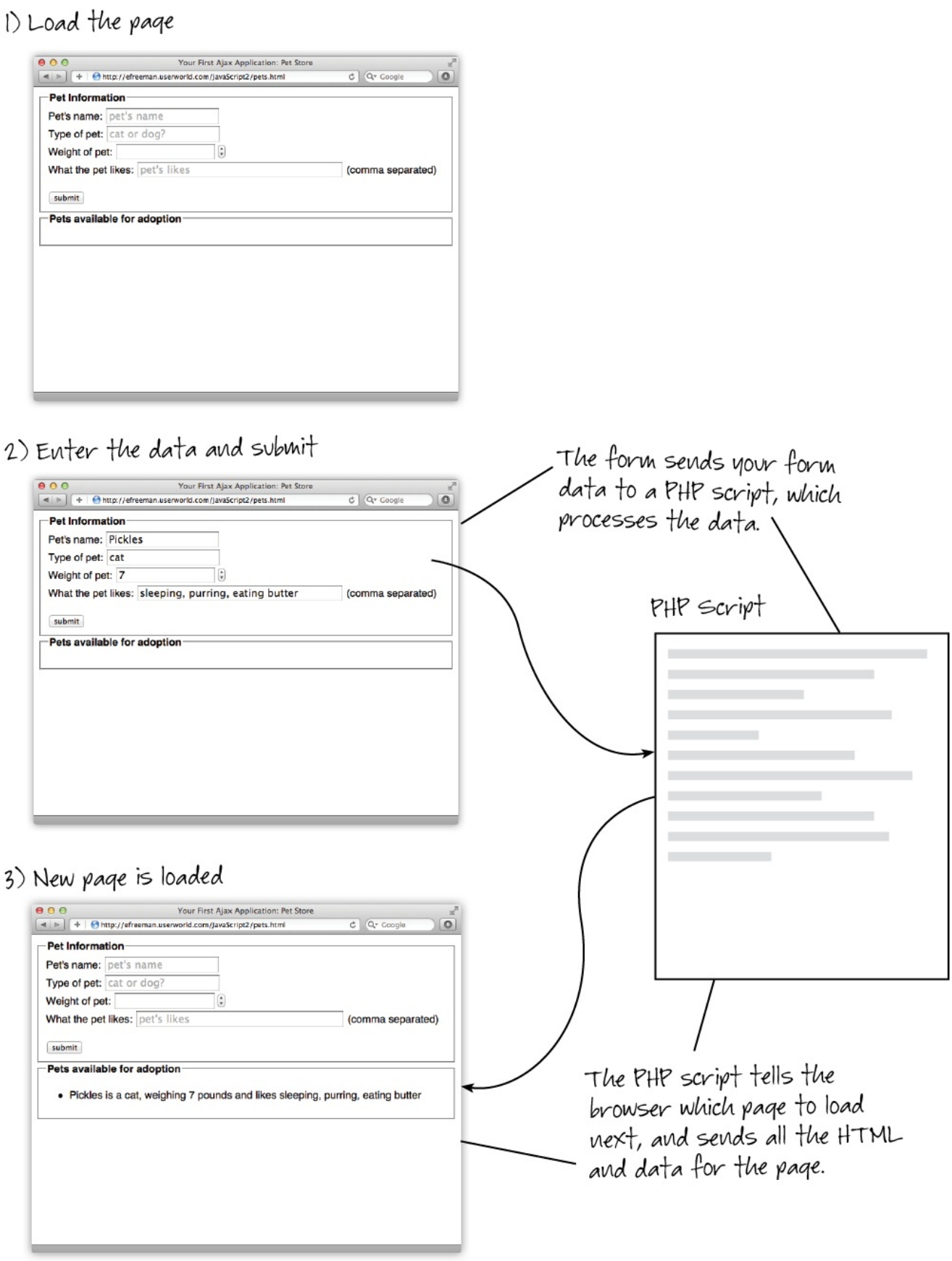
Now imagine you are using a desktop application like an editor. When you select a word and apply, say, "bold" to that word, you don't have to wait for the page to load, right? The word just turns bold right then and there. This is the way we expect applications to behave; we can see the results of the actions we take immediately, without having to wait for a new page to load with the result.

Ajax is a term that is used to describe a collection of techniques. These techniques allow you to create web pages that behave more like *applications* than *pages*. Some of these techniques you already know, like how to add and remove elements from the DOM so you can make your page respond to user input, or how to change the style of elements on the fly so you can change the way the page looks. You execute all of that without reloading the page or loading an entirely new page.

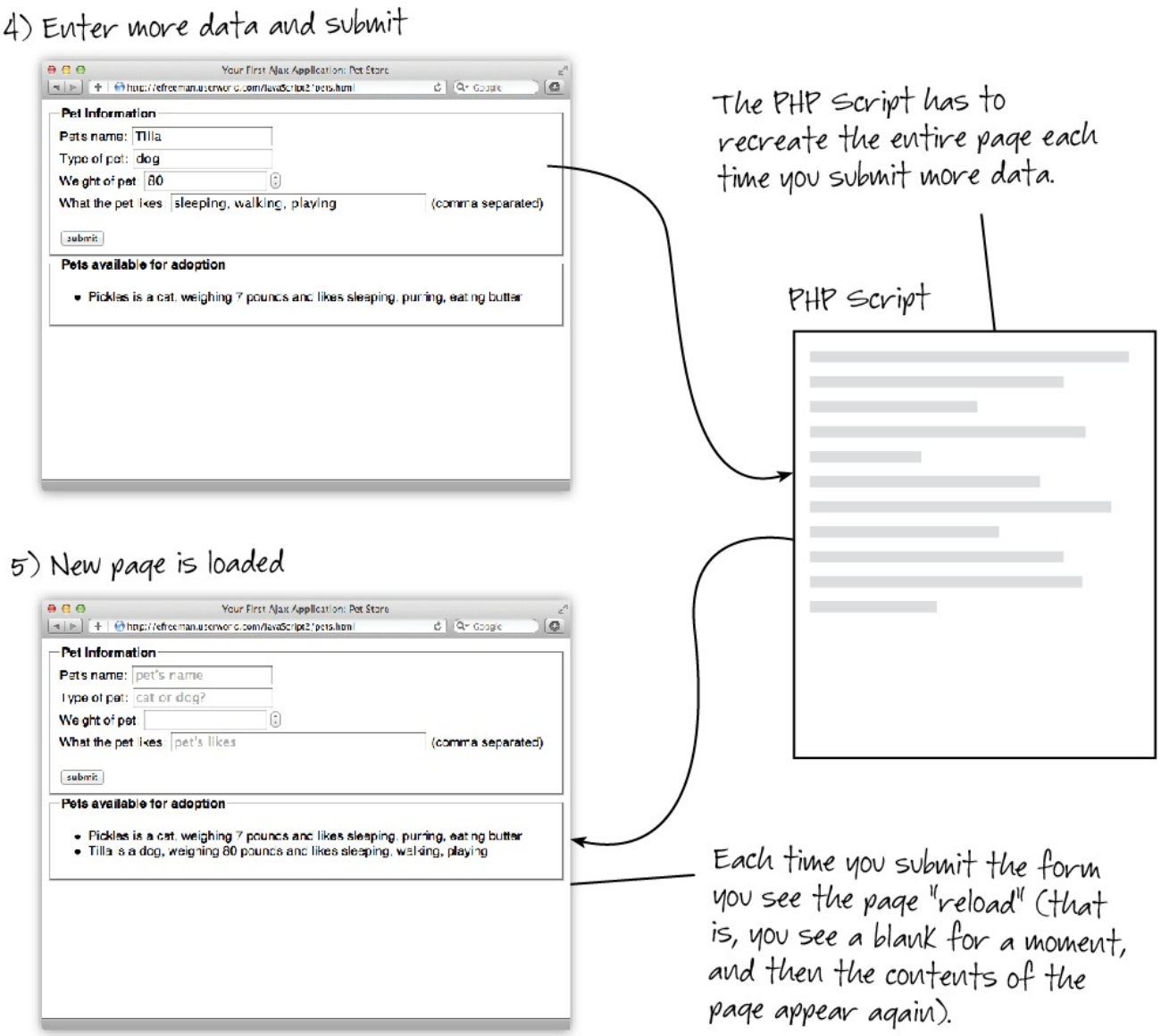
Ajax is also able to add new data to your web page, again without having to reload the page or load an entirely new page. That's important. Let's see what that means exactly; we'll take a look at how forms work without Ajax first, and then with Ajax, so you can see the difference.

Without Ajax

Imagine once again that you have a pet adoption center. The main page for your adoption center website includes a form at the top for submitting new pets available for adoption, and an area below where you display all available pets. If you submit a new pet using the form, it would work this way without Ajax:



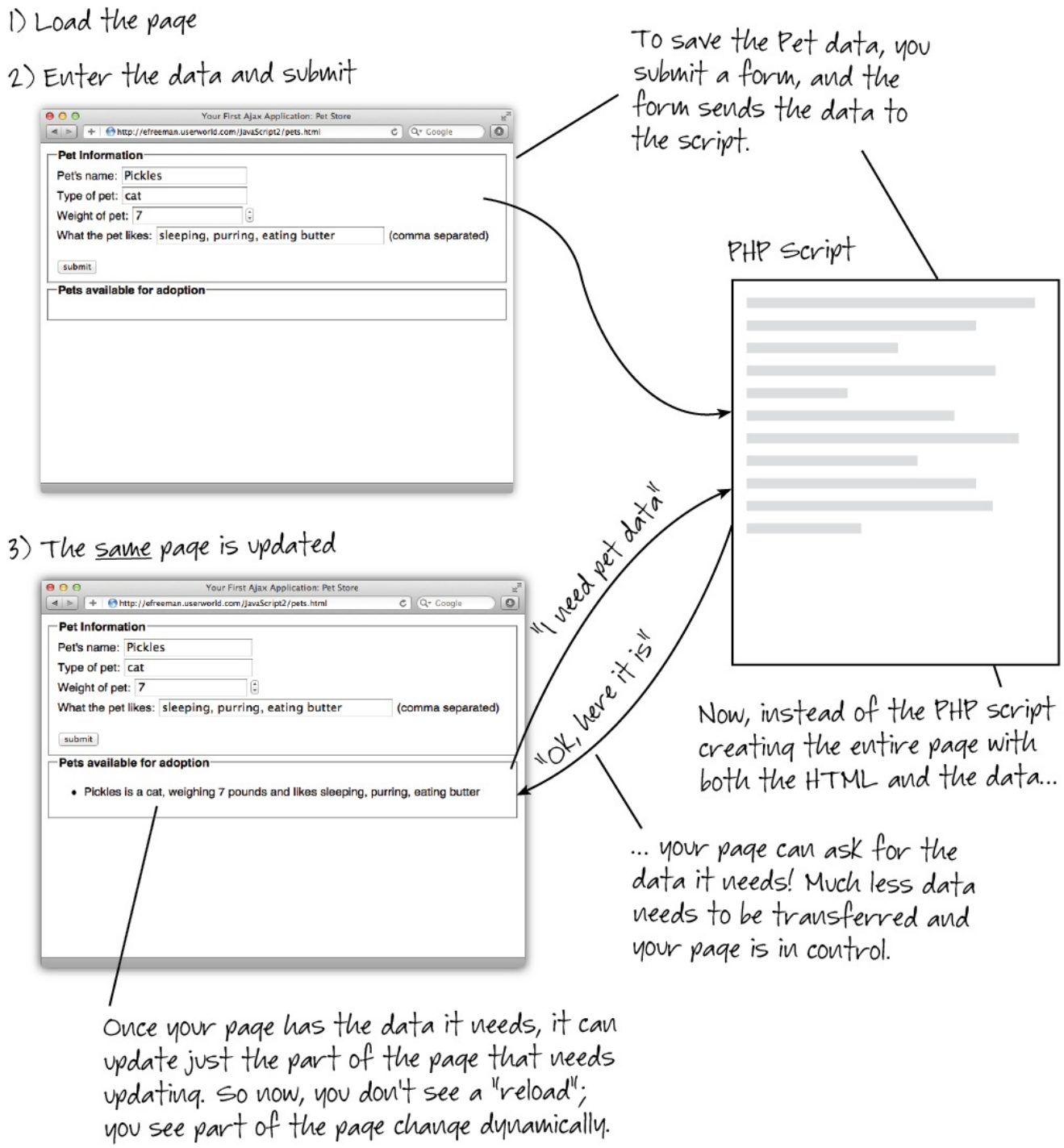
Now, if you wanted to submit a second pet, the same process would be required:



Each time you submit data for another pet, the PHP script has to recreate the entire page: the HTML and the data. This means you have to wait for the entire page to load each time you submit a new pet.

With Ajax

Now let's take a look at what this interaction looks like when we use Ajax:



We take these steps:

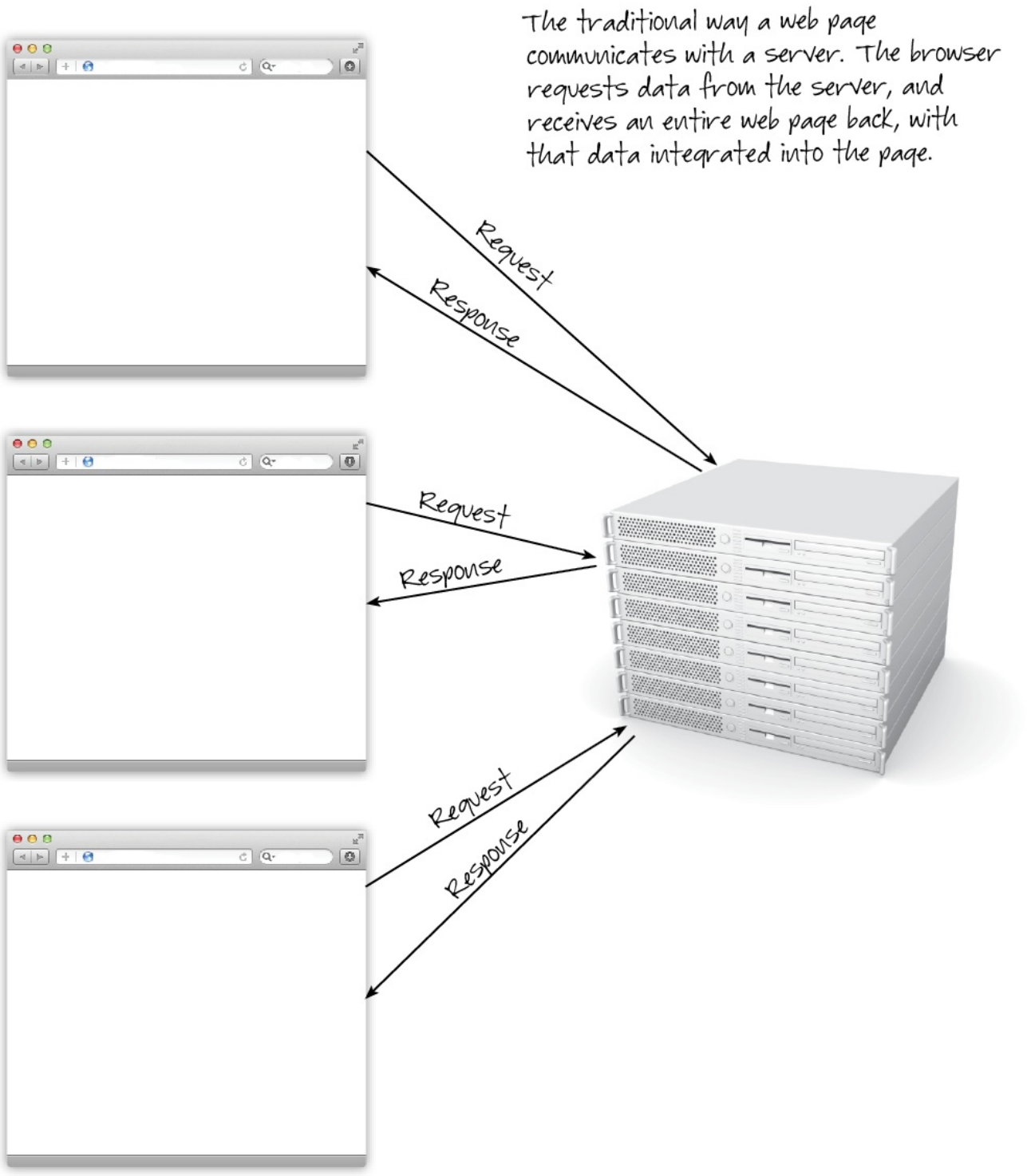
- Submit the form data to a PHP script.
- Stay on the current page, not loading a new page.
- Request pet data from the server script. Instead of responding with an entire web page, the server script responds with just the data we need.
- Take the data we get back from the server and update the page by adding a new list item to the DOM containing the pet information.

All these steps combined comprise "Ajax."

The Ajax Request/Response Model

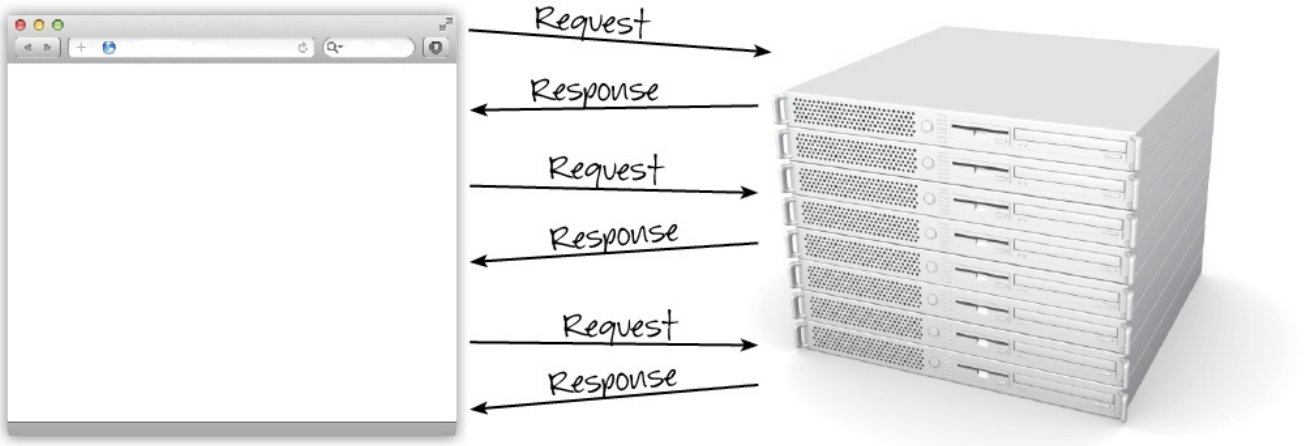
You already know a lot about how to update your page dynamically based on user interaction, so we'll focus on the

other primary capability of Ajax in the next few lessons: sending data to a server, requesting data from a server, and using data received from a server to update a page. In Ajax, this is known as the *Ajax Request/Response Model*. Let's compare the traditional model with the Ajax model again. Here's the traditional model:



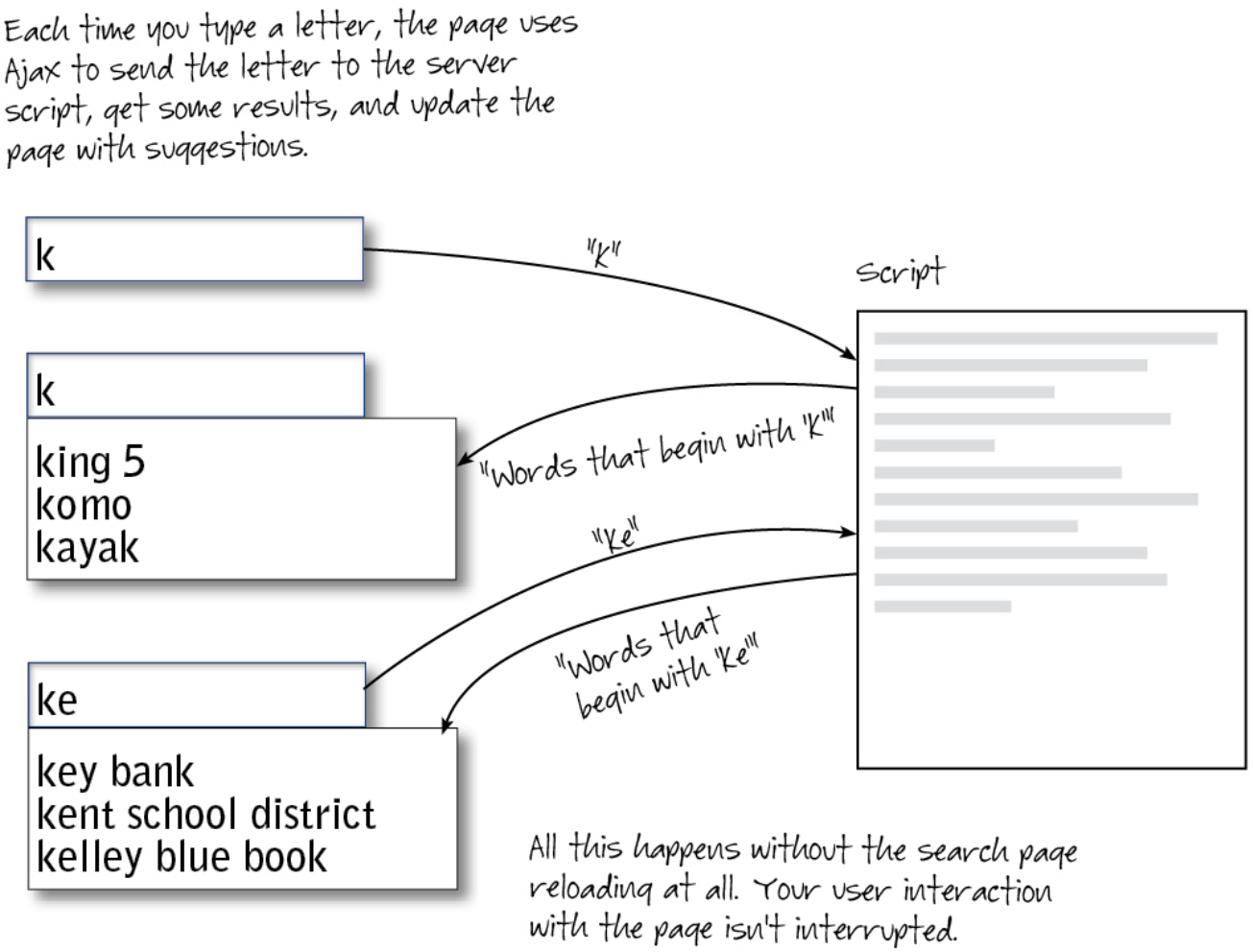
And here's the Ajax model:

With Ajax, the browser requests only data, and the server returns only data. The web page's content and appearance will change as a result, but there is no need for a whole new page to load.



Benefits of Ajax

One of the first companies to use Ajax in a big way was Google. You can see Ajax at work in many Google applications, including Google Maps and Google Search. Try going to use Google Search now. As you start typing in words, Google will suggest word completions for you. That's Ajax. Here's how it works:



This search page is:

- Not waiting until you submit the form to send data to the server. Instead, the page sends the information you type as you type it.
- Requesting small amounts of additional data, not an entirely new page.
- Not interrupting your user interaction experience with the page while the data you've typed is sent to the server. You just keep typing as you normally would.
- Getting the response from the server—the words that begin with the letters you've typed so far—and updating the page dynamically by adding those words to a list in the page.

It's not until you finish typing and submit your search request that a whole new page is loaded. Just imagine what this "suggest" experience would be like if a new page was loaded each and every time you typed a character into the search box—so slow and annoying!

The ability to send and receive data while you type is called **Asynchronous communication**. It means that your page can request data and wait for data without holding up the user interface of your web page. You can keep doing what you're doing on the web page while your page sends and waits for and receives data behind the scenes. This Asynchronous communication is the "A" in Ajax.

So, you've seen the *two key features and benefits* of using Ajax in your page:

- The ability to send data to and receive data from the server without causing the page you're on to hang or load a whole new page
- The ability to update a page dynamically with new information, often with the data received from a server

What You Need in Order to Use Ajax

Ajax isn't a single tool, but rather several technologies that work together to create web pages that act more like

applications. Creating Ajax web pages has become so popular that we now refer to pages like this as *web applications*. You'll find all kinds of web applications on the web, from mapping applications to games to utilities. Almost anything you can do on the desktop, you can now do on the web. So, what exactly do you need to create an Ajax web application?

- **A Web Browser** Web applications that use Ajax are built using web technologies, and used within a web browser.
- **HTML & CSS** You'll need HTML and CSS to create a web page to display the content of your web application in the browser.
- **JSON** You'll use the JSON format for the data that you'll send, store, and retrieve using your web application. JSON isn't required for Ajax applications; we could use another format instead, but we'll use JSON in this course.
- **JavaScript** JavaScript is where you'll do most of the work when you're building a web application. You'll use a built-in JavaScript object, the *XMLHttpRequest* object, to request and receive data from a web server. You'll see how this works in the next lesson. You'll use JavaScript to get data from a form, update a page by adding and removing elements from the DOM, as well as updating the style of your page.
- **A Web Server** The web server receives requests for data from your web application, and responds with that data. Sometimes this is as simple as hosting a file containing data on the web server, and requesting and receiving all of the contents of that file. Other times, you'll make requests to more complicated server applications that update and get results from a database or retrieve data from a web service like Twitter or Facebook. For this course, we'll keep it relatively simple because we're focusing on JavaScript, not server script development, but you'll get the idea (and you can work on more complicated server scripts if you decide to take the PHP or Ruby on Rails courses).

So get ready! In the next lesson, you'll be building your first Ajax web application!

Your First Ajax Application

Lesson Objectives

When you complete this lesson, you will be able to:

- create a simple HTML file with a little CSS for an Ajax application.
- use this JSON file as input data for an Ajax application.
- dynamically update your page with content loaded from a file using Ajax.
- use XMLHttpRequest to request the data.
- test and process the data you get back from a request.
- use content types and headers.

It's time to build your first Ajax application! As you learned in the previous lesson, you need these pieces:

- A web browser: check—you're reading this! :-)
- The HTML & CSS: we'll create a small HTML file shortly.
- The JSON Data: you need data; we'll be using the pets.json file you created in the Introduction to JSON lesson for this example.
- The JavaScript: you'll write the JavaScript code that uses Ajax.
- Web Server: we'll use the O'Reilly School of Technology web servers, just like we've done all along. If you want to use Ajax on your own computer at home, you'll need to set up a web server. We'll leave that up to you. Another option would be to get signed up for a hosted website, and upload all your files there.

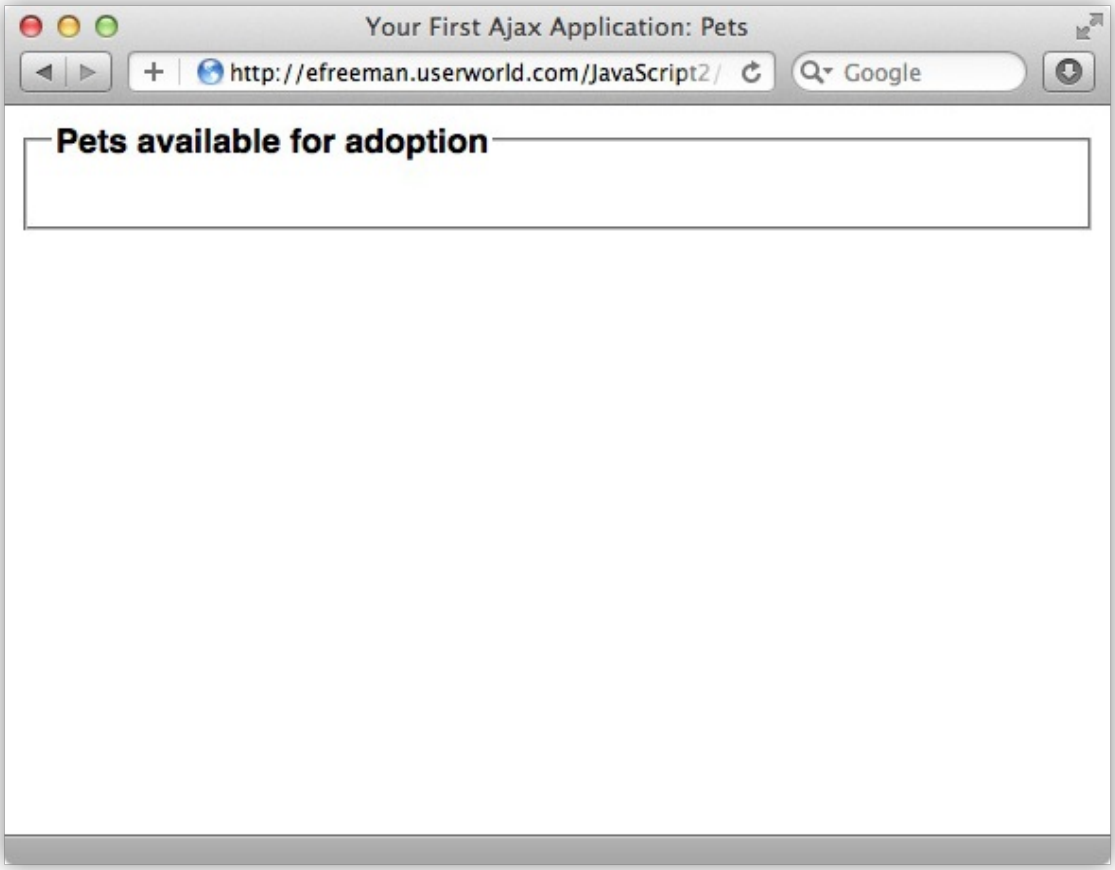
The HTML & CSS

Let's create a simple HTML file with a little CSS for our Ajax application. This application will load the pets data you entered earlier and then update the web page with that data:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>Your First Ajax Application: Pets</title>
  <meta charset="utf-8">
  <script src="pets.js"></script>
  <style>
        body {
            font-family: Helvetica, Arial, sans-serif;
        }
        legend {
            font-weight: bold;
        }
  </style>
</head>
<body>
<div>
<fieldset>
    <legend>Pets available for adoption</legend>
    <div id="pets">
    </div>
</fieldset>
</div>
</body>
</html>
```

Save this file in your **/javascript2** folder as **pets.html** and click Preview. You see the web page, it looks like like this:

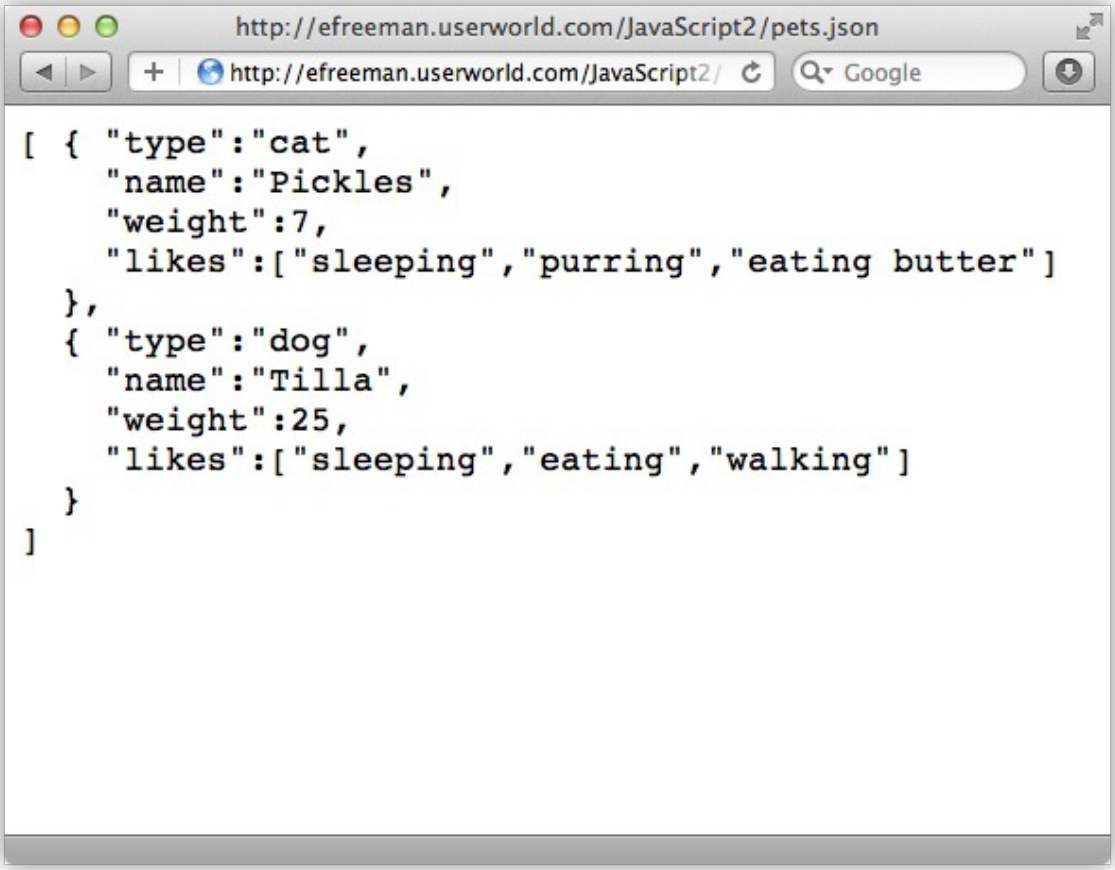


We link to a JavaScript file named **pets.js** in the head of the HTML; this is where you'll write JavaScript that uses Ajax to get the pets data and update the page. We'll update the page by adding the JSON we get from the **pets.json** file to the **"pets"** in the body of the HTML. We'll get to that in a bit.

In the HTML, we use a <fieldset> element. In case you haven't seen it before, you can use it to create a group of items. It's often used with forms to group form elements together. The text in the <legend> element is displayed at the top of the fieldset. Our list of pets will appear inside the fieldset once we start adding pets to it. We added a bit of CSS to change the font to a sans-serif font (which looks better on computer screens) and to make the legend text bold.

The JSON data

Next, open the **/javascript2/pets.json** file you created in the first lesson in HTML mode, and click Preview. We'll use this JSON file as input data for your first Ajax application. You see a web page with the JSON data, it looks like this:



The JavaScript

Now comes the really fun part—the JavaScript! Create a new file and type in this code:

CODE TO TYPE:

```
window.onload = init;

function init() {
    getPetData();
}

function getPetData() {
    var request = new XMLHttpRequest();
    request.open("GET", "pets.json");
    request.onreadystatechange = function() {
        var div = document.getElementById("pets");
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText != null) {
                div.innerHTML = this.responseText;
            }
            else {
                div.innerHTML = "Error: no data";
            }
        }
    };
    request.send();
}
```

Save this file in your **/javascript2** folder as **pets.js**. Open **pets.html**, and click Preview (Preview). Or, if your **pets.html** web page is already open, click **Reload** (to load the new JavaScript). You see the pets from the **pets.json**

file in your browser; it looks like this:

We're displaying only the JSON text, so it doesn't look very pretty, but you've just dynamically updated your page with content loaded from a file using Ajax. Congratulations!

Let's take a look at what's going on in this code. You probably recognize some of it, like where we set up the onload handler:

OBSERVE:

```
window.onload = init;

function init() {
    getPetData();
}

function getPetData() {
  ...
}
```

First we set up an **onload handler function** by setting the **onload property of the window object** to the **init** function. This tells JavaScript that after the browser has completed loading the page, it should call the **init()** function.

init() calls another function, **getPetData()**. **getPetData()** is responsible for getting the data in the **pets.json** file and loading it into the page.

Now we'll take a closer look at what's going on the **getPetData()** function.

The XMLHttpRequest Object

The **XMLHttpRequest** object is the heart of most Ajax programs. It lets you talk to the world outside the browser. You use it to send requests to a web server to retrieve data. As is the case with some other built-in JavaScript objects like **document**, **window** and **JSON**, **XMLHttpRequest** is a built-in object that is available in all modern browsers.

Note XMLHttpRequest is not supported by IE6. Still, even though IE6 isn't used much anymore, it's worth paying some attention to, just in case you need to support it with an Ajax application. In our example, you'll need to test to see whther the browser is IE6 and if it is, use the ActiveXObject instead.

OBSERVE:

```
function getPetData() {
    var request = new XMLHttpRequest();
    request.open("GET", "pets.json");
    request.onreadystatechange = function() {
       ...
    };
    request.send();
}
```

Let's break it down. First, we create a new **XMLHttpRequest** object and save it in the **request** variable. The **XMLHttpRequest** object contains properties and methods that you can use to make requests for data and receive responses to those requests.

We use **XMLHttpRequest** here to request the data in the file **pets.json**. To make the request for this data, we have to create a request; we do that with the **request.open()** method. We pass two arguments to this method: the **method** we're using to make the request, in this case **GET**, and the resource we're requesting, in this case the file **pets.json**. This is the *same* kind of **GET** request you make with an HTML form: it sends an HTTP request to a web server to "get" a resource. It is also common to make a request using **POST**; the differences between GET and POST are beyond the scope of this course.

The **request.open()** method doesn't actually send the request; to do that we need to use the method **request.send()**. But before we actually send the request, we want to indicate what we're going to do when we get a response!

The web server—in this case, the O'Reilly School of Technology web server— will get your request for the file **pets.json**. It will respond by sending back the data that you requested. What do we do when we get a response? Well, we'll call a function that we've assigned to the **request.onreadystatechange** property. This is similar to the way **window.onload** works; with **window.onload** you assign a function to the property that the browser will call when the page has finished loading. All you do is create the function; the browser calls it for you.

request.onreadystatechange works the same way. We assign a function to the **onreadystatechange** property of the **request** object. We're in essence saying, "When your **ready state changes**, call this function."

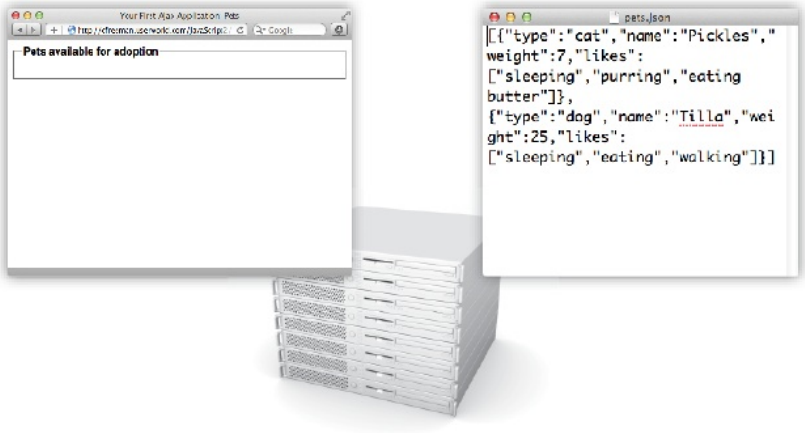
So, what's a ready state? Remember, the *request object* is the **XMLHttpRequest** object that you created to make all this Ajax communication happen; when it sends out a request, it goes through a series of *ready states* that indicate whether the response is ready for you to use. The ready state is one of several possible integer values that indicate the status of the XMLHttpRequest object. For instance, it could be sending the request, awaiting a response, or it could be done and ready for you to process the results. Here are the possible values for the ready state:

Number	Property	Description
0	UNSENT	open() hasn't been called yet.
1	OPENED	open() has been called.
2	HEADERS_RECEIVED	The headers have been received.
3	LOADING	The response body is being received.
4	DONE	The response is complete and ready for processing.

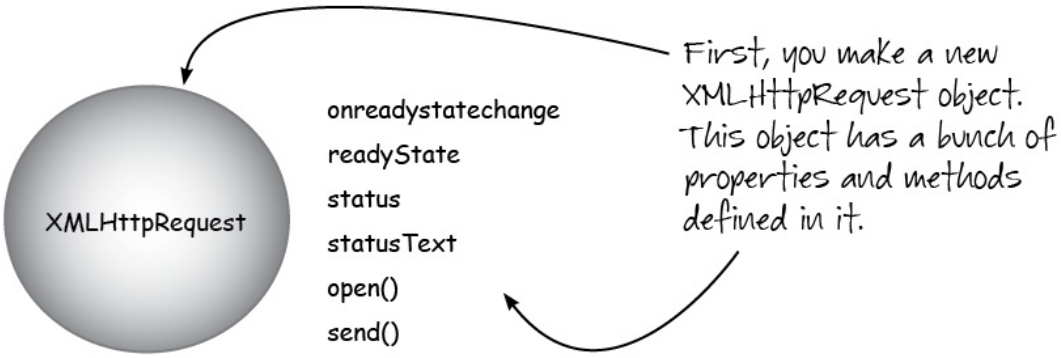
Here's an overview of how XMLHttpRequest works:

The browser, with pets.html loaded, and your Javascript code running in it.

pets.json

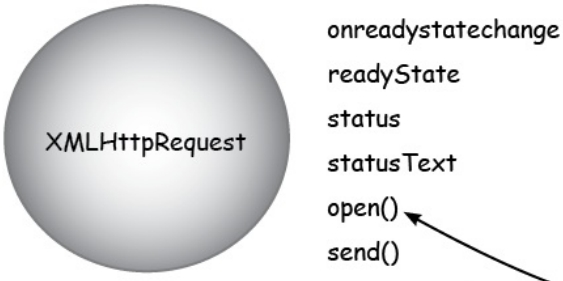


(1) var request = new XMLHttpRequest();



First, you make a new XMLHttpRequest object. This object has a bunch of properties and methods defined in it.

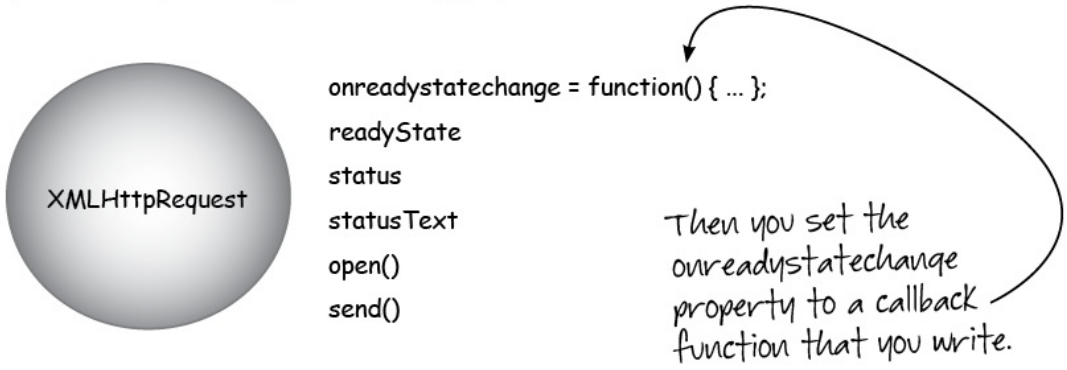
(2) request.open("GET", "pets.json");



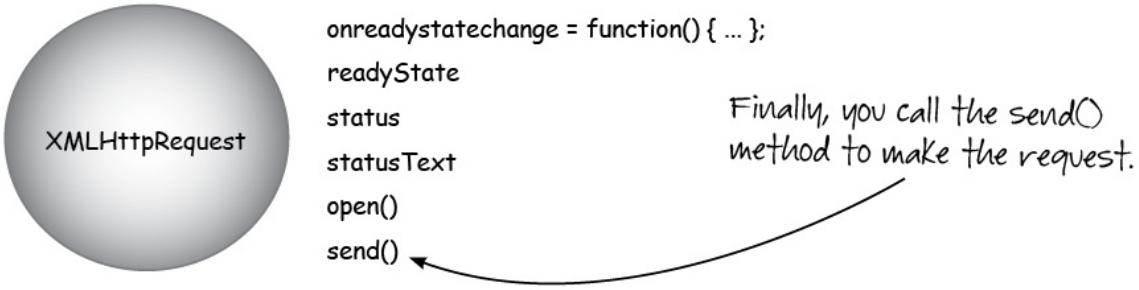
Next, you call the open() method in the request object, and tell it what kind of request you want to make.

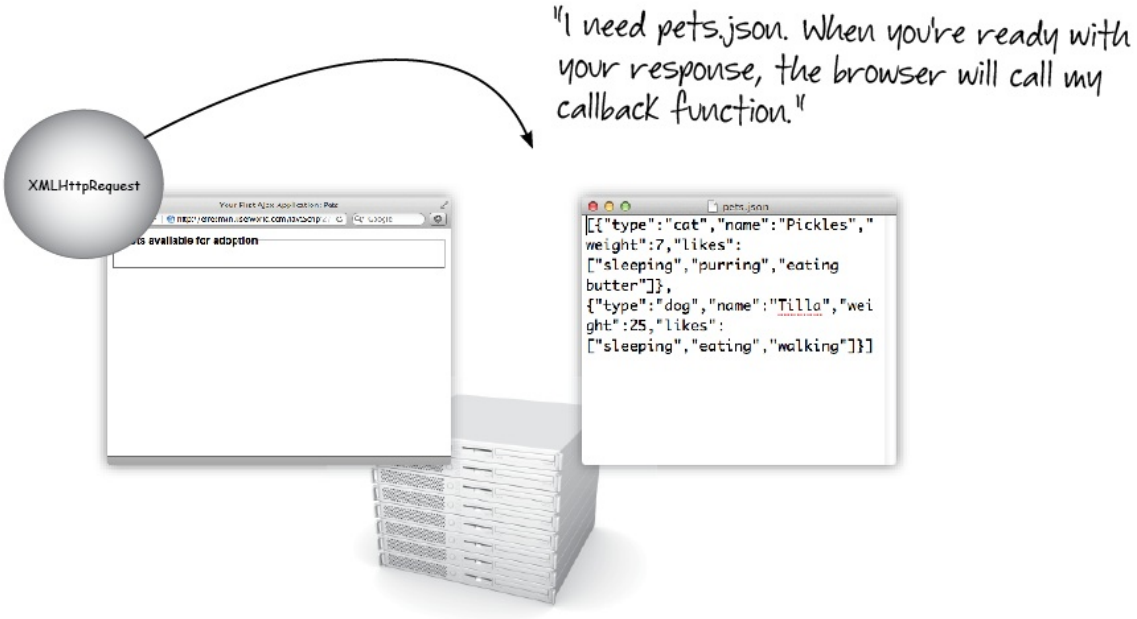
"GET", "pets.json"

(3) request.onreadystatechange = function() { ... };



(4) request.send();





Handling the Response

Now, let's delve into the function that gets called when our JavaScript gets a response from the server. This is the *callback function*:

OBSERVE:

```
function getPetData() {
    var request = new XMLHttpRequest();
    request.open("GET", "pets.json");
    request.onreadystatechange = function() {
        var div = document.getElementById("pets");
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText != null) {
                div.innerHTML = this.responseText;
            }
            else {
                div.innerHTML = "Error: no data";
            }
        }
    };
    request.send();
}
```

The **callback function** is an *anonymous function*. You've seen anonymous functions before, for instance, when we set the value of an object property to a method. The end result is that using an anonymous function is exactly the same as using a name, but it bypasses the step where the function is defined with a name. For instance, instead of:

OBSERVE:

```
request.onreadystatechange = function() {
   ...
};
```

...you could write:

OBSERVE:

```
request.onreadystatechange = handleRequest;
function handleRequest() {
   ...
}
```

In both cases, you're setting the value of the **onreadystatechange property** to a function; in the first example, that **function has no name**, while in the second, it has the name **handleRequest**. Using an anonymous function just skips the step where you name the function.

Anonymous functions are used frequently in JavaScript, so it's a good idea to get used to seeing them written.

Testing the Ready State

In our Ajax example, we set the value of the **request.onreadystatechange** property to a **callback function** that will be called when a change occurs in the *ready state* of the *request object*. Let's take another look at the code to see what happens in the **function** that's called when the ready state changes:

OBSERVE:

```
function getPetData() {
    var request = new XMLHttpRequest();
    request.open("GET", "pets.json");
    request.onreadystatechange = function() {
        var div = document.getElementById("pets");
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText != null) {
                div.innerHTML = this.responseText;
            }
            else {
                div.innerHTML = "Error: no data";
            }
        }
    };
    request.send();
}
```

We test *both* the **readyState** and the **status** in the **callback function**; we need to make sure both have a specific value before continuing. The **&&** makes sure that both expressions are true before the body of the **if** statement is run. Let's look at what the **readyState** and **status** tell us.

The ready state that we're most interested in is the **DONE** state; this indicates that the response is complete. So, in the **function**, we're testing to see whether the **readystate** is equal to the **DONE** value. But what is **this**?

You've actually encountered **this** before; in object methods, we can refer to the object whose method we're using as **this**. It refers to *this object*. It's works the same way here. By setting a **function** as the value of the **onreadystatechange** property of the **request** object, you've created an object method. So here, **this** is the **request** object itself—the object that contains the method.

Now that you know what **this** is, you can see that **this.readyState** is the ready state of *this object*—the request object. We're testing to see if it's **DONE**, where **DONE** is a property also defined in the request object, so we can compare the value of the **readyState** to the value of the **DONE** to see if they are equal. If they are, we know we got a response and that it's complete.

Testing the Status

But just knowing that the response is **DONE** isn't quite enough to be sure that we can proceed; we need to make sure that everything went okay. Just like when you type a URL into a browser window and send off an HTTP request to get a file, that request could fail if say, the server is down, or the file doesn't exist. You might have seen the infamous "500 Internal Server" error, or the "404 Not Found" error before.

We're sending an HTTP request here too; the server will respond and let us know if everything went okay or if something went wrong. This part of of the response is put into the request object's **status** property. If the **status** is 200, then we know everything went okay, and we can proceed.

Processing the Data

Ah, finally! Yes, it's taken a lot to get to this point, but now we're ready to process the data we got back from the request. We know that we have a response (readyState is DONE), and we know that everything went according to plan (status is 200), so let's get the JSON data in the response.

OBSERVE:

```
function getPetData() {
    var request = new XMLHttpRequest();
    request.open("GET", "pets.json");
    request.onreadystatechange = function() {
        var div = document.getElementById("pets");
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText != null) {
                div.innerHTML = this.responseText;
            }
            else {
                div.innerHTML = "Error: no data";
            }
        }
    };
    request.send();
}
```

When the response is complete, and if everything went as planned, the data is put into the request object's **responseText** property. Again, we refer to the request object using **this**, so we can access the data using **this.responseText**.

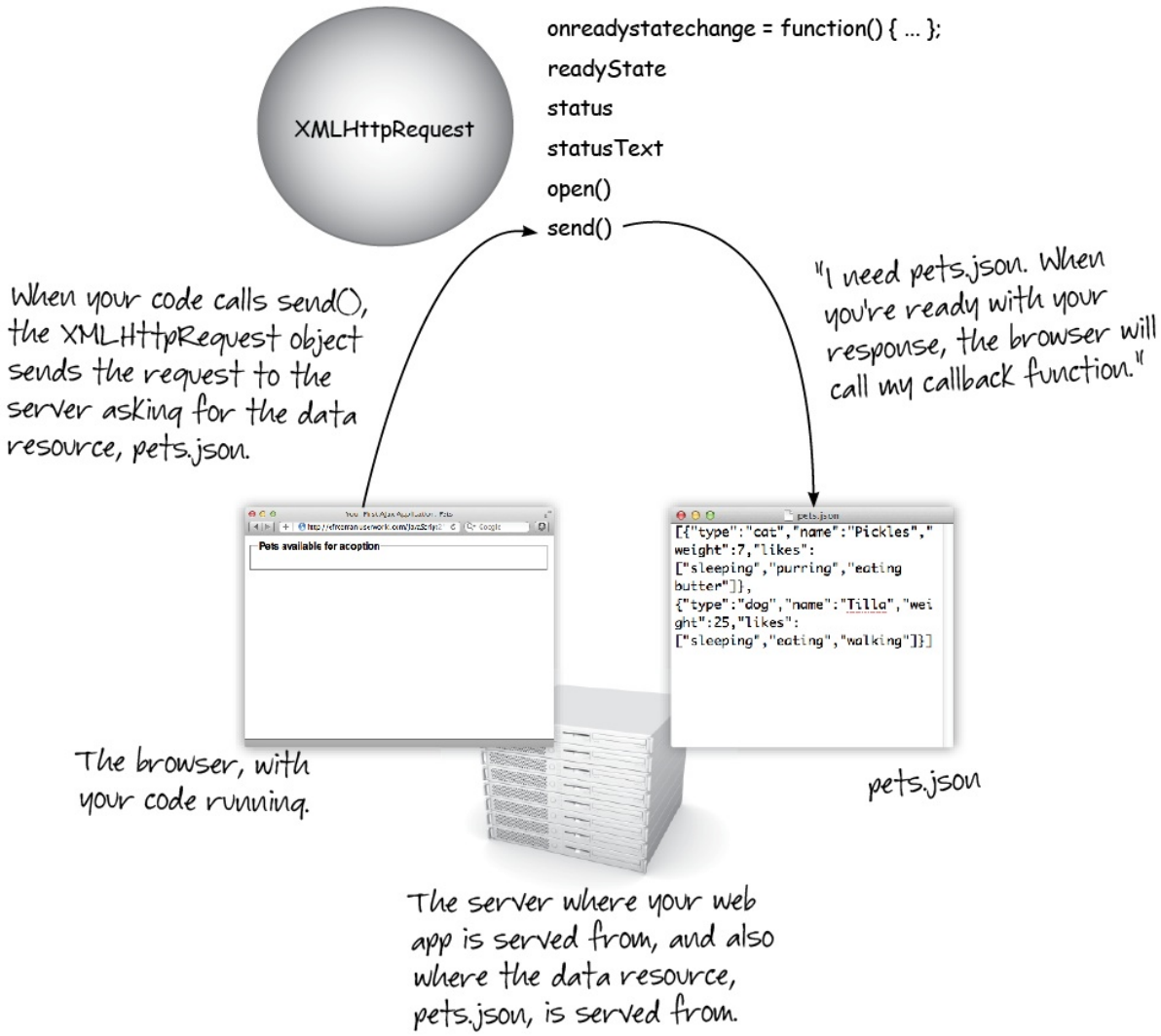
We want to make sure that the **responseText** actually has something in it, so we compare **responseText** to **null**. If it's not null, we know there is some data present, so we update the **innerHTML** property of the **"pets" ** with the JSON data that we got from the **pets.json** file. Notice that we got the **"pets" ** at the very top of the **callback function**. When you load the page, you see this data in the on the page.

If there was a problem and the "pets.json" file is empty, or the **responseText** didn't get set correctly and is null, then we can put the error message into the **"pets" ** instead.

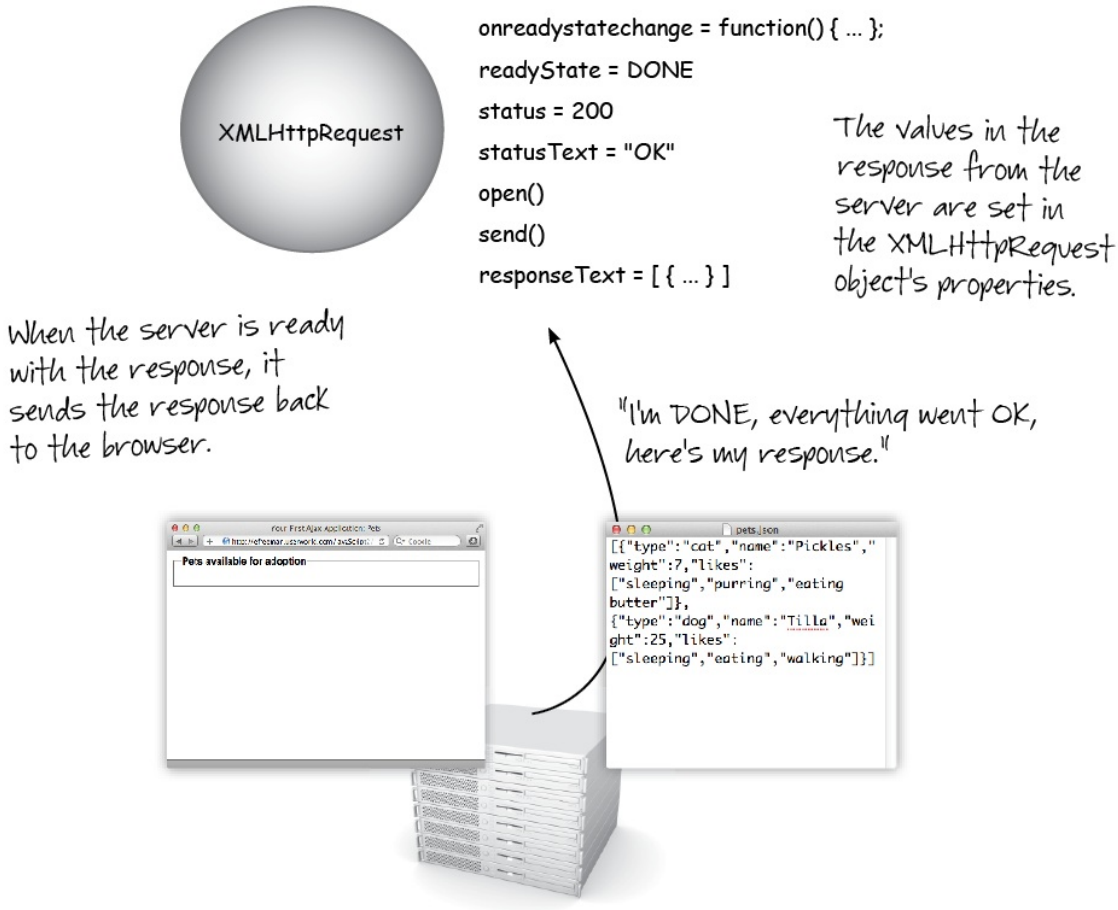
Overview of XMLHttpRequest Request and Response

Wow, that was a lot to go through. Study it and review the explanations above to make sure you understand each step. The **request object** contains *both* the request and the response. In other words, you use an **XMLHttpRequest** object to create and send the request to the web server, and also to get the response back from the server (including the ready state, the status, and the actual data). The **XMLHttpRequest** object handles all of that communication—to and from the server.

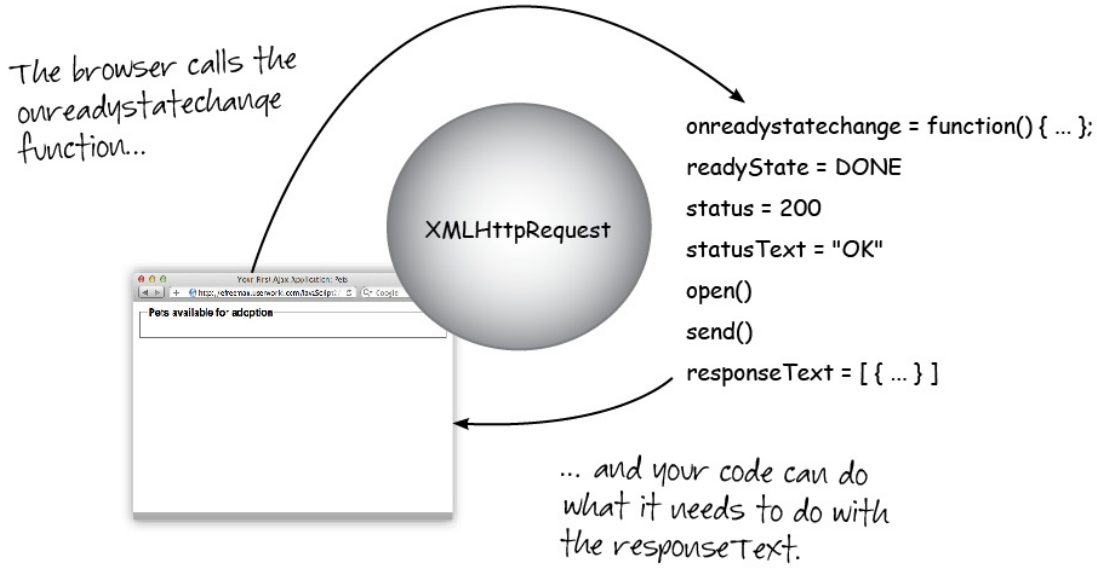
Step 1:


XMLHttpRequest
onreadystatechange = function() { ... };
readyState
status
statusText
open()
send()
When your code calls send(), the XMLHttpRequest object sends the request to the server asking for the data resource, pets.json.
"I need pets.json. When you're ready with your response, the browser will call my callback function."
Pets available for acoption
[{"type":"cat","name":"Pickles","weight":7,"likes":["sleeping","purring","eating butter"]},{"type":"dog","name":"Tilla","weight":25,"likes":["sleeping","eating","walking"]}]
The browser, with your code running.
pets.json
The server where your web app is served from, and also where the data resource, pets.json, is served from.

Step 2:



Step 3:



Asynchronicity Rocks!

As we learned earlier, one of the benefits of Ajax is that it is *asynchronous*. Let's take a closer look at exactly what that means:

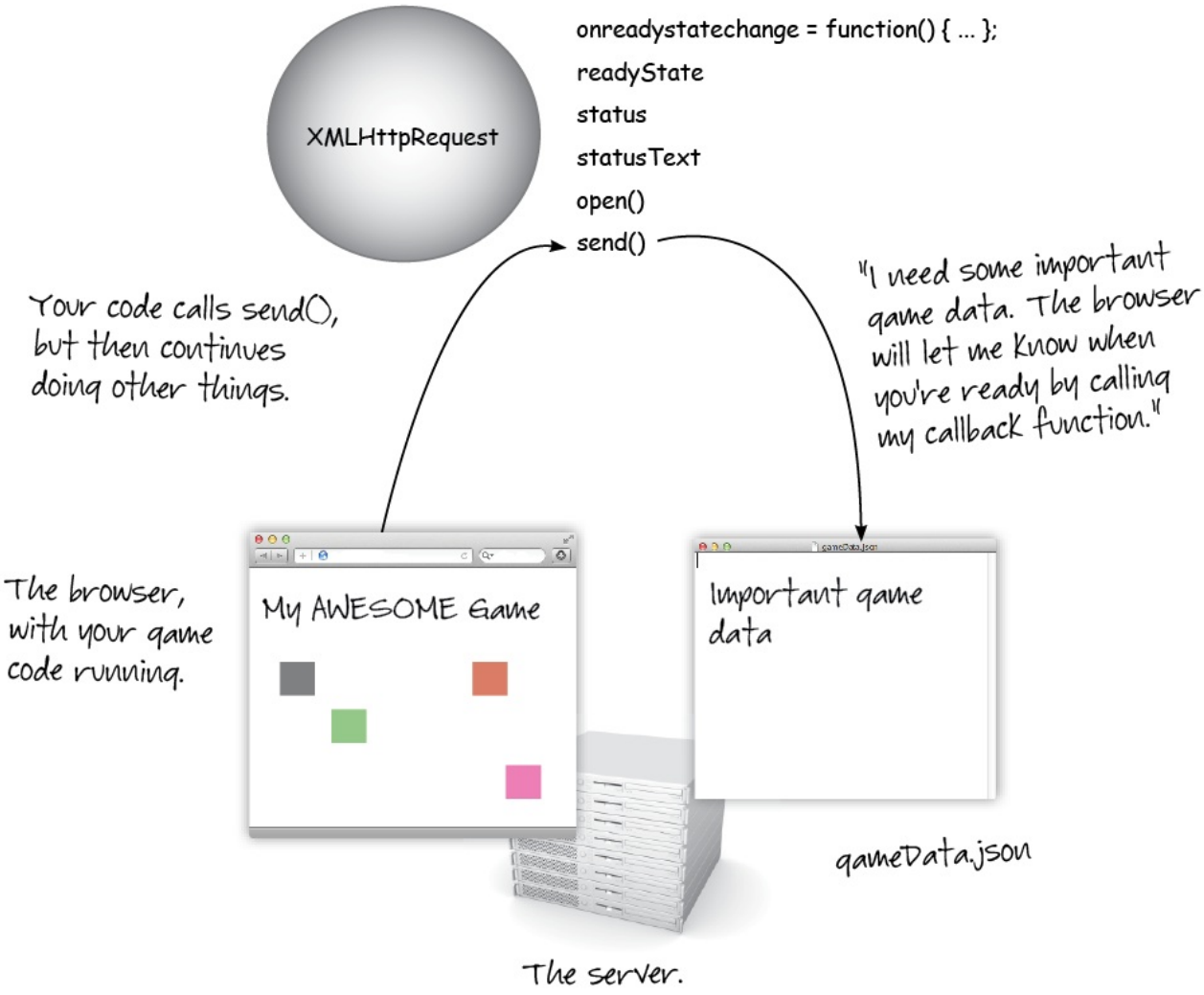
OBSERVE:

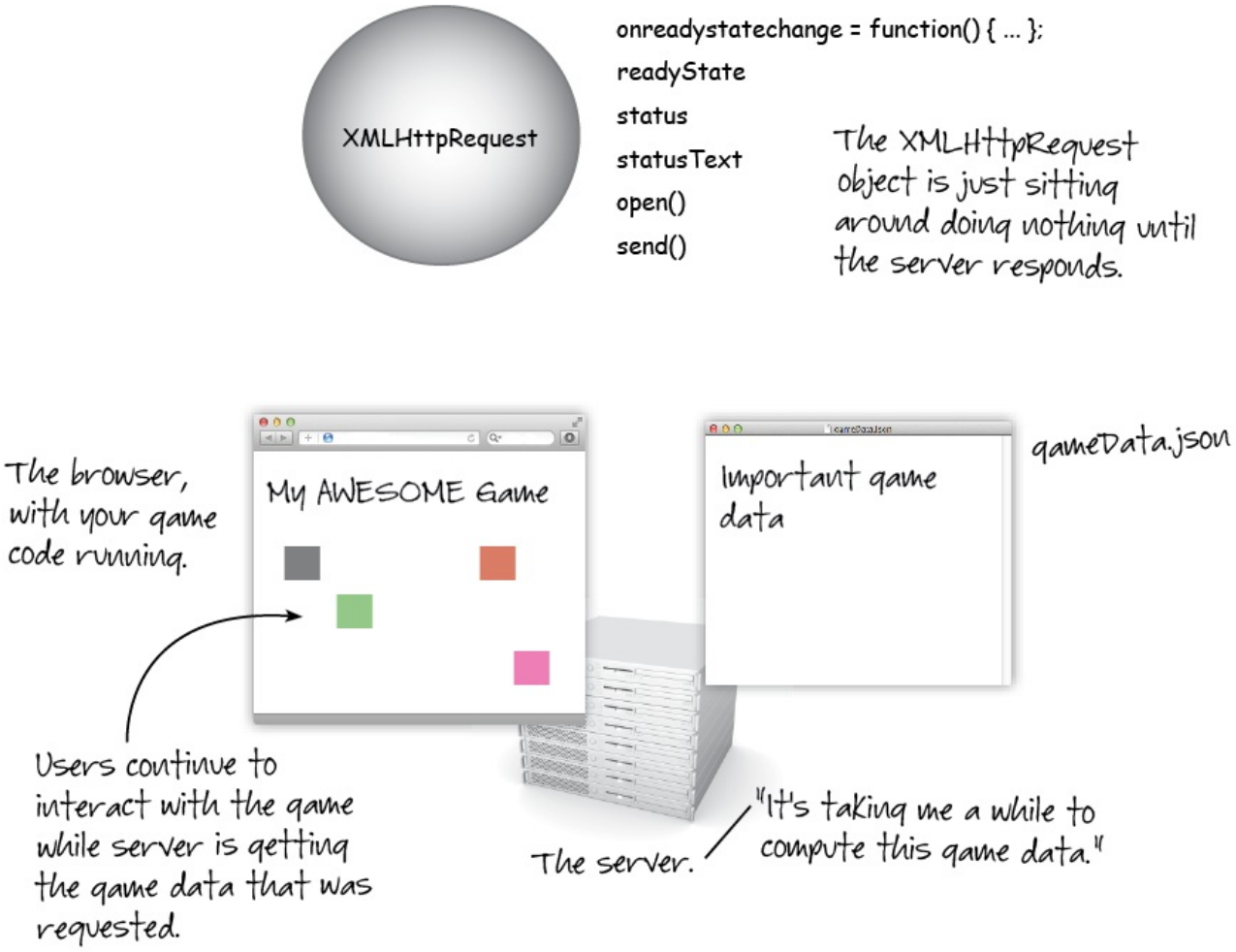
```
function getPetData() {
    var request = new XMLHttpRequest();
    request.open("GET", "pets.json");
    request.onreadystatechange = function() {
       ...
    };
    request.send();
}
```

In the function **getPetData()**, where we create the **XMLHttpRequest** object and use it to request data from the web server and get the response, we set up the **callback function** to handle the response. But we're not actually *calling* this function, in fact, we rely on the browser to call that function for us when the request object has received a response. We set up the function, and then **send** the request. After that, the **getPetData()** function ends and any other code we might run after calling **getPetData()** continues to run.

More importantly, the browser is free to respond to user interaction while we're waiting for the response to come back. So, let's say you have a page that has a lot of interactive items on it; you may have menus that expand and collapse when you move your mouse on and off of them. If your code uses XMLHttpRequest to go fetch some data, while it's off fetching that data, you can still interact with the page. You can hover your mouse over a menu and watch it expand and then collapse when you move your mouse off the menu—meaning that the browser is running other JavaScript code while it's waiting for the response. The JavaScript engine in the browser isn't hung up waiting for the response to your request for data; the browser will get the response at some point and will call your function. In the meantime, your other code is free to continue to work as normal.

This asynchronous benefit of Ajax doesn't matter much in our little Pets example, but imagine you're building an interactive game, and using XMLHttpRequest to send and receive data. In a highly interactive application, like a game, it's especially important that the user be able to continue to interact with the page while the data is being transferred. That's where the capacity to send and receive data asynchronously really comes in handy.


XMLHttpRequest
onreadystatechange = function() { ... };
readyState
status
statusText
open()
send()
Your code calls send(), but then continues doing other things.
"I need some important game data. The browser will let me know when you're ready by calling my callback function."
The browser, with your game code running.
My AWESOME Game
Important game data
gameData.json
The server.



If Ajax *wasn't* asynchronous, whenever you made a request to get some data, all action would be suspended until the response came back. For small applications like Pets, and small amounts of data like pets.json, you probably wouldn't notice, but if you were playing a game, or sending a request to a web service that has to search a huge database for results, you'd *really* notice a hangup if the data transfer wasn't asynchronous.

You can *make* Ajax **synchronous** if you really want to—that means as soon as you use the **send()** method of the request object to send the request, all other work stops. There might be a time when you need synchronous behavior. If you do, add an extra parameter to the **open()** method, like this:

OBSERVE:

```
request.open("GET", "pets.json", false);
```

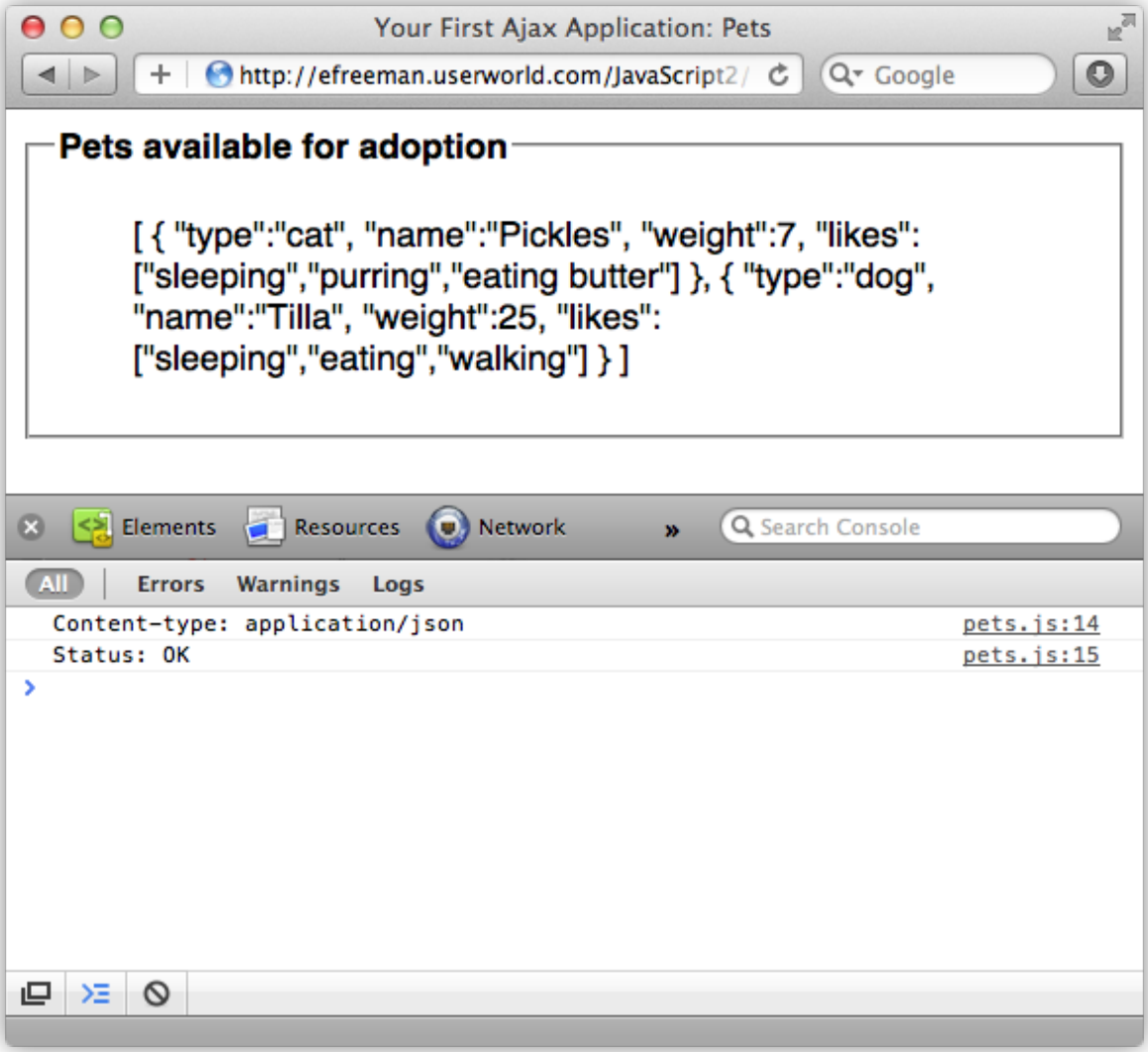
Content Types and Headers

There are two more pieces of information about the XMLHttpRequest object that come in handy, especially when you are debugging your code. They are the **content-type** of data in the response, and the text corresponding to the **status**. Update **pets.js** as shown:

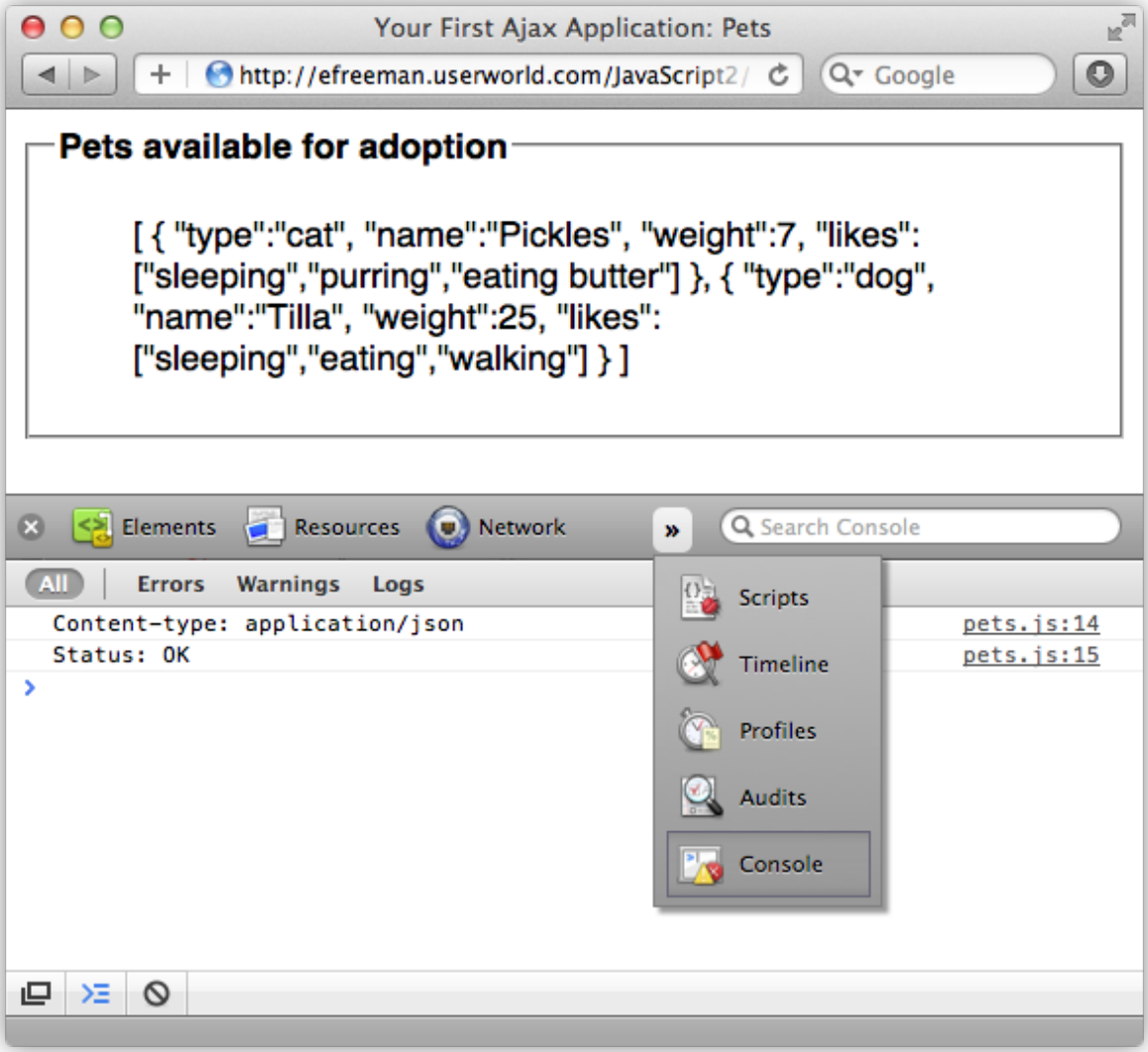
CODE TO TYPE:

```
window.onload = init;

function init() {
    getPetData();
}

function getPetData() {
    var request = new XMLHttpRequest();
    request.open("GET", "pets.json");
    request.onreadystatechange = function() {
        var div = document.getElementById("pets");
        if (this.readyState == this.DONE && this.status == 200) {
            var type = request.getResponseHeader("Content-Type");
            console.log("Content-type: " + type);
            console.log("Status: " + this.statusText);
            if (this.responseText != null) {
                div.innerHTML = this.responseText;
            }
            else {
                div.innerHTML = "Error: no data";
            }
        }
    };
    request.send();
}
```

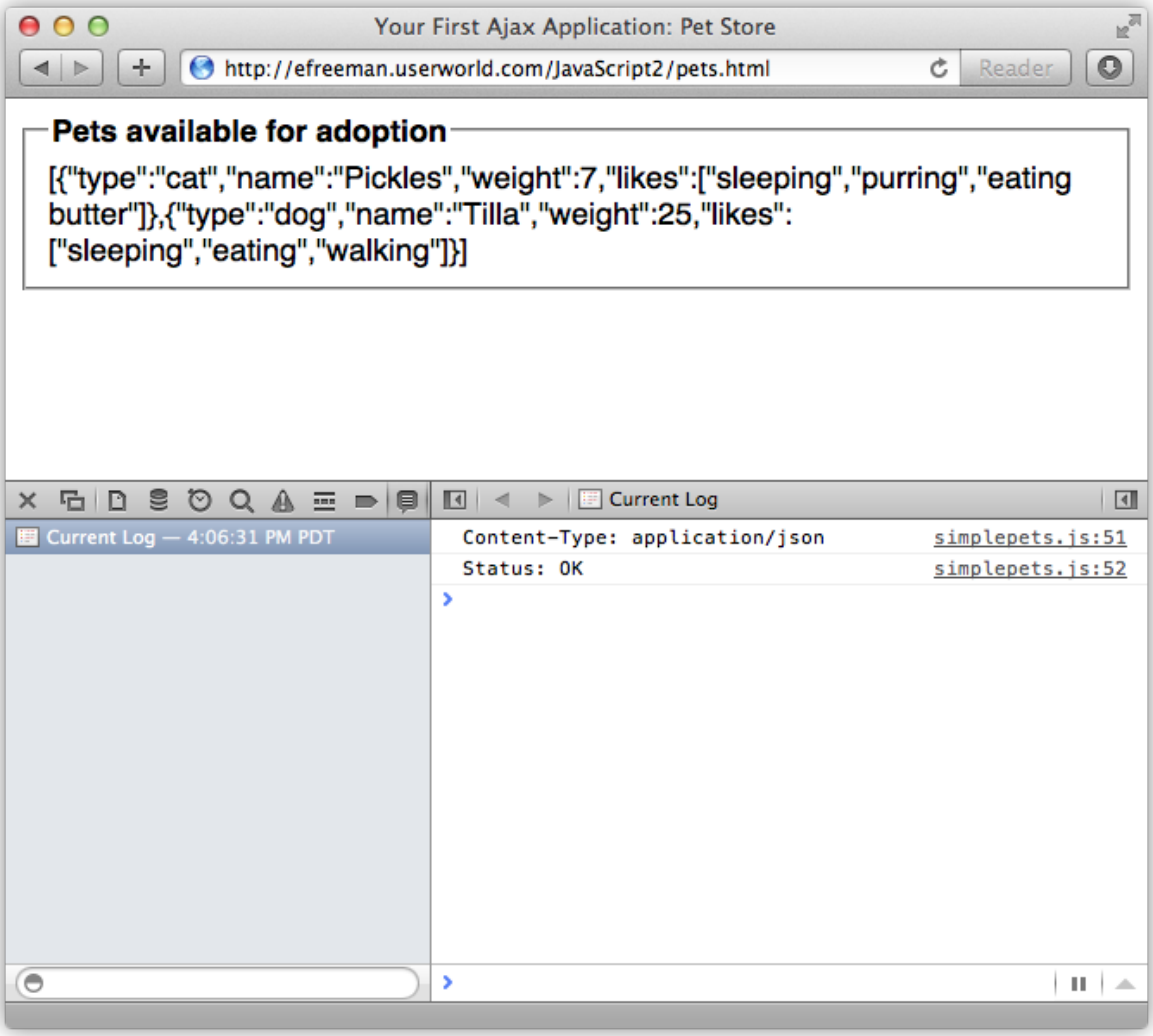
Save it, switch over to **pets.html** tab, and click Preview). You see both the content-type of the response and the status in your console window:



Make sure you choose the Console tab to view the results; if the window is narrow, you might have to use the drop-down menu to get to the console. For instance, in Safari 5 and Chrome, it will look like this:



In Safari 6, you access the console by going to the Log tab in the Web Inspector, and clicking on "Current Log":



The results show that the content type of the response data is **application/json**. The server knows it's serving us JSON data. Typically, you'll know what content type to expect as a response to a request, but if not, you can use the content type to determine how to handle the response. If you're expecting plain text, the content-type will be **text/plain**; for XML, the content-type will contain the letters "xml" (the full content-type can vary). If you are using XML, the data for the response will be in the property **responseXML**, *not* in responseText.

Note The JavaScript Console and Developer Tools in every browser tend to change with new browser versions, so your browser tools may look different.

Cross-Domain Security Restrictions

Before we leave the Pets example behind and move on to the To Do application, let's go over one last important aspect of Ajax.

You can only request data that is served from the domain from which the request is coming.

So, if you're running your web applications at the URL http://*yourusername*.oreillystudent.com/JavaScript2/pets.html, you can only requst data that is being served from the domain **oreillystudent.com**. The web application and the data it's requesting *must* be on the same domain, or the request will fail.

This is a security restriction built into browsers to keep malicious data out of your web application. In the pre-mashup world, this restriction was pretty helpful, but now that we have so many great web services around that serve up all kinds of interesting data that you could use to create the next million-dollar mashup, this restriction could be a hinderance.

Fortunately, there is a workaround: a **server-side script**, say, a PHP program, that goes and fetches the data you want from a *remote* web service, and then forwards it on to your web application. This server-side script, often called an

"intermediary" script, can verify the data you've requested (if necessary) before it ever gets to your web application. That way, you can make the request to the script (which *is* running on the same domain), the script requests the data from the web service, verifies the data, and then hands it back to your web application. This is transparent to your JavaScript Ajax code, but it does mean you'll need to write this intermediary program.

We covered a lot of complex Ajax details in this lesson. Take a break, let it all sink in a bit, and then do the project and the quiz. Then you'll be ready to tackle the To Do List application coming up next...

The To-Do List Application

Lesson Objectives

When you complete this lesson, you will be able to:

- create a JSON file that contains to-do items.
- create some HTML and CSS for this application.
- add content to your application.
- create and array of to-do objects.
- update your page with to-do items.

You've built an Ajax application that reads JSON data from an external file. The ability to retrieve data from an external source is one key feature of an Ajax application; a second feature that's just as important, is the ability to update your web page (or web *application*) dynamically with that data.

In this lesson, we'll begin building a To-Do List application. Eventually this application will let you both retrieve *and* save to-do list items using a web application that you've built. For this first part of the To-Do List application, we'll expand the Pets application and actually parse the JSON data, then update your web page using that data.

You'll recognize the code below. Feel free to reuse some of your project solution, or, if you'd prefer, you can start from scratch. We'll assume that you're starting over, so you can create fresh files using the instructions below.

Create the Files for the To-Do List Application

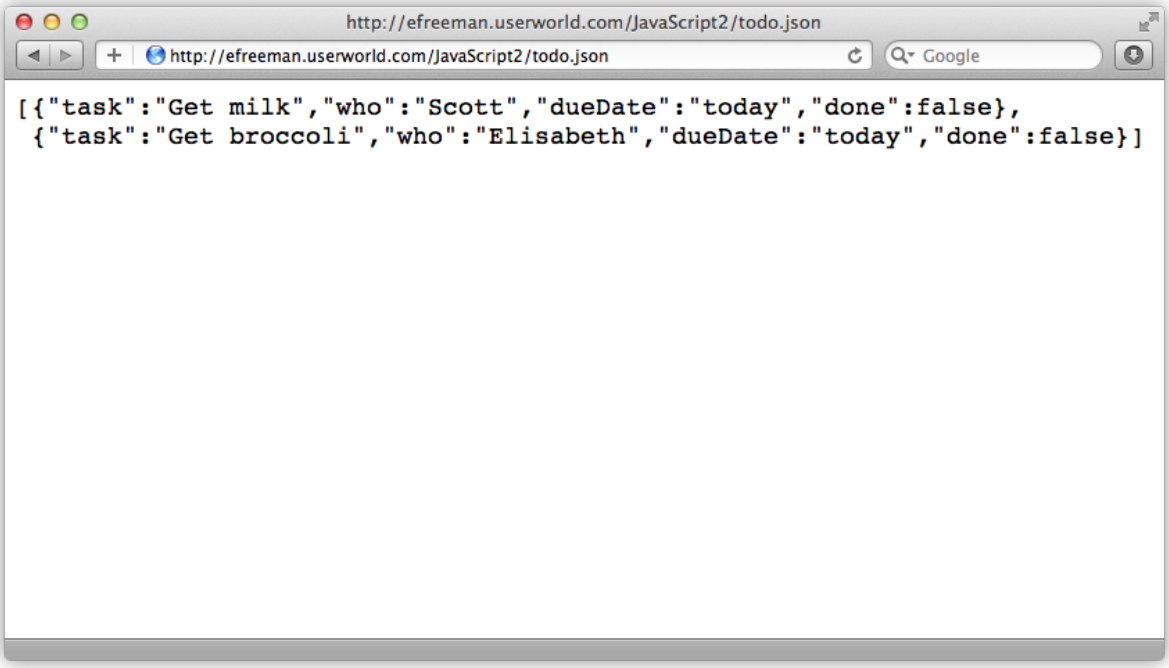
Creating the JSON Data File

To begin building the To-Do List application, you'll create a JSON file containing some to-do items. (You'll learn how to do that through the application itself.) Just like with the Pets example, you'll create a new file and write some data using the JSON format (or you can reuse the to-do list you created in the previous lesson's project):

CODE TO TYPE:

```
[{"task":"get milk","who":"Scott","dueDate":"today","done":false},
 {"task":"get broccoli","who":"Elisabeth","dueDate":"today","done":false}]
```

Save this file in your **/javascript2** folder as **todo.json** and, in HTML mode, click Preview. You see your to-do list JSON data in a web page, it looks like this:



Your web application will read in these items using Ajax. Notice that there are two to-do items: one for Scott and one for Elisabeth. Each item has four properties: **task**, **who**, **dueDate**, and **done**. All the property values are strings, except for **done**, which is a Boolean.

If you had to represent these to-do items as an object, what would that object look like? Give it some thought and we'll come back to that quesion shortly.

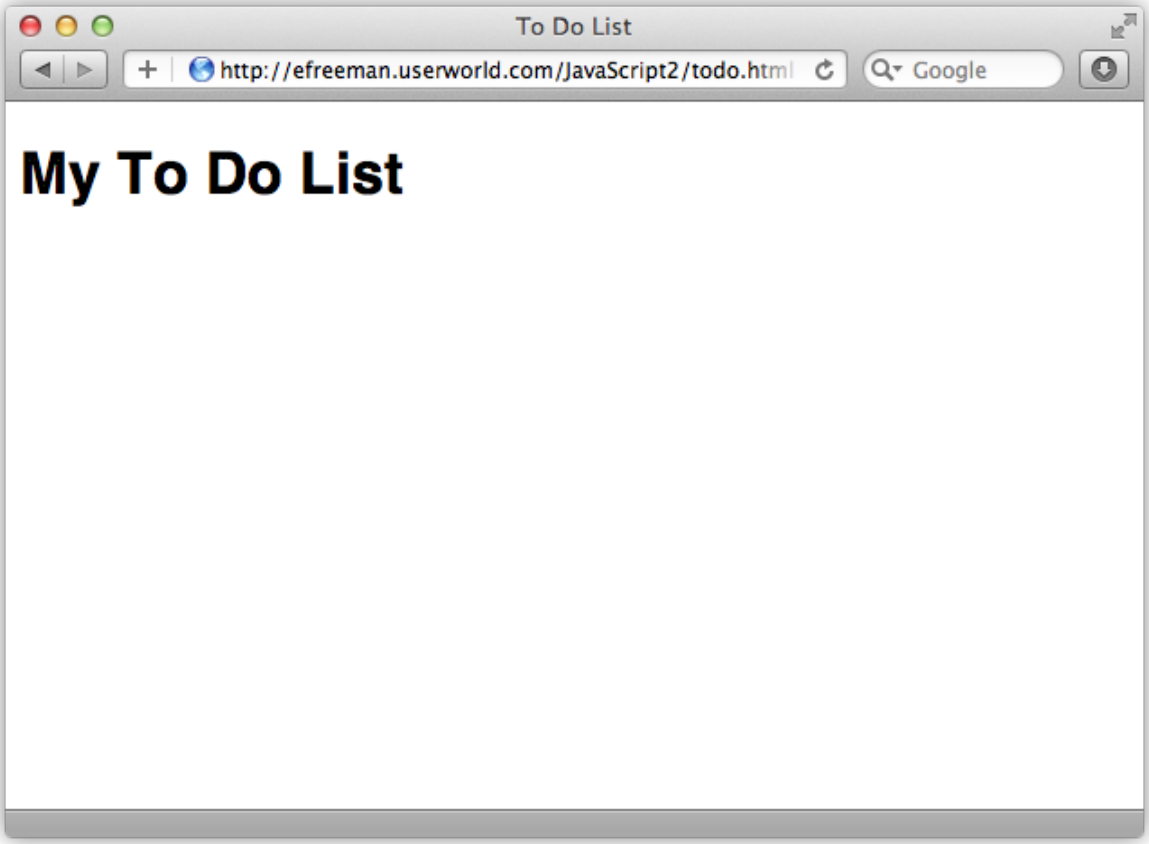
Creating the HTML and CSS

Next, let's create the HTML & CSS for this application. We'll keep it relatively simple, just like we did with pets; for now, all we need is a heading and a element. Create a new file and type in this HTML and CSS:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>To-Do List</title>
  <meta charset="utf-8">
  <script src="todo.js"></script>
  <style>
        body {
            font-family: Helvetica, Arial, sans-serif;
        }
  </style>
</head>
<body>
  <h1>My To-Do List</h1>
  <div id="todoList">
  </div>
</body>
</html>
```

Save this file in your **/javascript2** folder as **todo.html** and click Preview. You see this web page:



There's nothing in the web page, because we haven't added any content to it yet.

Adding the Content

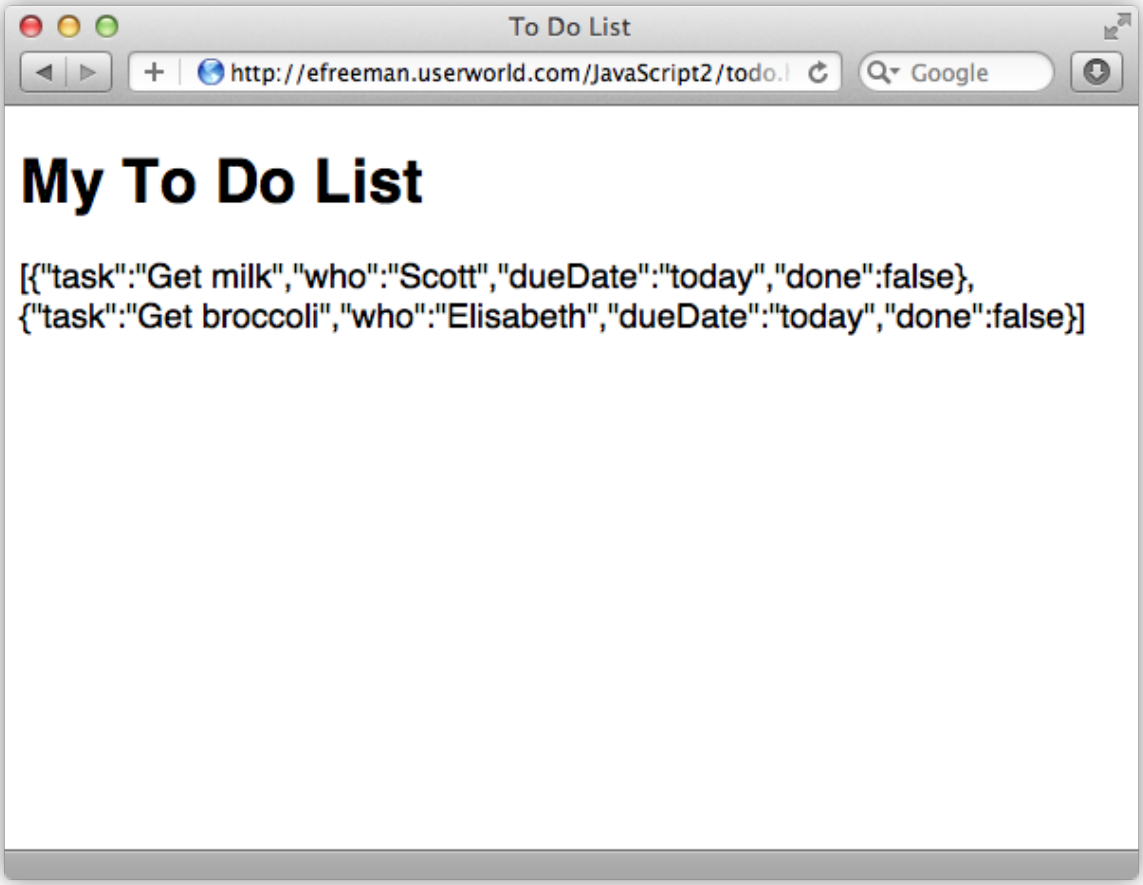
Okay, go ahead and create the JavaScript to read in the JSON from the "todo.json" file, just like we did with Pets:

CODE TO TYPE:

```
window.onload = init;

function init() {
    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        var listDiv = document.getElementById("todoList");
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                listDiv.innerHTML = this.responseText;
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}
```

Save it in your **/javascript2** folder as **todo.js**. Open your **todo.html** file (which links to this **todo.js** file

with the <script> element at the top), and click Preview. The JSON data from **todo.json** in your web page, looks like this:



So far, so good. It's the same process we applied to Pets, just revised for a To-Do List, and using slightly different JSON data. Here are the steps we take to get there:

- Call the function init() once the page finishes loading.
- Call getTodoData() from init(), which is where we use Ajax to retrieve the todo.json data.
- Create an XMLHttpRequest object to make the request for the data.
- Create an onreadystatechange handler for the request.
- Add the data to the page.

So now that you've got your hands on the JSON data, let's do something better than just spitting it out to the web page as is. What we'd really like to do is make a nice-looking list, right?

Create an array of To-Do Objects

To create a nice to-do list out of the JSON data, we need to *parse* the data using **JSON.parse()**. As we learned earlier, **JSON.parse()** parses, or *deserializes* JSON-formatted data and turns that data into a JavaScript object. In our case, that object will be a Todo object.

If you were to create a Todo object in JavaScript, you'd write something like this:

OBSERVE:

```
var todoObject = {
    task: "Get Milk",
    who: "Scott",
    dueDate: "today",
    done: false
};
```

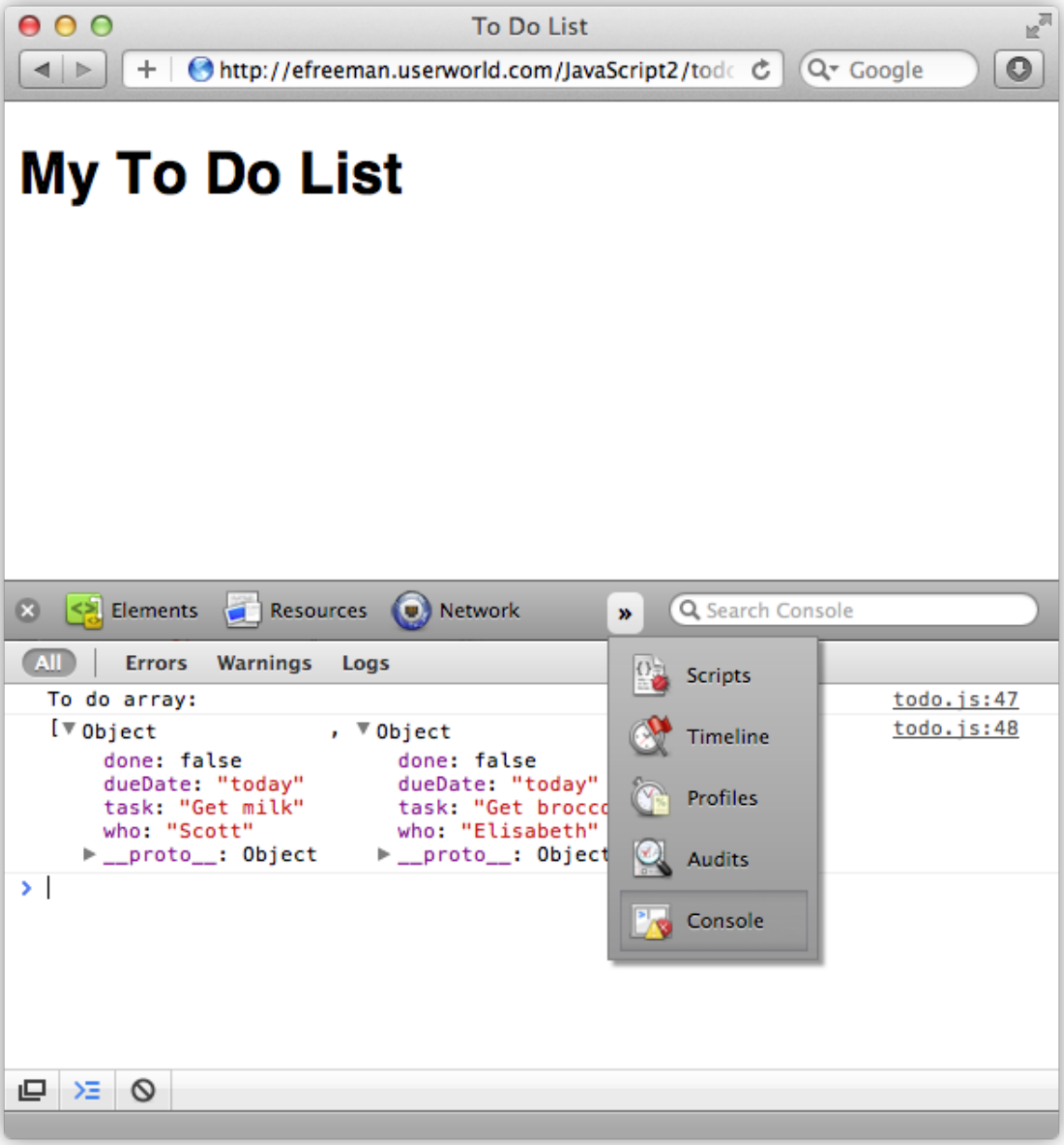
Remember, this is a *literal object*, that is, an object you typed in literally, rather than created using a constructor. Notice how closely this **todoObject** resembles the items in the **todo.json** file you created earlier.

When we read in the JSON data and turn the JSON to-do items into JavaScript objects, we need somewhere to put those objects; we'll stash them in an array. For this step, we'll take a look at that array in the console. After that, we'll actually use the array of objects to update the page. Update **todo.js** as shown:

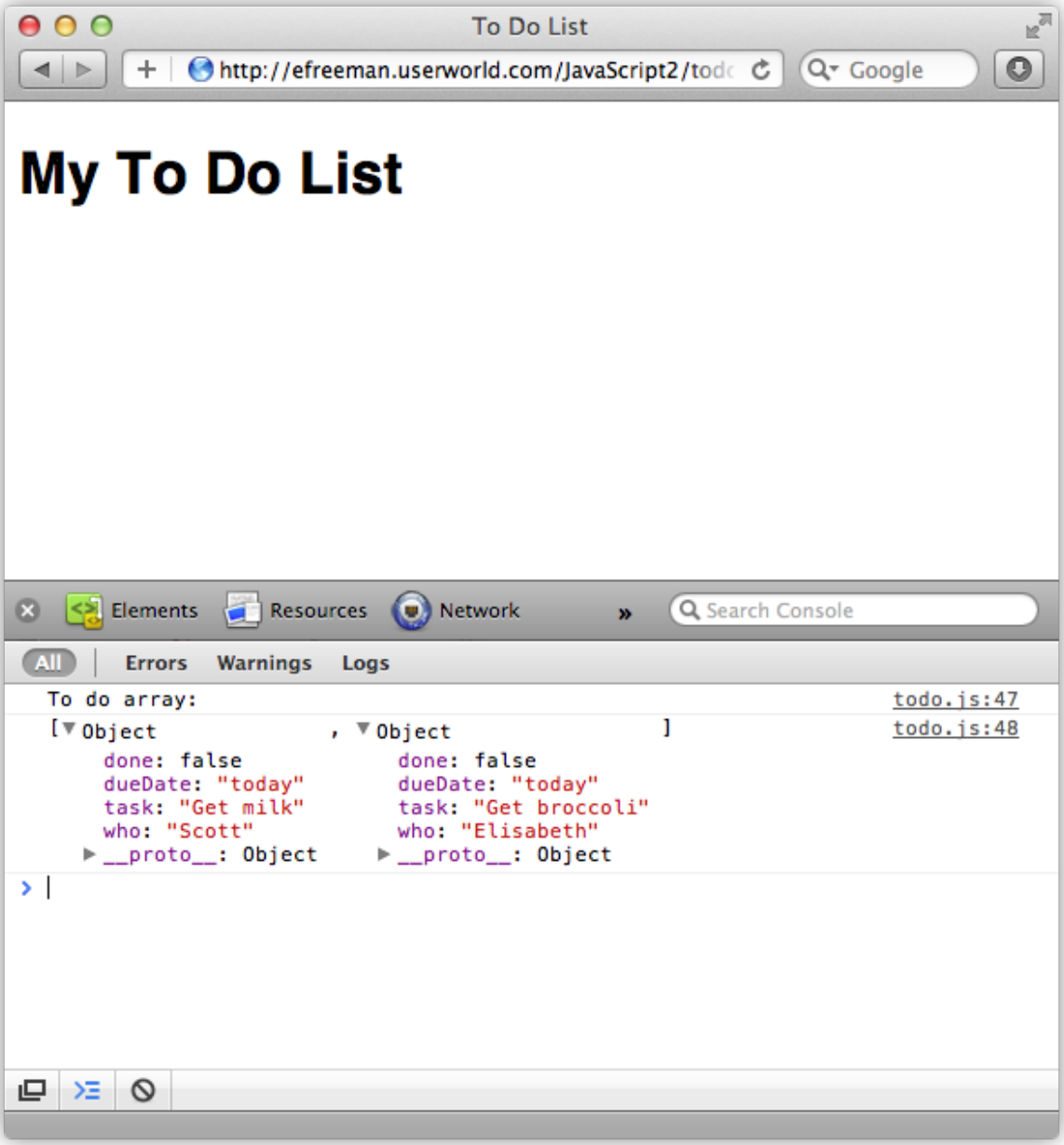
CODE TO TYPE:

```
var todos = new Array();

window.onload = init;

function init() {
    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        var listDiv = document.getElementById("todoList");
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                listDiv.innerHTML = this.responseText;
                parseTodoItems(this.responseText);
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
    console.log("To-do array: ");
    console.log(todos);
}
```

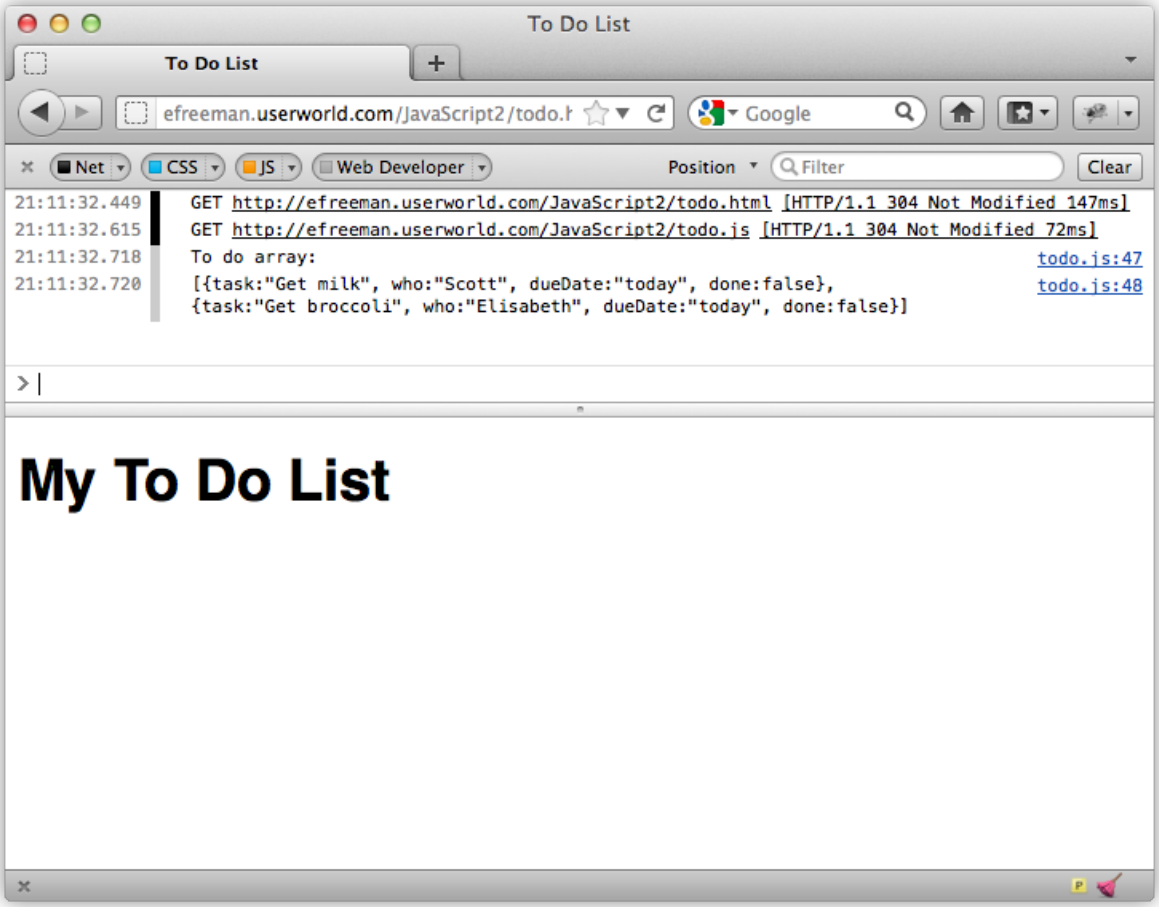
Save it, open your **todo.html** file, and click Preview. You won't see anything in the web page now, but you'll see some output in your developer console. Remember, if you're in Safari or Chrome, you might need to use the arrow in the Developer console window to access the console:



In the console, you see an array of your to-do objects, like this in Safari or Chrome:



...and like this in Firefox (using Firebug):



Let's walk through the new code. First, we add an empty array, **todos**, to hold all the objects that we create as we parse the JSON data from the **todo.json** file:

OBSERVE:

```
var todos = new Array();
```

Note that this is a global variable so we can access it from within any function.

Next, we update the **onreadystatechange** handler in the **getTodoData()** function to call the function **parseTodoItems()** and pass in the **responseText**, which holds the JSON data from the **todo.json** file:

OBSERVE:

```
parseTodoItems(this.responseText);
```

And of course, we implement the **parseTodoItems()** function:

OBSERVE:

```
function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
    console.log("To-do array: ");
    console.log(todos);
}
```

This function is where the magic happens: where we turn JSON from a file into real HTML objects that dynamically appear in your web page. Well, it won't seem like magic anymore, because you'll know how to do it.

In **parseTodoItems**, the JSON data is passed in as a variable named **todoJSON**. In the function we test to make sure **todoJSON** is not **null** or the **empty string**. If the file is empty (that is, there are no to-dos yet), then we have no JSON to parse and nothing to add to the page. If we *try* to parse an empty JSON string, we'll get an error message. We can avoid that by testing the string first.

We'll look at strings in more detail in a later lesson, but for now, we're using the **trim()** function (actually a String method!) on the string **todoJSON**. This gets rid of extra white space at the beginning or end of a string. So if you have a string like this:

OBSERVE:

```
var myString = "  There are a few empty spaces.   "
```

...with empty spaces at the begining and/or end, after calling **myString.trim()**, you'll have a string like this:

OBSERVE:

```
"There are a few empty spaces."
```

If the string is empty, then we just return from the function (that is, the rest of the function isn't executed).

Now let's review the next bit of code:

OBSERVE:

```
var todoArray = JSON.parse(todoJSON);
if (todoArray.length == 0) {
    console.log("Error: the to-do list array is empty!");
    return;
}
```

Once we know the string isn't empty, we can parse it using **JSON.parse()**. This turns the **todoJSON** string into an object. In our case, we know that we actually have an *array of objects* because that's how we designed the data in the todo.json file (take another look at the contents of the file and notice the [and] surrounding the rest of the data; that means array). So instead of putting the result of the call to **JSON.parse()** into an object variable, we are putting it into an array variable, the **todoArray**.

So, why don't we just put the result of the **JSON.parse()** into the **todos** array? That's a great question! And the answer is: because later on, we're going to add a form to the page to let you add new to-do items to your list. We want to keep the to-dos that are in the *file* separate from the to-dos managed by the *page*, so we have this temporary array, **todoArray**, which we use to get the to-do items from the file. Later we'll add them to the **todos** array.

Before we try to add items from the **todoArray** to the **todos** array, we need to make sure there's something in it. What if you made an *empty* array (Like this: [ ]) in your todo.json file? That would be a valid JSON string, but it wouldn't

contain any objects. We need to check for that, and return from the function if the array is empty.

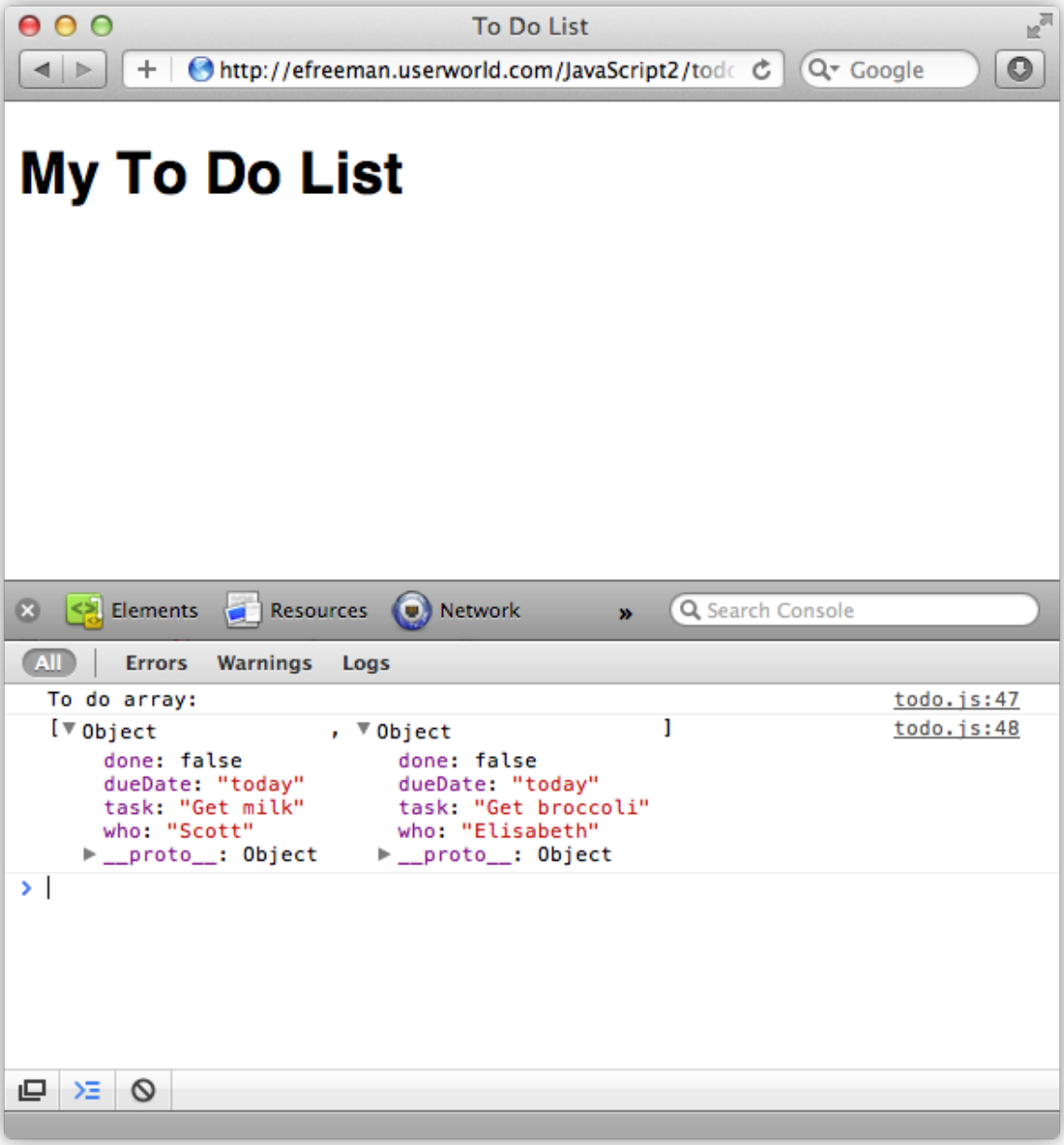
Okay, at this point we have an array with at least one object in it. Here's the next (and final) piece of code from the **parseTodoItems** function:

OBSERVE:

```
for (var i = 0; i < todoArray.length; i++) {
    var todoItem = todoArray[i];
    todos.push(todoItem);
}
console.log("To-do array: ");
console.log(todos);
```

Now, we want to put the objects in the **todoArray** into the **todos** array. We'll iterate (loop) through all the items in the **todoArray** and copy them into the **todos** array. First we store the item from the **todoArray[i]** in the variable **todoItem**, and then use the Array method **push()** to add the **todoItem** onto the end of the **todos** array. (We don't actually need the intermediate **todoItem** variable; can you see how you'd do this without it?)

Finally, to make sure the **todos** array has the right stuff in it, we use **console.log()** to display the array in the console. Here's how it looks in Safari and Chrome:



Update the Page with To-Do Items

The next step is to get the items from the **todos** array into your web page. We're creating a to-do *list*, so let's use an HTML list to put the items in. First, change **todo.html** so that the "todoList" element is a **<ul>** instead of a ** **, like this:

CODE TO TYPE:

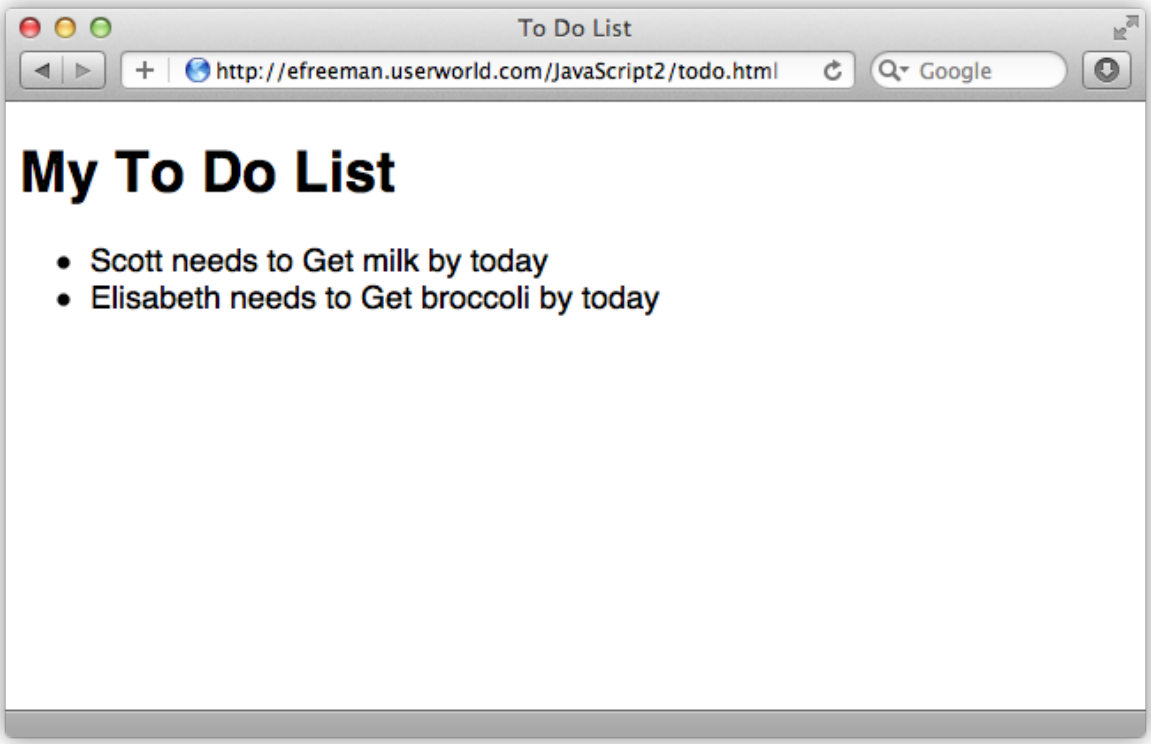
```
<!doctype html>
<html>
<head>
<title>To-Do List</title>
  <meta charset="utf-8">
  <script src="todo.js"></script>
  <style>
        body {
            font-family: Helvetica, Arial, sans-serif;
        }
  </style>
</head>
<body>
  <h1>My To-Do List</h1>
  <div id="todoList"><ul id="todoList">
  </div></ul>
</body>
</html>
```

Of course, the "todoList" list is empty (it has no <li> elements in it) because we're going to use JavaScript to add <li> elements created using the objects stored in the **todos** array. Update **todo.js** as shown:

CODE TO TYPE:

```
var todos = new Array();

window.onload = init;

function init() {
    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                parseTodoItems(this.responseText);
                addTodosToPage();
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
    console.log("To-do array: ");
    console.log(todos);
}

function addTodosToPage() {
    var ul = document.getElementById("todoList");
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = document.createElement("li");
        li.innerHTML =
            todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
        ul.appendChild(li);
    }
}
```

Save it, open your **todo.html** file, and click Preview. You see the to-do items in your **todo.json** file displayed as list items in your web page, like this:



Try adding another to-do item to your **todo.json** file and reload the page. Do you see it?

So how did we add the array items to the page? Let's review:

OBSERVE:

```
if (this.responseText) {
    parseTodoItems(this.responseText);
    addTodosToPage();
}
```

In the **onreadystatechange** handler in **getTodoData()**, in addition to calling **parseTodoItems()**, we're calling **addTodosToPage()**. This gets called only after all the JSON has been processed and all the to-do items have been added to the **todos** array, so we know all the data will be in the array at this point.

This **addTodosToPage** function actually adds the items to the page by adding them to the "todoList" :

OBSERVE:

```
function addTodosToPage() {
    var ul = document.getElementById("todoList");
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = document.createElement("li");
        li.innerHTML =
            todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
        ul.appendChild(li);
    }
}
```

In **addTodosToPage()**, first we **get the ** element with the id "todoList" from the page, so we can add a new list item to it. Then we **loop through all the objects in the todos array** and create a temporary variable **todoItem** for each item in the array. Then we **create a new <li> element** element for each item using the **document.createElement()** method, and set the HTML contents of that element to a string with data from the **todoItem** object. The **todoItem** is an *object* that was created when we parsed the JSON in the todo.json file using **JSON.parse()**, an object that looks something like this:

OBSERVE:

```
var todoObject = {
    task: "Get Milk",
    who: "Scott",
    dueDate: "today",
    done: false
};
```

You can access each property in the object as you normally would, using dot notation. The string we create to add to the list item uses the properties of the **todoItem**, like **todoItem.who**, **todoItem.task**, and **todoItem.dueDate**.

Once we've set the content of the **<li>** element, we can **add it to the "todoList" element** using the **appendChild()** method.

As soon as you add the element to the page with **appendChild()**, the page updates to reflect the new element and you see your to-do list item.

Did you notice that we're not passing the **todos** array into the **addTodosToPage** function? That's because it's a global variable, and we don't need to—we can access it from any function. We made the **todos** array a global variable so we could access it in both the **parseTodoItems()** and **addTodosToPage()** functions.

In this lesson, you've brought together several things you already knew about— JSON, objects, arrays, using XMLHttpRequest to get data from a file, and updating the page with new elements—to create an Ajax web application that loads to-do items from a file, todo.json, and updates a page with those to-do items. All these parts work together in your web application to do something that's really kinda cool!

Saving Data with Ajax

Lesson Objectives

When you complete this lesson, you will be able to:

- add a form to the HTML page that lets you enter new To-Do items and save them in the JSON file.
- style your forms.
- validate your forms.
- use Ajax to write data to a file.

Adding and Saving a To-Do List Item

We're off to a great start with our To-Do List application; we've got a simple Ajax web application that reads To Do data from a JSON file, and updates a page dynamically with that data.

In this lesson, we continue building the application. We'll add a form to the HTML page that lets you enter new To-Do items and save them in the JSON file, so that the next time you load the page, your new items will be there. We'll do some form validation too, which is always good practice (and will be good review of JavaScript techniques). You've used Ajax to *read* data from a file; in this lesson, you'll see how to use Ajax to *write* data to a file (with a little help from a PHP script).

So, let's get to it!

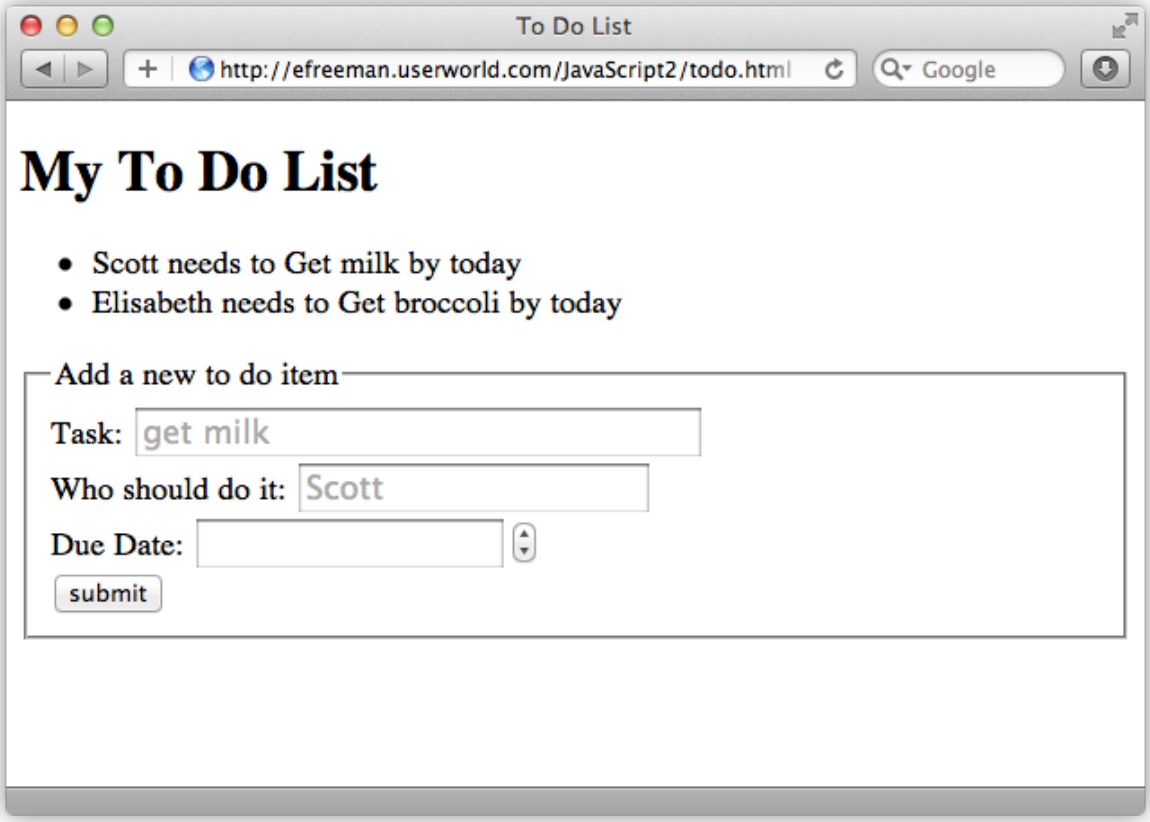
Add a Form to Your Page and Style It

To add new items to our to-do list using a web page, we need a form. Add a form to **todo.html** as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>To-Do List</title>
  <meta charset="utf-8">
  <script src="todo.js"></script>
  <style>
        body {
            font-family: Helvetica, Arial, sans-serif;
        }
  </style>
  <link rel="stylesheet" href="todo.css">
</head>
<body>
  <h1>My To-Do List</h1>
  <ul id="todoList">
  </ul>

  <form>
  <fieldset>
    <legend>Add a new to-do item</legend>
    <div class="tableContainer">
      <div class="tableRow">
        <label for="task">Task: </label>
            <input type="text" id="task" size="35" placeholder="get milk">
      </div>
      <div class="tableRow">
        <label for="who">Who should do it: </label>
            <input type="text" id="who" placeholder="Scott">
      </div>
      <div class="tableRow">
        <label for="dueDate">Due Date: </label>
            <input type="date" id="dueDate">
      </div>
      <div class="tableRow">
        <label for="submit"></label>
        <input type="button" id="submit" value="submit">
      </div>
    </div>
  </fieldset>
  </form>
</body>
</html>
```

Save it and click Preview. You see the web page, which looks like this:



When you preview, the application loads the to-do items you already have in the **todo.json** file, using the code you added in **todo.js** in the previous lesson (your list may look a little different from mine if you've added or changed items).

In this version of **todo.html**, we added a form with three data inputs and a submit button. Two of the data inputs are for text—the task, and who should do it—and the third is for a date. In many browsers, the date field will look like a text input, but in some browsers you may get a popup date picker.

We've added some extra s for styling the table; you'll see how this works in the next step.

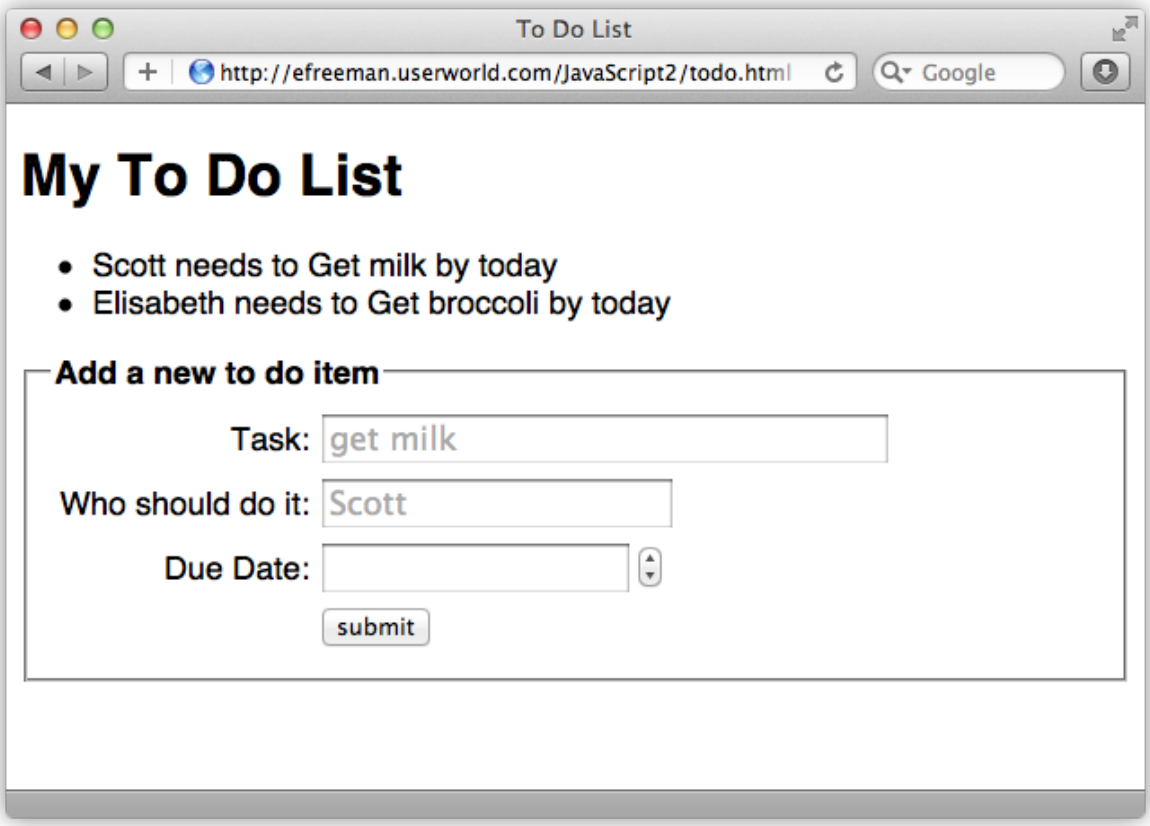
Style the Form

That HTML doesn't look too bad, but let's make it look even better with a little CSS. We removed the CSS we had previously, and added a link to a file. Go ahead and create a new file, and we'll add the extra CSS we need to style the form. Create a new file and type in this CSS:

CODE TO TYPE:

```
body {
    font-family: Helvetica, Arial, sans-serif;
}
legend {
    font-weight: bold;
}
div.tableContainer {
    display: table;
    border-spacing: 5px;
}
div.tableRow {
    display: table-row;
}
div.tableRow label {
    display: table-cell;
    text-align: right;
}
div.tableRow input {
    display: table-cell;
}
```

Save this file in your **/javascript2** folder as **todo.css**. Open **todo.html**, which links to this **todo.css** file at the top (with the <link> element), and click Preview. Your page now looks like this:



The fields are aligned properly in this form which gives it a more professional appearance than the previous one.

This CSS uses *table display* properties to lay out the s and form controls, similar to the way an HTML table is laid out. If you're not familiar with table display, don't worry about it; it's just a way of laying out a page. In fact, CSS table display properties are the same properties that are used by default in the browser to lay out HTML tables. Using table display works really well for forms, because forms often have a regular, table-like structure to them.

Now that you've got the web page ready to go for entering new to-do items, it's time to update your JavaScript so that you can do something with the items you enter.

Process the Form with JavaScript

You can enter data in the form in the web page you created in the previous step, but nothing happens when you click **submit**, because we have no **action** in the <form> element, and no JavaScript to capture a click on the submit button. So, as it is, the form does nothing.

In this next step, we need to get the data from the form and check it. We want to get the form data using our own JavaScript code so we can validate the data and then do something with it. So, we'll need a way to determine when the submit button is clicked so we can get the data that was entered into the form. We'll add a **click handler** function to the Submit button.

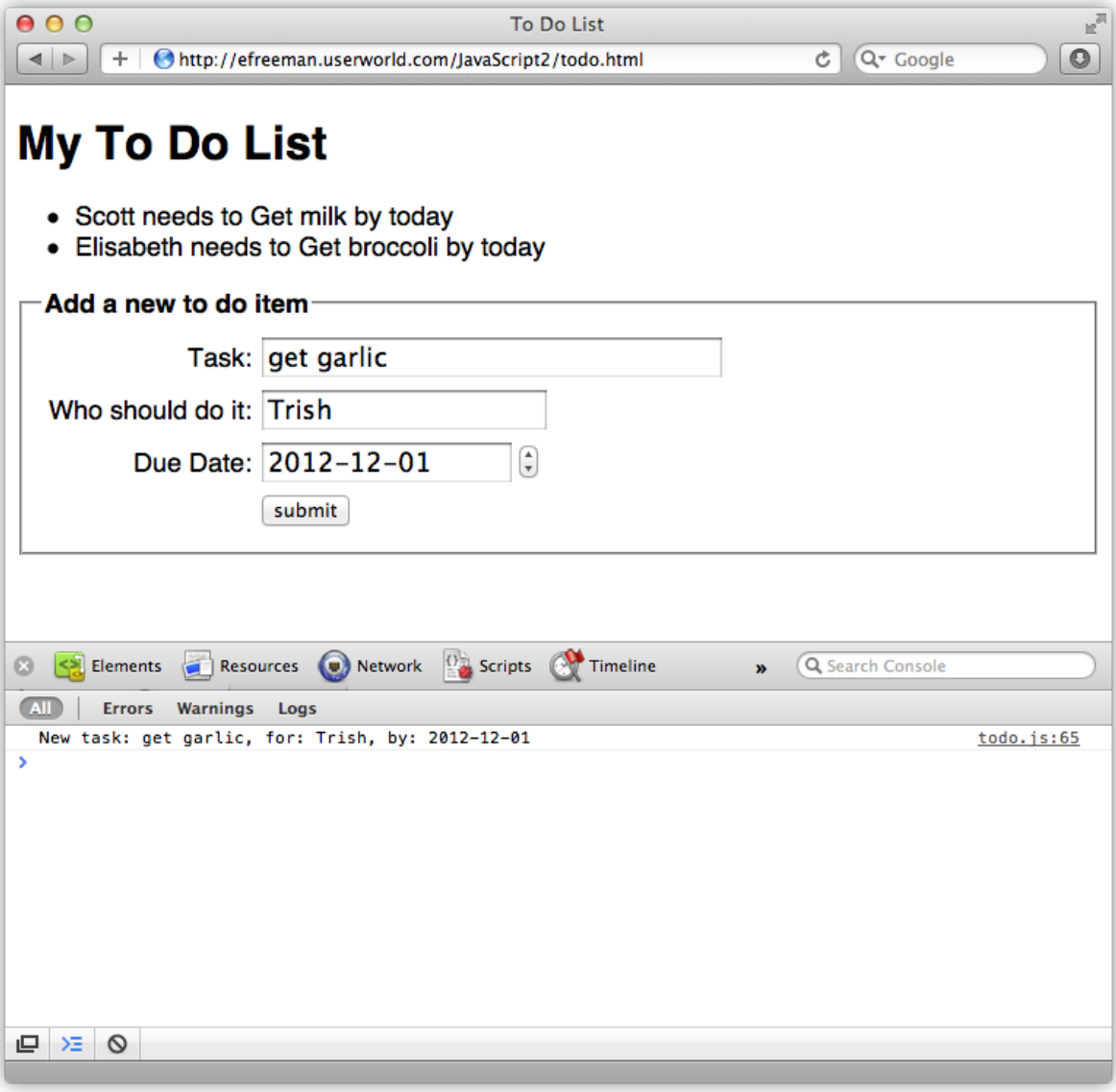
Update **todo.js** as shown:

CODE TO TYPE:

```
var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;
    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                parseTodoItems(this.responseText);
                addTodosToPage();
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
}

function addTodosToPage() {
    var ul = document.getElementById("todoList");
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = document.createElement("li");
        li.innerHTML =
            todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueD
ate;
        ul.appendChild(li);
    }
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    console.log("New task: " + task + ", for: " + who + ", by: " + date);
}
```

```
function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}
```

Save it, open **todo.html**, and click Preview. Open the developer console so you can see the console message we're using to display the data. Then, type some data into the form. After entering the data and clicking **submit**, you see the console message showing the data you entered. It looks like this:



We get the data from the form with the function **getFormData()**, which we set up as the **click handler** for the **submit button**:

OBSERVE:

```
var submitButton = document.getElementById("submit");
submitButton.onclick = getFormData;
```

In **getFormData()**, we check to make sure you enter values for the various fields:

OBSERVE:

```
function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    console.log("New task: " + task + ", for: " + who + ", by: " + date);
}
```

We get the value of each field in the form using **document.getElementById()** to first get the form control element, and then using the element's **value()** method to get the value of the control. In our form, each control will be a string. If any of the fields is empty, the function just **return**s, without displaying anything in the console. Notice that we're using a helper function, **checkInputText()** to make sure that each form control has a value:

OBSERVE:

```
function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}
```

checkInputText() has two parameters: **value** and **msg**. The **value** is the string data that you enter into the form control, and the **msg** is the text to display in the alert if you didn't enter anything for that control. We check to make sure **value** isn't empty; if it is, we display the **msg** and return true. If it is **not** empty, we return false. Look back at **getFormData()** and you'll see that if we return false from **checkInputText()** for each form field, then we **display the console message** showing the data that you entered into the form.

Create a New To-Do Object

It's good to see the form data we enter in the console, but we really want to see it in the page. We also need to add the new to-do item to our **todos** array, which means we need to create a to-do object. You probably remember from the previous lesson that when we read in the to-dos from the file **todo.json**, each to-do item stored in in the **todos** array is an **object**.

How can we create a to-do object for the data that you've entered in the form? We need a to-do object constructor. Do you remember how to create a constructor function? Modify **todo.js** as shown:

CODE TO TYPE:

```
function Todo(task, who, dueDate) {
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;
    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                parseTodoItems(this.responseText);
                addTodosToPage();
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
}

function addTodosToPage() {
    var ul = document.getElementById("todoList");
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = document.createElement("li");
        li.innerHTML =
            todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueD
ate;
        ul.appendChild(li);
    }
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
```

```
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    console.log("New task: " + task + ", for: " + who + ", by: " + date);
    var todoItem = new Todo(task, who, date);
    todos.push(todoItem);
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}
```

Here, we use the **Todo()** constructor function to create a new **Todo** object, and push that object onto the end of the **todos** array. The **Todo()** constructor takes three arguments: **task**, **who**, and **dueDate**. The constructor function sets the object properties to the values that are passed into it. In **getFormData()**, we pass in the values that we got from the form for those arguments. In addition, the **Todo()** constructor sets the **done** property to **false**. Can you guess why?

We'll go back to the **done** property in a later lesson, but for now assume that all the to-do items are still to be done!

Adding the New To-Do Item to the Page

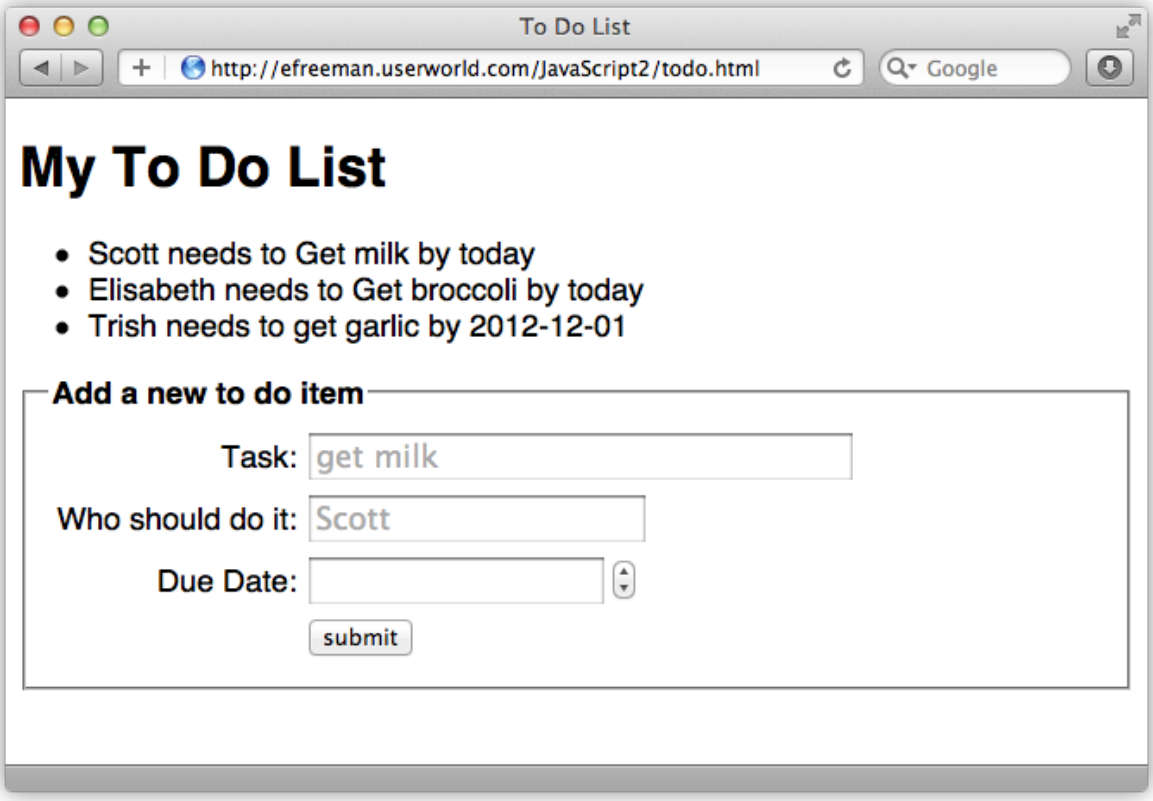
Now we'll add the to-do item to the page so we can see the result of all of our hard work in the web page, rather than just in the console. Modify **todo.js** as shown:

CODE TO TYPE:

```
function Todo(task, who, dueDate) {
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;
    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                parseTodoItems(this.responseText);
                addTodosToPage();
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
}

function addTodosToPage() {
    var ul = document.getElementById("todoList");
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = document.createElement("li");
        li.innerHTML =
            todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueD
ate;
        ul.appendChild(li);
    }
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
```

```
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    console.log("New task: " + task + ", for: " + who + ", by: " + date);
    var todoItem = new Todo(task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}

function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = document.createElement("li");
    li.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
    ul.appendChild(li);
    document.forms[0].reset();
}
```

Save it, open **todo.html** and click Preview. Enter the information for a to-do item in the form and click **submit**, and your new item appears in the page:



To add the to-do item to the page, we use a new function, **addTodoToPage()**, and call it just after we add the new **Todo** object to the **todos** array in the function **getFormData()**.

OBSERVE:

```
function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = document.createElement("li");
    li.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
    ul.appendChild(li);
    document.forms[0].reset();
}
```

The Todo object that's passed into **addTodoToPage()** gets the parameter name **todoItem**. To add it to the page, first we get the "todoList" ; then we create a new <li> element to add to this list, and set the innerHTML of the <li> to the information in the **todoItem** object. Then we add the new list item to the "todoList" list. Finally, we reset the form to make it easier to create another to-do item using the form.

Don't mix up the two functions: **addTodoToPage()**, which we just created to add a single to-do item entered using the form to the page, and the function we added earlier (in the previous lesson), **addTodosToPage()**, which adds all the to-do items in the JSON file, **todo.json**, to the page when the page first loads. The two functions look similar (and have similar names), but have slightly different purposes. Can you see why we need both? Can you think of how we might make our code more efficient by combining some of the functionality of the two functions? We'll come back to these and other pressing questions in the next lesson!

The Case of the Disappearing To-Do Item

Try adding a couple more to-do items to your page. Now, reload the page. What happens? Hmm. That's interesting.

Saving the Data with Ajax

Okay, we've got the new to-do item to display in the to-do list on the page, but how do we save it in the **todo.json** file so that it doesn't disappear when you reload the page?

We can use Ajax to send the data in the **todos** array (where all of our todos are stored) to a PHP script stored on the server (the same server where your **todo.json** file is stored, in our case, the O'Reilly School of Technology server) that will write the to-do items into the **todo.json** file.

You might be wondering, "Why can't we write the to-do items directly to the file using JavaScript?" Currently, JavaScript does not allow you to read from or write to files directly on a filesystem, for security purposes. (Imagine if you went to a malicious website that could read or write from your hard drive. Bad news!) That's why we need the help of a server script.

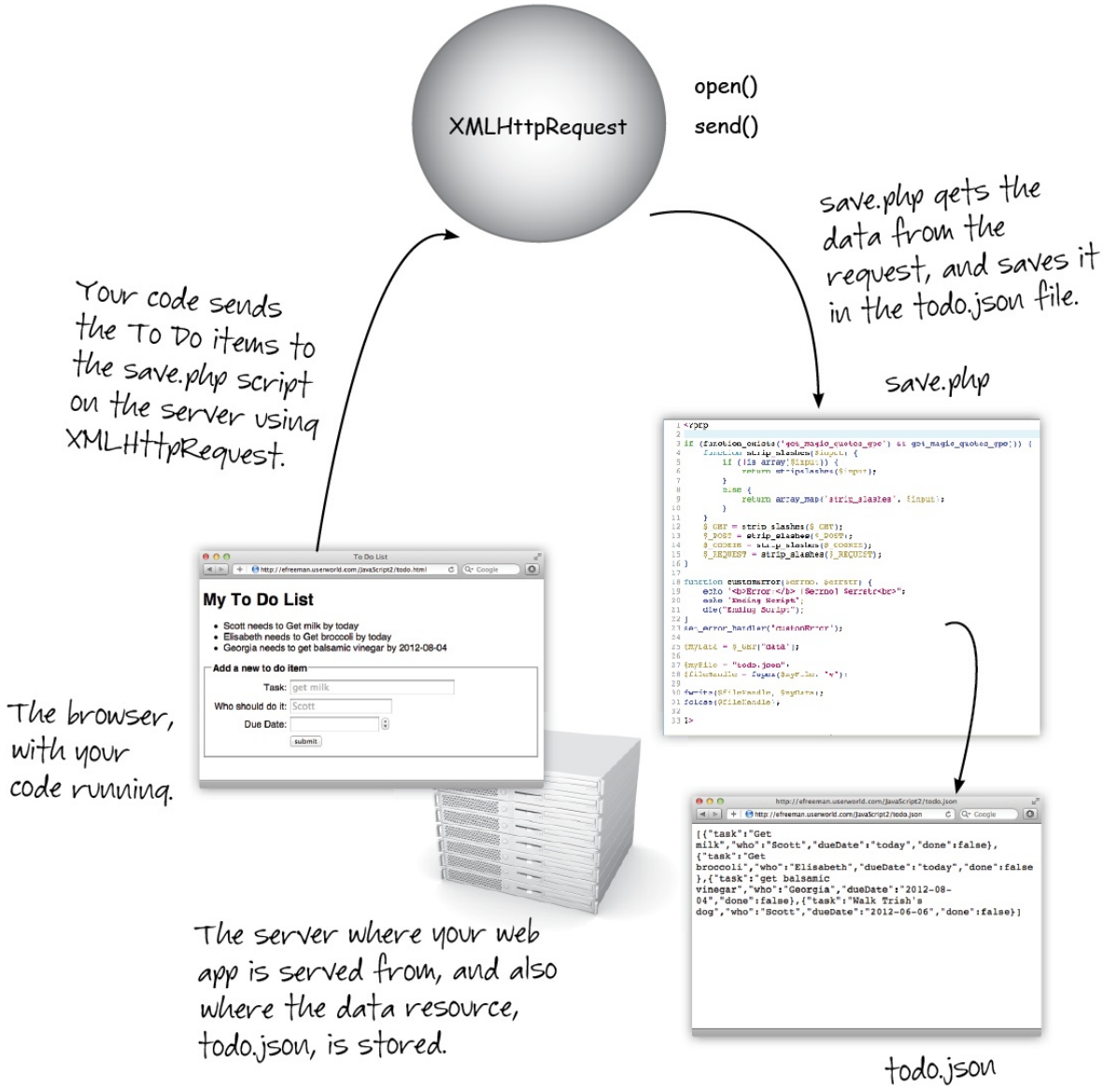
Note Work is underway on a file system library for JavaScript that will allow you to read and write files on your local file system (your own computer, not the file system on the server!), but this is a new project still in development as of this writing.

So you may think that we read directly from the **todo.json** file. Actually we don't. We use an **HTTP request** to get the data, and the web server reads the data from the file and returns it to the browser, which places the data in the **XMLHttpRequest** object, and our JavaScript code gets it from there. Whew!

We need to do something similar, only in reverse to write the data. We'll use the **XMLHttpRequest** object to send some data back to the web server. But we need a program on the web server that knows what to do with the data. We can't just randomly send data to the web server; we need to send it to a specific program—the server script—that knows how to update your **todo.json** file.

It's really similar to what happens when you submit a form that has a server specified in the **action** attribute, but instead of using the submit action with the form, we'll send the data to the server using the **XMLHttpRequest** object.

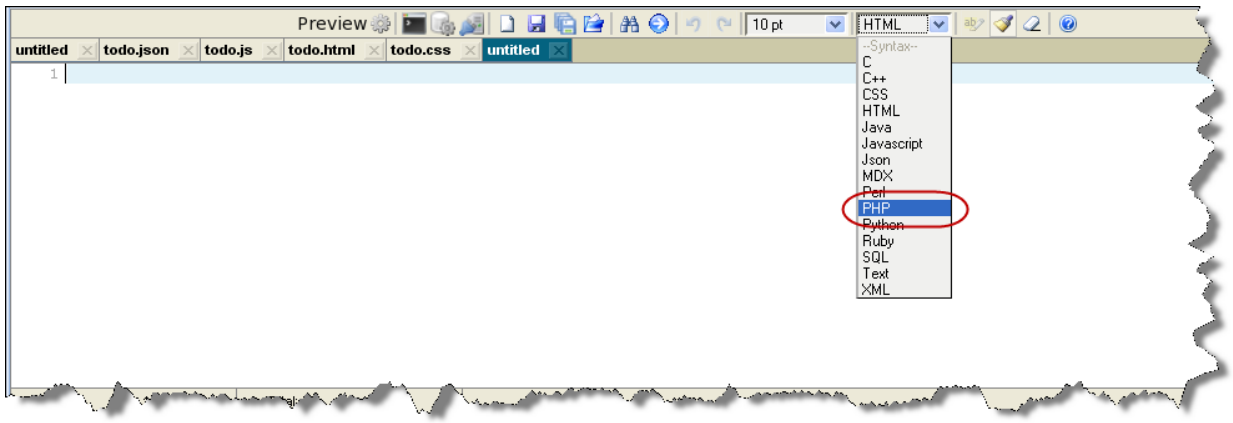
Here's how it works:



Creating the PHP Server Script

First, we need to create the script we're using to save the to-do items we send it in the **todo.json** file.

You can switch into PHP mode to add the PHP file if you like; just use the pull-down menu to use PHP instead of HTML:



CODE TO TYPE:

```
<?php

if (function_exists('get_magic_quotes_gpc') && get_magic_quotes_gpc()) {
    function strip_slashes($input) {
        if (!is_array($input)) {
            return stripslashes($input);
        }
        else {
            return array_map('strip_slashes', $input);
        }
    }
    $_GET = strip_slashes($_GET);
    $_POST = strip_slashes($_POST);
    $_COOKIE = strip_slashes($_COOKIE);
    $_REQUEST = strip_slashes($_REQUEST);
}

function customError($errno, $errstr) {
    echo "<b>Error:</b> [$errno] $errstr<br>";
    echo "Ending Script";
    die("Ending Script");
}
set_error_handler("customError");

$myData = $_GET["data"];

$myFile = "todo.json";
$fileHandle = fopen($myFile, "w");

fwrite($fileHandle, $myData);
fclose($fileHandle);

?>
```

 Save it in your **/javascript2** folder as **save.php**.

Don't worry, for this course, we don't expect you to know PHP (although, if you want to learn, there's a great course on PHP and how to write web pages that use PHP scripts: Introduction to PHP. For information about this and all O'Reilly School of Technology courses, contact info@oreillyschool.com.

Anyway, since you know JavaScript, you can probably read through this PHP script and understand some of it. The first part of the file is responsible for stripping those extra slashes that get added to your string when you send a string that contains quotation marks. The next part is an error handling function (**customError**), and below that is where the script actually gets the data from the request and writes it to the file. Again, don't worry about these details, just be aware that this script writes all the to-do items you send it to the file **todo.json** (overwriting what was there before, which will be important in just a moment when we consider what data to send the script from our JavaScript).

Adding the JavaScript

Now that you have the PHP script in place and ready to accept requests from your JavaScript, it's time to update the JavaScript to save your to-do items!

CODE TO TYPE:

```
function Todo(task, who, dueDate) {
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;
    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                parseTodoItems(this.responseText);
                addTodosToPage();
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
}

function addTodosToPage() {
    var ul = document.getElementById("todoList");
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = document.createElement("li");
        li.innerHTML =
            todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueD
ate;
        ul.appendChild(li);
    }
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
```

```
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    console.log("New task: " + task + ", for: " + who + ", by: " + date);
    var todoItem = new Todo(task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
    saveTodoData();
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}
function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = document.createElement("li");
    li.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
    ul.appendChild(li);
    document.forms[0].reset();
}

function saveTodoData() {
    var todoJSON = JSON.stringify(todos);
    var request = new XMLHttpRequest();
    var URL = "save.php?data=" + encodeURI(todoJSON);
    request.open("GET", URL);
    request.setRequestHeader("Content-Type",
                             "text/plain;charset=UTF-8");
    request.send();
}
```

Save it, open **todo.html**, and click Preview. Add one or two to-do items. Now, reload the page. They should still be there. Open your **todo.json** file; you should see all your to-do items there. (Make sure you close it again before you add any more to-do items).

Let's go over how this works, step-by-step. We added a new function **saveTodoData()** where all the action happens. We call this function from **getFormData()**, after we add a **todoItem** to the page. Let's see what **saveTodoData()** does:

OBSERVE:

```
function saveTodoData() {
    var todoJSON = JSON.stringify(todos);
    var request = new XMLHttpRequest();
    var URL = "save.php?data=" + encodeURI(todoJSON);
    request.open("GET", URL);
    request.setRequestHeader("Content-Type",
                             "text/plain;charset=UTF-8");
    request.send();
}
```

We don't pass any arguments to **saveTodoData()**. Why? Because **saveTodoData()** uses the **todos** array, which is a global variable, so we don't need to pass it. First, **saveTodoData()** converts the **todos** array to a JSON string using **JSON.stringify()**. The **JSON.stringify()** method takes an object or an array and turns it into a string that we can then send as data to a script and save in a file. We save that JSON string in the variable **todoJSON**. We put *every* to-do item in the **todos** array in the JSON string (because the **save.php** script overwrites the entire **todo.json** file each time we call it; you'll see more on this in a moment).

Next, we create a new **XMLHttpRequest** object, and save it in the variable **request**. Just like when we used

XMLHttpRequest to send a request to get data from a server, we will use XMLHttpRequest to send a request to send data to a server. We send that data along with the request. There are two ways to send data to a server script using Ajax: GET and POST. (These are the same GET and POST you can use with forms to send data to a server). Let's take a quick look at each of these methods of sending data.

GET

When you use GET to send a request with data to a server, you create a URL that looks very similar to the URLs you typically use, except you add some data on the end of the URL. You append the data you want to send on the end of the URL, with a "?" between the URL of the script and the data. This is the method we use in **saveTodoData()**. We append the to-do items to the end of the URL for the request to **save.php**. But we can't just send the data as is, because it contains all kinds of characters that would mess up the URL and confuse the browser and the server. So, we have to encode the URL first so all the characters that would confuse the browser and the server get replaced with their encoded versions. For instance, the encoded version of a space (" ") is %20, and the encoded version of a double quote (") is %22.

In **saveTodoData()**, we create the URL for the request like this:

OBSERVE:

```
var URL = "save.php?data=" + encodeURI(todoJSON);
```

In the first part of our URL, we specify the name of the script we want to send the request to: **save.php**.

This URL is a **relative URL**, which means we don't specify the full http://... URL. Rather, we just specify the name of the script, **save.php**, because we know it's in the same directory from which the web application is loaded. This request will be converted automatically to a request for the full URL for us, which is pretty convenient You could use the full URL (known as the **absolute** URL) if you wanted, but that's not required.

After the name of the script, we add **?** to separate the path to the script from the data we're sending to the server.

Next, we type **data=**, which gives a name to the data we're sending. This is just like a variable name, and it allows the PHP script to use that name to GET the data from the URL. In the PHP script, we had the line:

OBSERVE:

```
$myData = $_GET["data"];
```

The result of this PHP code is that it can access the data in the URL and store it in the variable **$myData**, which it then uses to write the data to the file, **todo.json**.

Finally, we append the data in the **todoJSON** string to the end of the URL variable. Remember that **todoJSON** is comprised of all the to-do items in our **todos** array, converted into JSON.

We **encode** the **todoJSON** string using the built-in JavaScript function, **encodeURI()**, which takes a string and turns it into a form suitable for sending as part of a URL with a GET request. So the actual URL that we use in the XMLHttpRequest looks something like this:

OBSERVE:

```
save.php?data=%5B%7B%22task%22:%22get%20milk%22,%22who%22:%22Scott%22,%22dueDate
%22:%22today%22,%22done%22:false%7D,%7B%22task%22:%22get%20broccoli%22,%22who%22
:%22Elisabeth%22,%22dueDate%22:%22today%22,%22done%22:false%7D,%7B%22task%22:%22
get%20balsamic%20vinegar%22,%22who%22:%22Georgia%22,%22dueDate%22:%222012-08-04%
22,%22done%22:false%7D%5D
```

POST

If we want to use POST instead of GET to send the data to the server we can (although in this particular example, we'd have to change the code in the PHP script too, so it won't work). In general, in order to use POST, instead of putting the data in the URL itself, we send it using the request object. We use a similar format for the data (an encoded string), but rather than adding it to the URL string, we send it when we call **request.send()**.

Sending the Request

Okay, we've created a **URL** that contains the name of the script to which we're sending the request, **save.php**, and we've got the data we're sending—the **todos** array converted into JSON—encoded and added to the URL, so how do we send the request?

OBSERVE:

```
function saveTodoData() {
    var todoJSON = JSON.stringify(todos);
    var request = new XMLHttpRequest();
    var URL = "save.php?data=" + encodeURI(todoJSON);
    request.open("GET", URL);
    request.setRequestHeader("Content-Type",
                             "text/plain;charset=UTF-8");
    request.send();
}
```

To send the request, we use the **request.open()** method to tell the **XMLHttpRequest** object which URL we want to send the request to, and the kind of request it is (in this case, it's a GET request). Unlike the request we sent to *retrieve* the data in the **todo.json** file, we don't have to set up an **onreadystatechange** handler now, because we're not expecting any data back from the server (although in some situations, you might expect some data back; it's possible to both send and receive data in the same request).

So we're almost ready to send the request, but first we need to tell the server the kind of data we're sending. There are many kinds of data we could send in a request: XML, HTML, plain text, binary data (like if we're sending an image), and so on. In our case, we're just sending plain text: a JSON string.

Any request sent from a browser to a server contains a lot of information. A request always has a **header** that includes information like the kind of request we're making (called the request "method," in our case, this is GET), information about the encoding, information about the browser sending the request, and the URL to which we're sending the request.

To tell the server the type of data we're sending with the request, we **set one of the fields in the header** named **Content-Type**, and provide the correct value for plain text. This value consists of a MIME type, "text/plain" (a string that represents the type of the file that conforms to the MIME standard of file types), and a "charset" that tells the browser the character encoding of the data. This encoding is usually "UTF-8" which is the current standard for encoding characters (because, among other capabilities, it can encompass all the languages in the world).

So now we're ready to send! We call **request.send()** to send the request to the server, which sends all the data we want to save in **todo.json** along with the request.

When we send the request, the **save.php** script is called on the server, which saves the to-do data into the file **todo.json**. If you recall, we mentioned that the script overwrites anything that's already in this file. That's the reason we send *all* of the to-do items each time we make the request. It's not the most efficient way to do it, but it's a bit less complex and a good place for us to start. Can you think of a better way to do it?

You've updated the application so that you can add new items to your to-do list, *and* saved those items back to the **todo.json** file. You can see how Ajax is a powerful way to increase the functionality and flexibility of your web applications. The To-Do List application is much more useful with the ability to save your to-do items, don't you think?

However, our solution isn't perfect by any means. What happens if more than one person uses this web application? For instance, if you give the URL of the web application to a friend, will they be able to see your to-do items? Will they be able to add items to *your* to-do list?

Creating a to-do list application that can support more than one user is beyond the scope of this course, but you now have a solid understanding of what Ajax is, and how Ajax works.

So now you know that Ajax is a set of techniques that allow you to create what we call "single-page web applications"—web applications that you can use much like a regular desktop application. With Ajax techniques, you can update a web page dynamically with new data, you can change the page when the user interacts with it, and you can retrieve and save data the user enters into the page, all without having to "load" another web page.

In the next lesson we'll look at how to make JavaScript applications better by adding some efficiency to the To-Do List application.

Document Fragments

Lesson Objectives

When you complete this lesson, you will be able to:

- use document fragments to add elements to a page.
- use document fragments in a to-do list application.
- combine common code.
- enhance a to-do list application.

As you now know, a big part of writing Ajax applications, and in fact, most JavaScript *web applications*, is the ability to add and remove elements from the page dynamically, using JavaScript. In the To-Do List application, we add elements from the **todo.json** file to the page when we first load the page, and we add a new to-do item each time the user enters data into the form and clicks the **submit** button.

Each time you add (or remove) an element from a web page, you're accessing the DOM tree—the Document Object Model tree—that is the internal representation of the web page in the browser. For fairly small applications like this one, accessing the DOM doesn't cause any problems because it's not happening too frequently. However, for large applications, frequent access to the DOM can become a problem; each time you access the DOM, the browser has to stop everything and update the entire page. If you have hundreds or even thousands of elements in a page, and you're adding and removing elements frequently, all those DOM operations can really slow things down.

One way we can make adding new elements to a page more efficient is to use *document fragments*.

Using Document Fragments to Add Elements to a Page

A document fragment is a *fragment* of a DOM tree, a chunk of tree that's separated from the rest of the DOM. Why is this useful? Because we can add elements to it *before* we add them to the DOM, avoiding the more expensive DOM operations until we've got our new structure created and ready to go. Once the document fragment contains everything we want to add to the DOM, we can add it using one operation instead of many.

The best way to understand document fragments, of course, is to dive right in and start making them. Let's start by creating a small page with some simple JavaScript:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>Document Fragments</title>
  <meta charset="utf-8">
  <script src="frag.js"></script>
  <style>
        div.box {
            position: relative;
            width: 100px;
            height: 100px;
        }
  </style>
</head>
<body>
   <div id="container">
   </div>
</body>
</html>
```

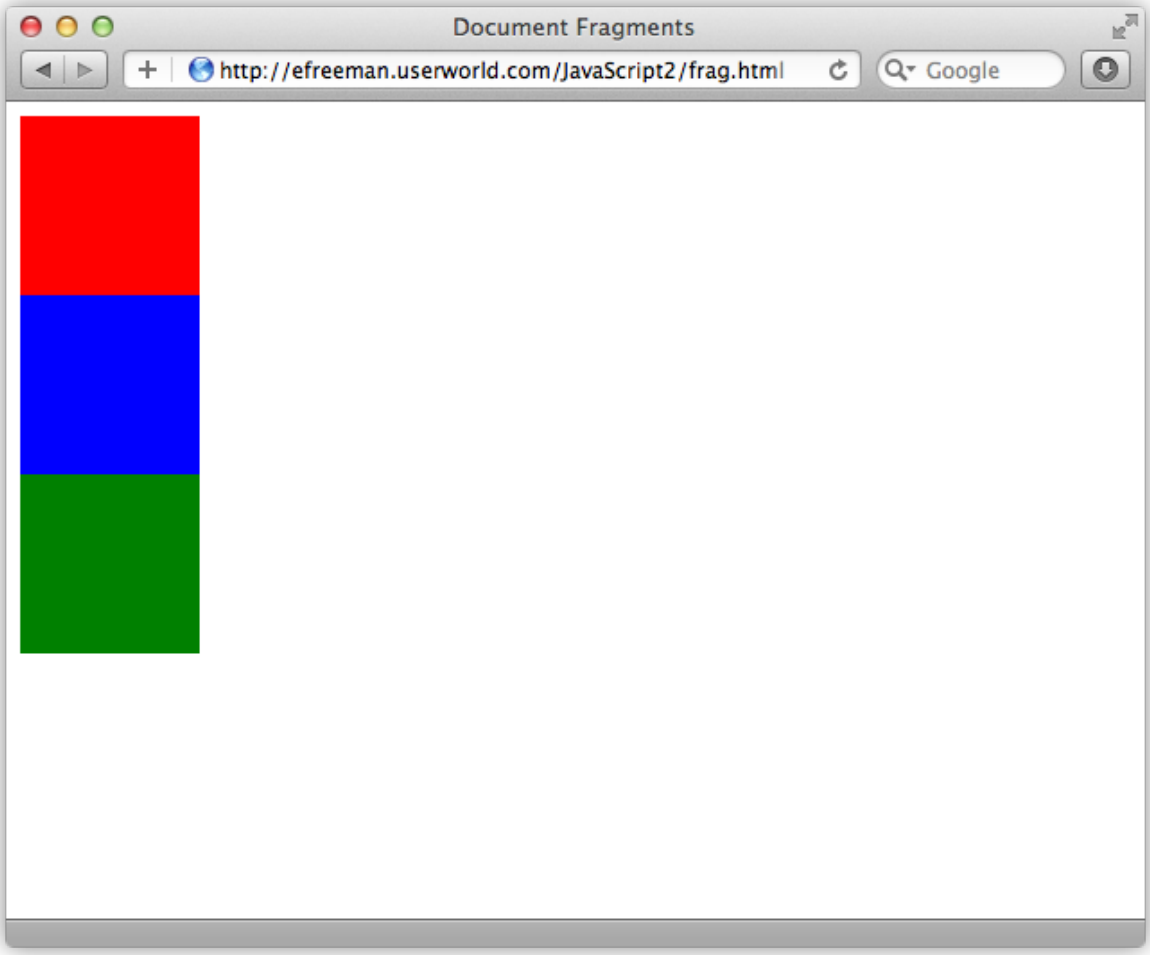
Save it in your **/javascript2** folder as **frag.html**. This page doesn't contain much of anything, so don't preview until after you've added the JavaScript to create the new elements and add them to the page.

Take note of the CSS in the document. We'll use this CSS to style the new elements with the class "box" that we'll add to the page using JavaScript. Create a new file and type in this JavaScript:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var colors = [ "red", "blue", "green" ];
    var container = document.getElementById("container");
    for (var i = 0; i < 3; i++) {
        var box = document.createElement("div");
        box.setAttribute("class", "box");
        box.style.backgroundColor = colors[i];
        container.appendChild(box);
    }
}
```

Save it in your **/javascript2** folder as **frag.js**. Open your **frag.html** file, which links to this **frag.js** file at the top (with the <script> element), and click Preview. You see a page like this:

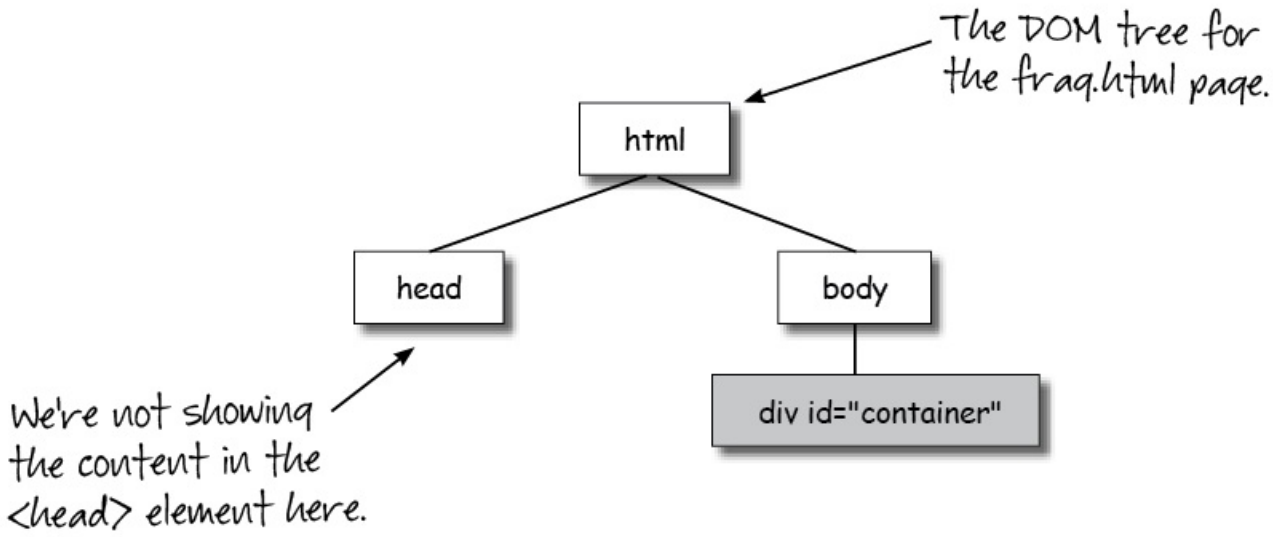


The JavaScript here is fairly straightforward; we use a loop to create three new elements, each with the class "box", and add them one at a time to the DOM by calling

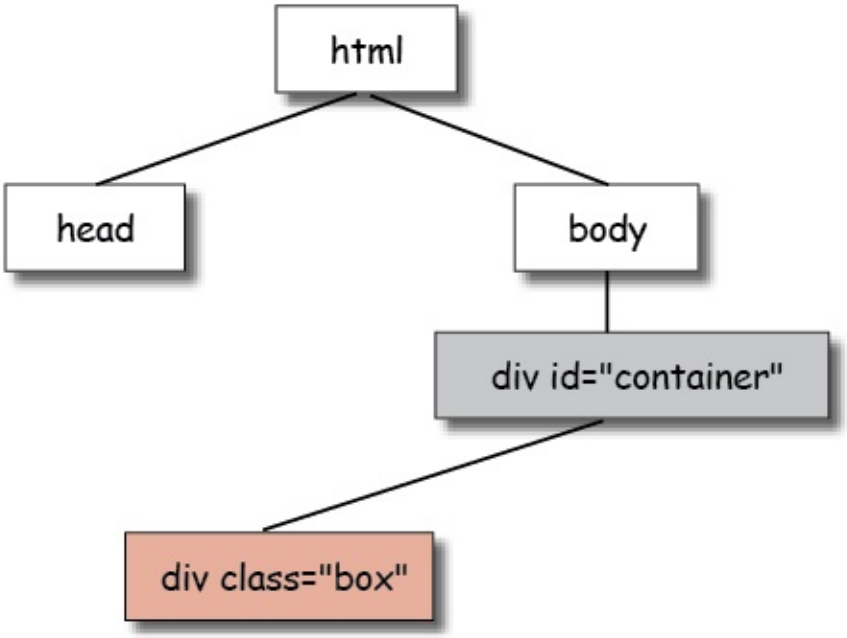
OBSERVE:

```
container.appendChild(box);
```

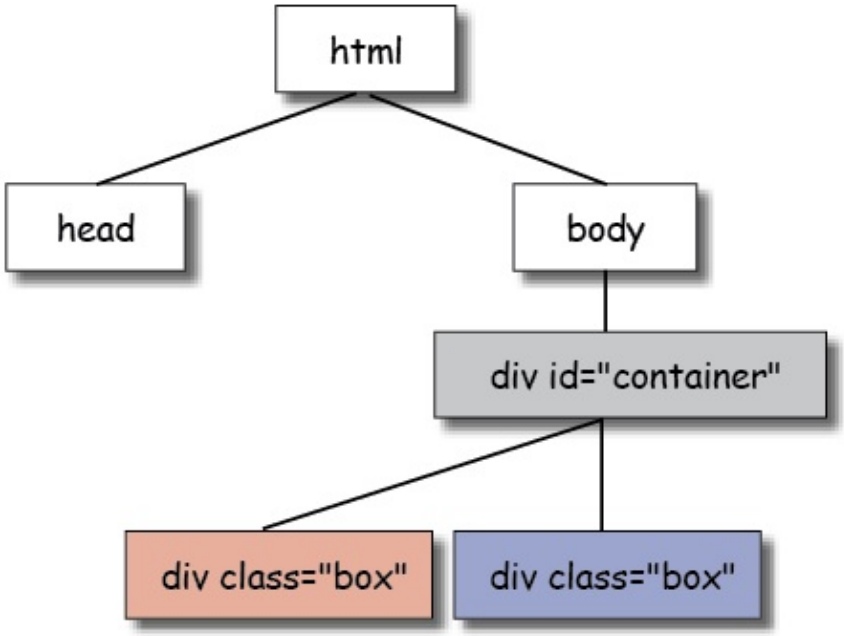
Because **container** is an element in the DOM, each time you call **appendChild(box)**, you add a new element directly to the DOM:



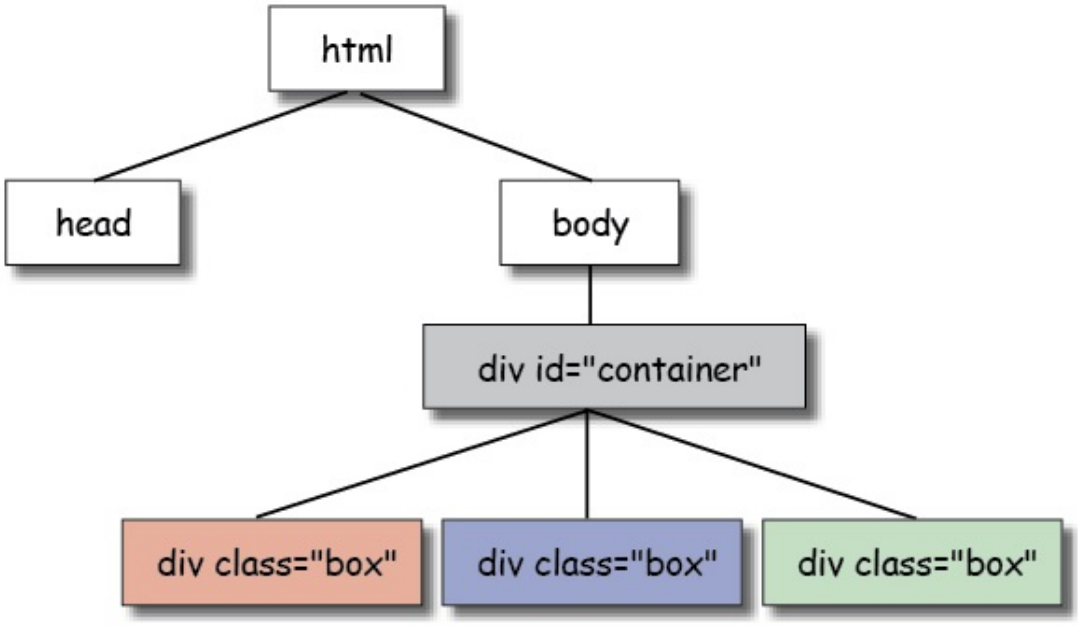
The first time through the loop, we add the red "box" :



Then the blue:



And finally the green:



For our small loop of three elements, this is fine, but imagine you're adding 100 or 1000 elements, like you might if you were, say, initializing the state of a game application that has many elements representing game components. In that case, updating the DOM for each individual element like this will slow down the initialization process substantially.

We can improve the performance of this code by using document fragments.

A document fragment is an empty node. It's just like the nodes you find in a DOM tree—where "node" just means an element object in the tree—except that it doesn't represent any specific element. It's a completely generic node that doesn't mean anything. It's powerful though, because you can add elements to the fragment as children, just like you can with a DOM tree.

Let's update the code for the example to use a document fragment. Modify **frag.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var colors = [ "red", "blue", "green" ];
    var container = document.getElementById("container");
    var fragment = document.createDocumentFragment();
    for (var i = 0; i < 3; i++) {
        var box = document.createElement("div");
        box.setAttribute("class", "box");
        box.style.backgroundColor = colors[i];
        container.appendChild(box);
        fragment.appendChild(box);
    }
    container.appendChild(fragment);
}
```

Save it, then open your **frag.html** file and click Preview. Your page will look exactly the same, but it will be just a tiny bit more efficient. We like that.

Let's walk through the code and see how it works:

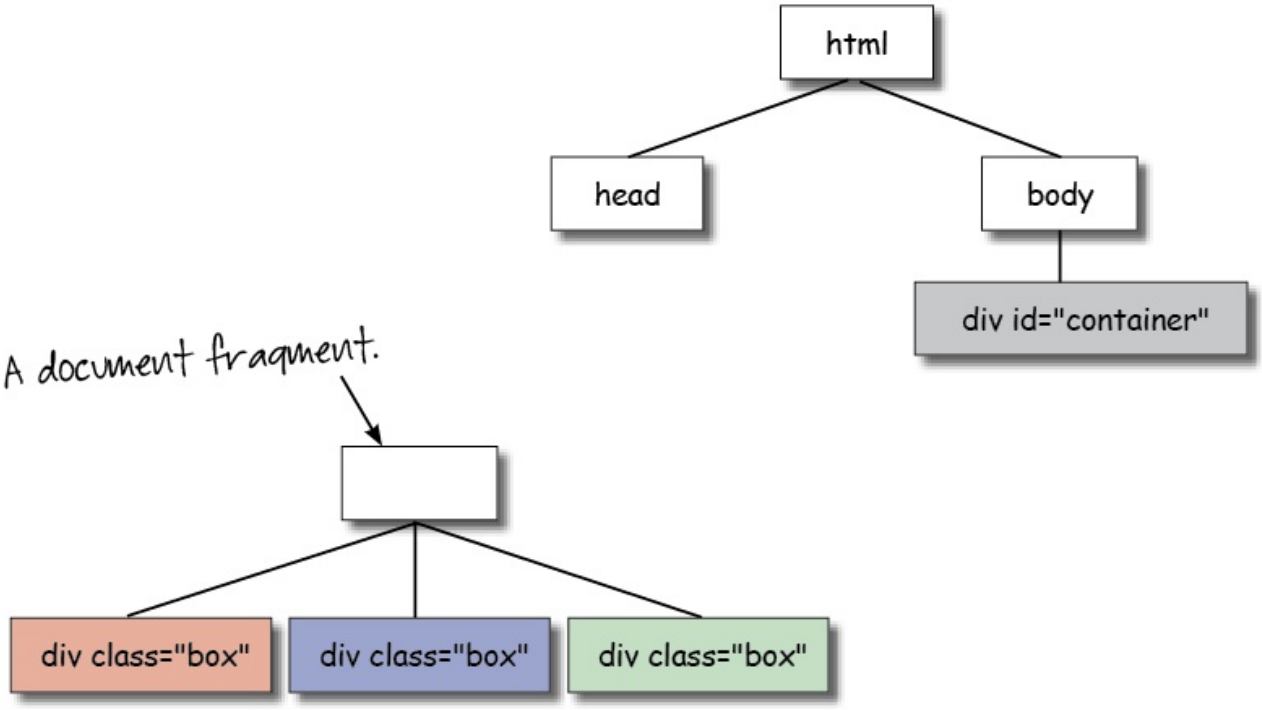
OBSERVE:

```
function init() {
    var colors = [ "red", "blue", "green" ];
    var container = document.getElementById("container");
    var fragment = document.createDocumentFragment();
    for (var i = 0; i < 3; i++) {
        var box = document.createElement("div");
        box.setAttribute("class", "box");
        box.style.backgroundColor = colors[i];
        fragment.appendChild(box);
    }
    container.appendChild(fragment);
}
```

We still need to get the "container" from the DOM because we'll use it later. After we get that, we **create an empty document fragment using document.createDocumentFragment()**. Think of it like an empty element just hanging out in your program, separate from the DOM, except that it doesn't actually represent any specific element; it's just a generic node waiting for you to add things to it.

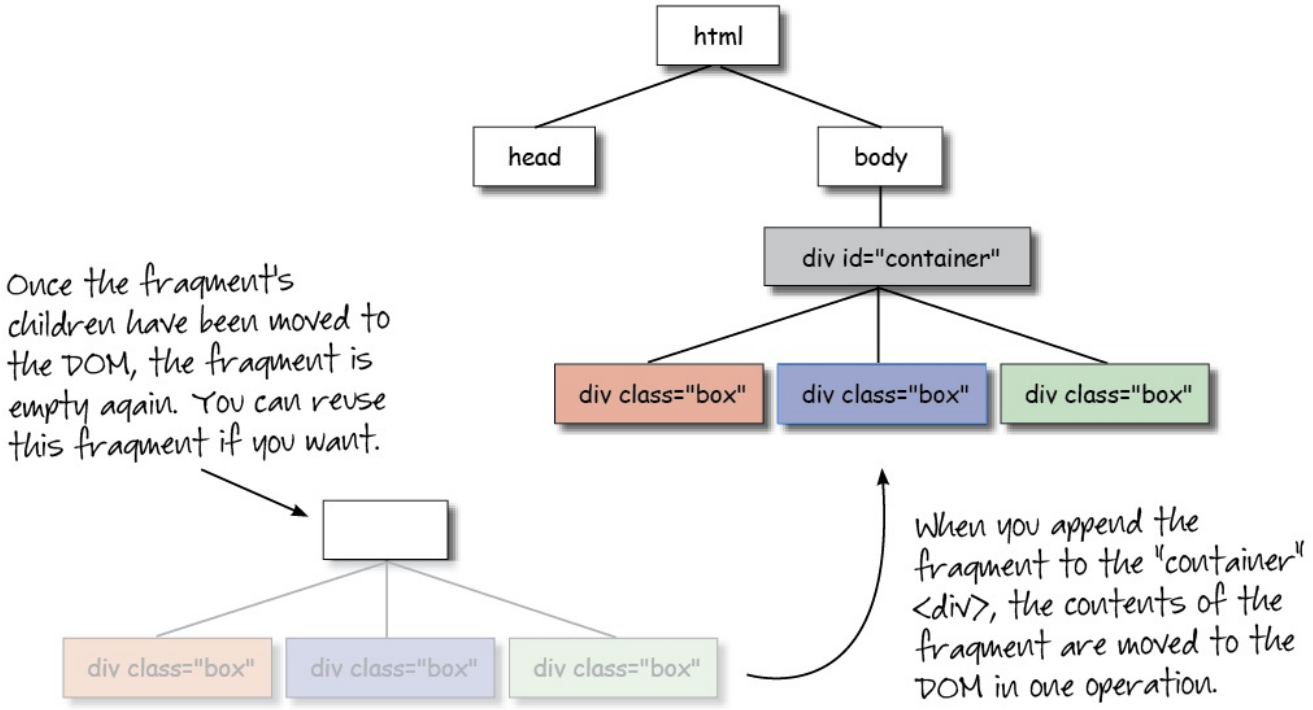
Next we loop, as before, only this time, each time through the loop, we **add a new "box" ** to the **fragment** rather than the DOM. We still use the **appendChild()** method as usual, because a fragment object is just like an element object and as such, it has all the same methods.

Once we've completed the loop, here's what we've got: we have a DOM tree with an empty "container" , and a document fragment sitting off separately from the DOM with three "box" s, waiting to be added to the DOM:



Finally, we **add the elements in the fragment to the DOM** by calling **appendChild()** again. Notice that we're calling **appendChild() on the "container" ** and **passing the fragment as the argument**.

Now this is where adding a fragment to the DOM differs a little from adding an element to the DOM using **appendChild()**. Instead of appending the fragment along with everything contained in the fragment to the DOM, only the elements contained in the fragment are moved to the DOM, like this:



...which is perfect, because now our "container" contains the three "box" s just like we want! All the new elements are added to the DOM in *one* operation, instead of the three separate operations that we needed before when we weren't using the fragment. This is a lot more efficient, and we're left with an empty fragment (which you can even use again if you want). Can you think of any reason you wouldn't want the actual fragment in the DOM?

Using Document Fragments in the To-Do List Application

Now that you know how to use a document fragment, let's see if we can improve the To-Do List application we've been building. If you remember from the previous lessons, the To-Do List application adds to-do items to the page when the page is first loaded by reading them from a file using XMLHttpRequest, and then adding each item to the page in a loop from an array.

Also, recall that we have two very similar functions in our JavaScript: **addTodosToPage()**, which is the function that's called when the page is loaded to add the to-do items from **todo.json** to the page, and **addTodoToPage()**, which is the function that's called when you add a to-do item using the form.

Compare the two functions:

OBSERVE:

```
function addTodosToPage() {
    var ul = document.getElementById("todoList");
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = document.createElement("li");
        li.innerHTML =
            todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
        ul.appendChild(li);
    }
}
```

...and:

OBSERVE:

```
function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = document.createElement("li");
    li.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
    ul.appendChild(li);
    document.forms[0].reset();
}
```

The main difference is that **addTodosToPage()** adds to-do items to the page from the **todos** array, while **addTodoToPage()** adds the one **todoItem** that's passed into the function. But other than that, they both essentially do the same thing: they each **create a new <li> element, and add it to the DOM**. **addTodosToPage()** does this each time through the loop, while **addTodoToPage()** does it once.

We can improve this code in two ways. First, we can combine some of the common code that is used by both functions so we aren't repeating ourselves. Second, we can make **addTodosToPage()** more efficient by adding all the to-do items to a document fragment before adding them to the DOM.

Combining Common Code

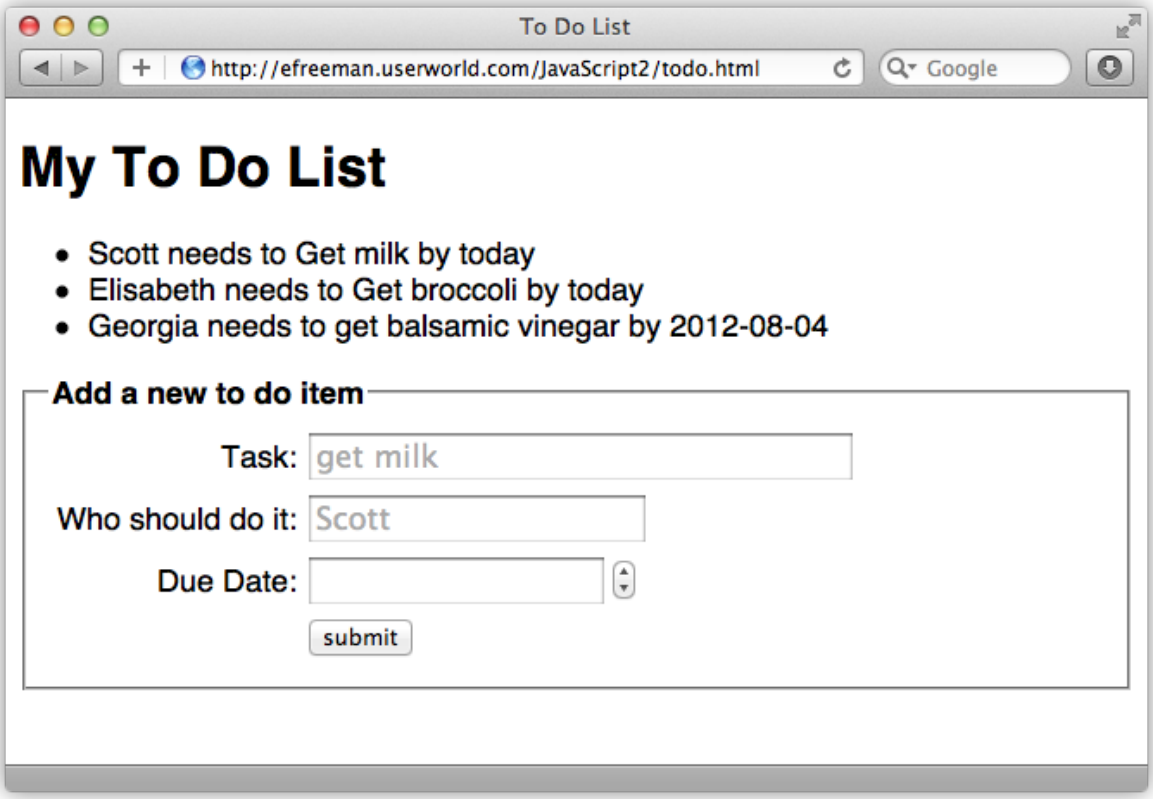
Let's take it one step at a time. First, we'll combine some common code into a new function, **createNewTodoItem()**. Modify **todo.js** as shown:

CODE TO TYPE:

```
function Todo(task, who, dueDate) {
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;

    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                parseTodoItems(this.responseText);
                addTodosToPage();
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
}
function addTodosToPage() {
    var ul = document.getElementById("todoList");
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = createNewTodo(todoItem);
        var li = document.createElement("li");
        li.innerHTML =
            todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueD
ate;
        ul.appendChild(li);
    }
}
function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = createNewTodo(todoItem);
    var li = document.createElement("li");
    li.innerHTML =
```

```
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
    ul.appendChild(li);
    document.forms[0].reset();
}

function createNewTodo(todoItem) {
    var li = document.createElement("li");
    li.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
    return li;
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    var todoItem = new Todo(task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
    saveTodoData();
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}

function saveTodoData() {
    var todoJSON = JSON.stringify(todos);
    var request = new XMLHttpRequest();
    var URL = "save.php?data=" + encodeURI(todoJSON);
    request.open("GET", URL);
    request.setRequestHeader("Content-Type",
                             "text/plain;charset=UTF-8");
    request.send();
}
```

Save it, then open your **todo.html** file and click Preview. Your To-Do List application should work exactly the same way as it did before; that is, it should load any to-do items you have in your **todo.json** file when you load the page, and you should be able to add items using the form.



We took the code that is common to the **addTodosToPage()** and **addTodoToPage()** functions:

OBSERVE:

```
var li = document.createElement("li");
li.innerHTML =
    todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
ul.appendChild(li);
```

...and put it in a new function, **createNewTodo()**:

OBSERVE:

```
function createNewTodo(todoItem) {
    var li = document.createElement("li");
    li.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
    return li;
}
```

Now that the common code is in one function (**createNewTodo()**) that returns the new <li> element, we can call this function whenever we need to create a new <li> element, and pass in the **todoItem** object to use. So, that's what we did in the **addTodosToPage()** and **addTodoToPage()** functions. Instead of having both these functions create the <li> element for the to-do item, now they just call **createNewTodo()** to do it for them.

Improving the Code with a Document Fragment

Next, we can improve the function **addTodosToPage()** by using a document fragment. Modify **todo.js** as shown:

CODE TO TYPE:

```
function Todo(task, who, dueDate) {
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;

    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                parseTodoItems(this.responseText);
                addTodosToPage();
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
}
function addTodosToPage() {
    var ul = document.getElementById("todoList");
    var listFragment = document.createDocumentFragment();
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = createNewTodo(todoItem);
        ul.appendChild(li);
        listFragment.appendChild(li);
    }
    ul.appendChild(listFragment);
}
function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = createNewTodo(todoItem);
    ul.appendChild(li);
    document.forms[0].reset();
}
```

```
function createNewTodo(todoItem) {
    var li = document.createElement("li");
    li.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
    return li;
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    var todoItem = new Todo(task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
    saveTodoData();
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}

function saveTodoData() {
    var todoJSON = JSON.stringify(todos);
    var request = new XMLHttpRequest();
    var URL = "save.php?data=" + encodeURI(todoJSON);
    request.open("GET", URL);
    request.setRequestHeader("Content-Type",
                             "text/plain;charset=UTF-8");
    request.send();
}
```

Save it, open your **todo.html** file, and click Preview. Your To-Do List application will behave the same way as it did before.

Now, instead of adding each to-do item from the **todo.json** file to the DOM one at a time, we're adding them all in one chunk by using a document fragment:

OBSERVE:

```
function addTodosToPage() {
    var ul = document.getElementById("todoList");
    var listFragment = document.createDocumentFragment();
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = createNewTodo(todoItem);
        listFragment.appendChild(li);
    }
    ul.appendChild(listFragment);
}
```

Just like we did before in our simple "box" example, we **create a fragment** where we can add all the new to-do items, and then **add each to-do item in the loop to the fragment**. Once we've added all the to-do items in the **todos** array to the fragment, we can **add the new list item elements in the fragment to the DOM**. Remember, the elements in the fragment are moved from the fragment to the DOM, and then the fragment is left empty, still separate from the DOM.

Enhancing the To-Do List Application

We've made some subtle improvements to the To-Do List application. Next, let's enhance the application by adding a little more structure to each to-do item in the to-do list and make use of the **done** property. We'll even add some CSS style to make it look better. Modify **todo.js** as shown:

CODE TO TYPE:

```
function Todo(task, who, dueDate) {
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;

    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                parseTodoItems(this.responseText);
                addTodosToPage();
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
}

function addTodosToPage() {
    var ul = document.getElementById("todoList");
    var listFragment = document.createDocumentFragment();
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = createNewTodo(todoItem);
        listFragment.appendChild(li);
    }
    ul.appendChild(listFragment);
}

function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = createNewTodo(todoItem);
    ul.appendChild(li);
    document.forms[0].reset();
```

```
}

function createNewTodo(todoItem) {
    var li = document.createElement("li");
    li.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;
    var spanTodo = document.createElement("span");
    spanTodo.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;

    var spanDone = document.createElement("span");
    if (!todoItem.done) {
        spanDone.setAttribute("class", "notDone");
        spanDone.innerHTML = "     ";
    }
    else {
        spanDone.setAttribute("class", "done");
        spanDone.innerHTML = " &#10004; ";
    }

    li.appendChild(spanDone);
    li.appendChild(spanTodo);
    return li;
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    var todoItem = new Todo(task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
    saveTodoData();
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}

function saveTodoData() {
    var todoJSON = JSON.stringify(todos);
    var request = new XMLHttpRequest();
    var URL = "save.php?data=" + encodeURI(todoJSON);
    request.open("GET", URL);
    request.setRequestHeader("Content-Type",
                             "text/plain;charset=UTF-8");
    request.send();
}
```

If you save and run this code now, it will run as expected, but won't look much different, so you might want to wait until we add the CSS below to run it. Before we add the CSS, let's step through the changes in the JavaScript.

All the changes are in the **createNewTodo()** function:

OBSERVE:

```
function createNewTodo(todoItem) {
    var li = document.createElement("li");
    var spanTodo = document.createElement("span");
    spanTodo.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;

    var spanDone = document.createElement("span");
    if (!todoItem.done) {
        spanDone.setAttribute("class", "notDone");
        spanDone.innerHTML = "     ";
    }
    else {
        spanDone.setAttribute("class", "done");
        spanDone.innerHTML = " &#10004; ";
    }

    li.appendChild(spanDone);
    li.appendChild(spanTodo);
    return li;
}
```

First, notice that we add two elements to the <li> element holding each to-do item. Before, we simply added some text to the **innerHTML** of the <li> element directly, now we add two elements: **one to hold the to-do information** about who needs to do the task, what the task is, and when it's due (the same information we added directly to the <li> element previously); and another ** to hold the information about whether an element is done or not**. We're representing the done or not done information with a class so we can style the appropriately (as you'll see after you add the CSS below). If the to-do item is *not* done, we add the class "notDone" to the **spanDone** element, and if it *is* done, then we add the class "done". Also notice that we're setting the **innerHTML** of the "notDone" to five spaces— is the HTML entity for a non-breaking space—and setting the **innerHTML** of the "done" to two spaces and a check mark—✔ is the HTML entity for the check mark. You'll see what this looks like in a minute, after you add the CSS and preview again.

After creating the <li> element and the two elements, we **add the two elements to the **: first the **spanDone**, so we can show if an item is done or not on the left part of the to-do item, and then the **spanTodo**, which contains the text of the to-do item (the same text as we were showing previously). Then we return the <li> element, just like we did before. Remember that both the **addTodosToPage()** and **addTodoToPage()** functions call the **createNewTodo()** function, so they get the <li> element back and add it to the DOM, so we can see our to-do item.

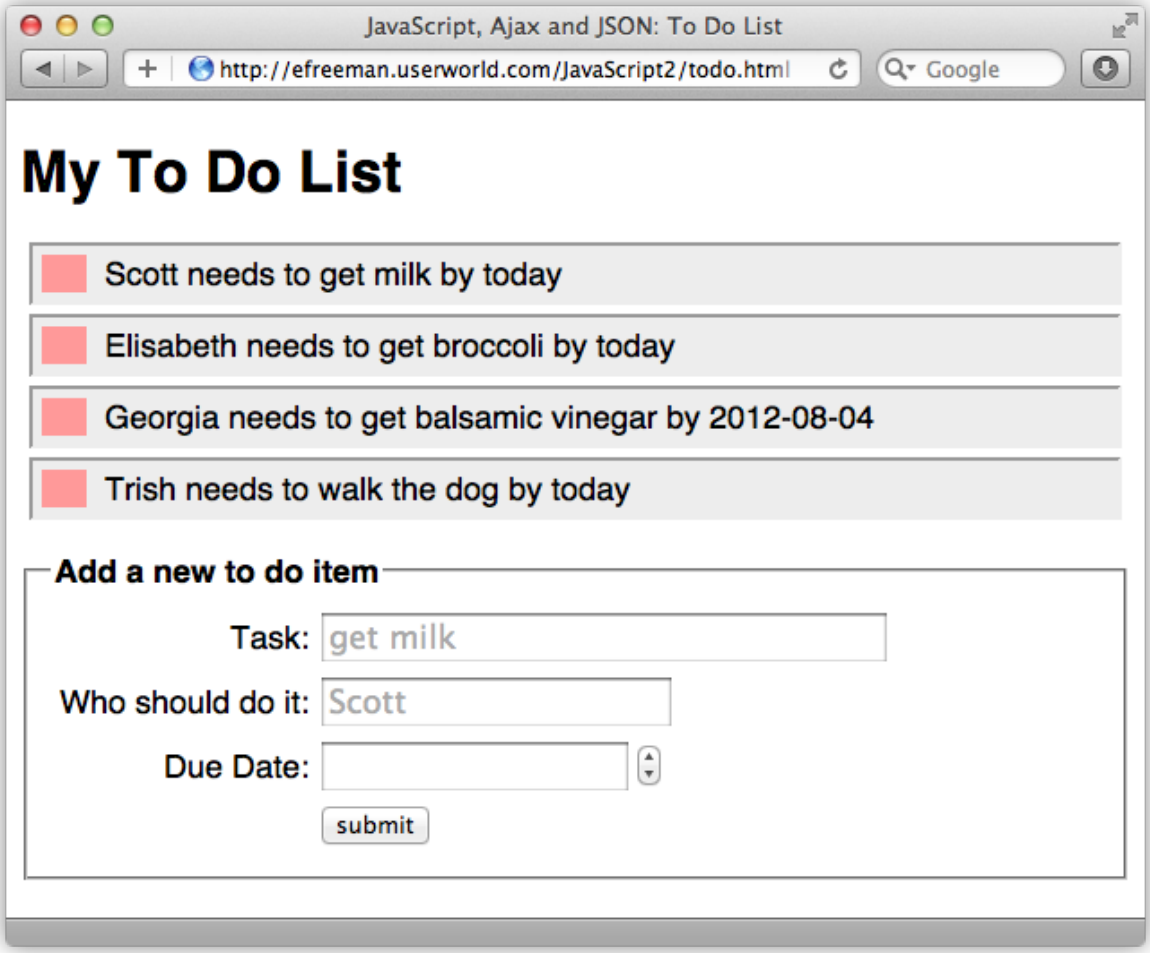
Okay, we're almost there! Modify your **todo.css** file as shown so you can see the result of adding the two elements to the HTML:

CODE TO TYPE:

```
body {
    font-family: Helvetica, Ariel, sans-serif;
}
legend {
    font-weight: bold;
}
div.tableContainer {
    display: table;
    border-spacing: 5px;
}
div.tableRow {
    display: table-row;
}
div.tableRow label {
    display: table-cell;
    text-align: right;
}
div.tableRow input {
    display: table-cell;
}
ul#todoList {
    list-style-type: none;
    margin-left: 0px;
    padding-left: 0px;
}
ul#todoList li {
    padding: 5px;
    margin: 5px;
    background-color: #ededed;
    border: 2px inset #ededed;
}
ul#todoList li span.notDone {
    margin-right: 10px;
    background-color: #FF9999;
}
ul#todoList li span.done {
    margin-right: 10px;
    background-color: #80CC80;
}
```

 Save it, open your **todo.html** file, and click 

. Now you should see your to-do items looking a lot snazzier:

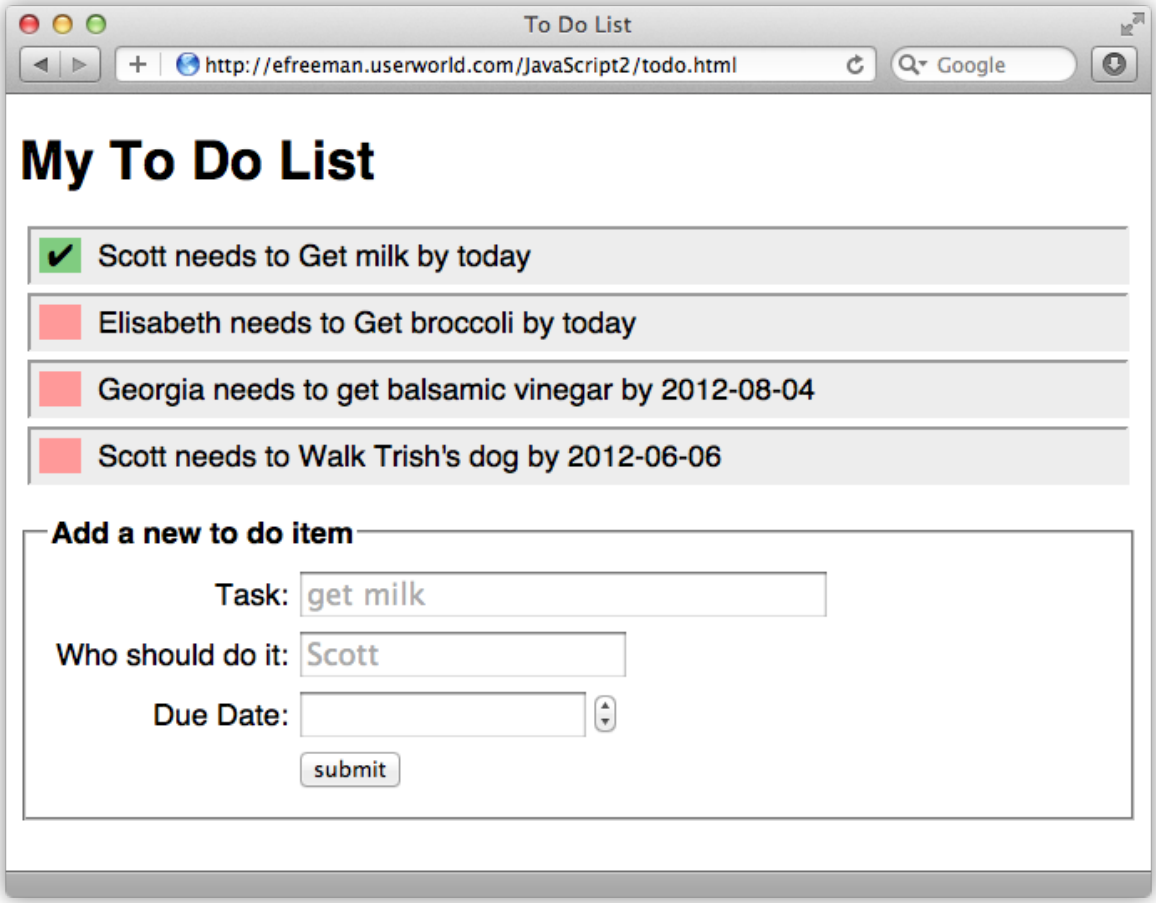


To test the "done" vs. "notDone" code, make sure you have one to-do item in your **todo.json** file that uses "true" for the **done** property. You can do this by editing your **todo.json** file and updating the code. For instance, I updated the first item in my file, so my JSON looks like this:

OBSERVE:

```
[{"task":"Get milk","who":"Scott","dueDate":"today","done":true},
 {"task":"Get broccoli","who":"Elisabeth","dueDate":"today","done":false},
 {"task":"get balsamic vinegar","who":"Georgia","dueDate":"2012-08-04","done":false},
 {"task":"Walk Trish's dog","who":"Scott","dueDate":"2012-06-06","done":false}]
```

Once you've made this change (or a similar change in your **todo.json**, which likely has different items in it at this point!), either reload your already-open browser window with your to-do list application, or click Preview on your **todo.html** file again. You'll see a nice-looking green box with a check mark indicating that your to-do item is done!



Take a look at the CSS to see how we styled the to-do items and notice the two classes, "done" and "notDone". We're using them in our JavaScript code to get the look of a "done" item (a green box with a check mark) and a "notDone" item (a red box with no check mark).

Now, you've probably noticed that the only way to mark an item as "done" is to edit your **todo.json** file, which doesn't seem like a very user-friendly process. Don't worry about that for now, we'll come back to this issue in a later lesson.

In this lesson, most of what you learned is behind-the-scenes stuff. You learned how to improve your code by reducing redundant code, combine common code into one function, and make your code more efficient by using document fragments.

As a programmer, you'll often discover ways to make your code better by reducing redundant code or making it more efficient, especially as you begin to work on bigger projects. We call this *refactoring* your code. The main goals of refactoring are to reduce the complexity of your code, improve its readability, and make it easier to maintain. You'll want to get into the habit of assessing your code frquently as you build it, to see if it needs refactoring.

Enjoy the quiz and project, and stay tuned for the next lesson, where you'll learn a whole new technique for storing your to-do items!

Introduction to Local Storage

Lesson Objectives

When you complete this lesson, you will be able to:

- use local storage in JavaScript.
- find out what local storage is available in your browser.
- use the localStorage object with JavaScript.
- iterate through local storage.
- remove items from local storage.

In previous lessons, you learned how to use XMLHttpRequest in your web application to communicate with a server, both to retrieve data to update your web page, and to save data so that the next time a web application is loaded, the data is still there.

XMLHttpRequest is a great solution for saving and retrieving long-term data, or data that you want your web application to be available to users of any browsers.

Sometimes though, you only need to store data for a short period of time, for example, preferences for a shopping session. Other times you might want to save some data that a user has downloaded so that if they return to your page again, your application doesn't have to download it again if it's still saved.

In these cases, you can use a technique called *Local Storage*.

What is Local Storage?

Local Storage is, as its name would suggest, storage that's *local*. Local where? In your browser! Let's check it out. Open a new file and type the code as shown:

CODE TO TYPE:

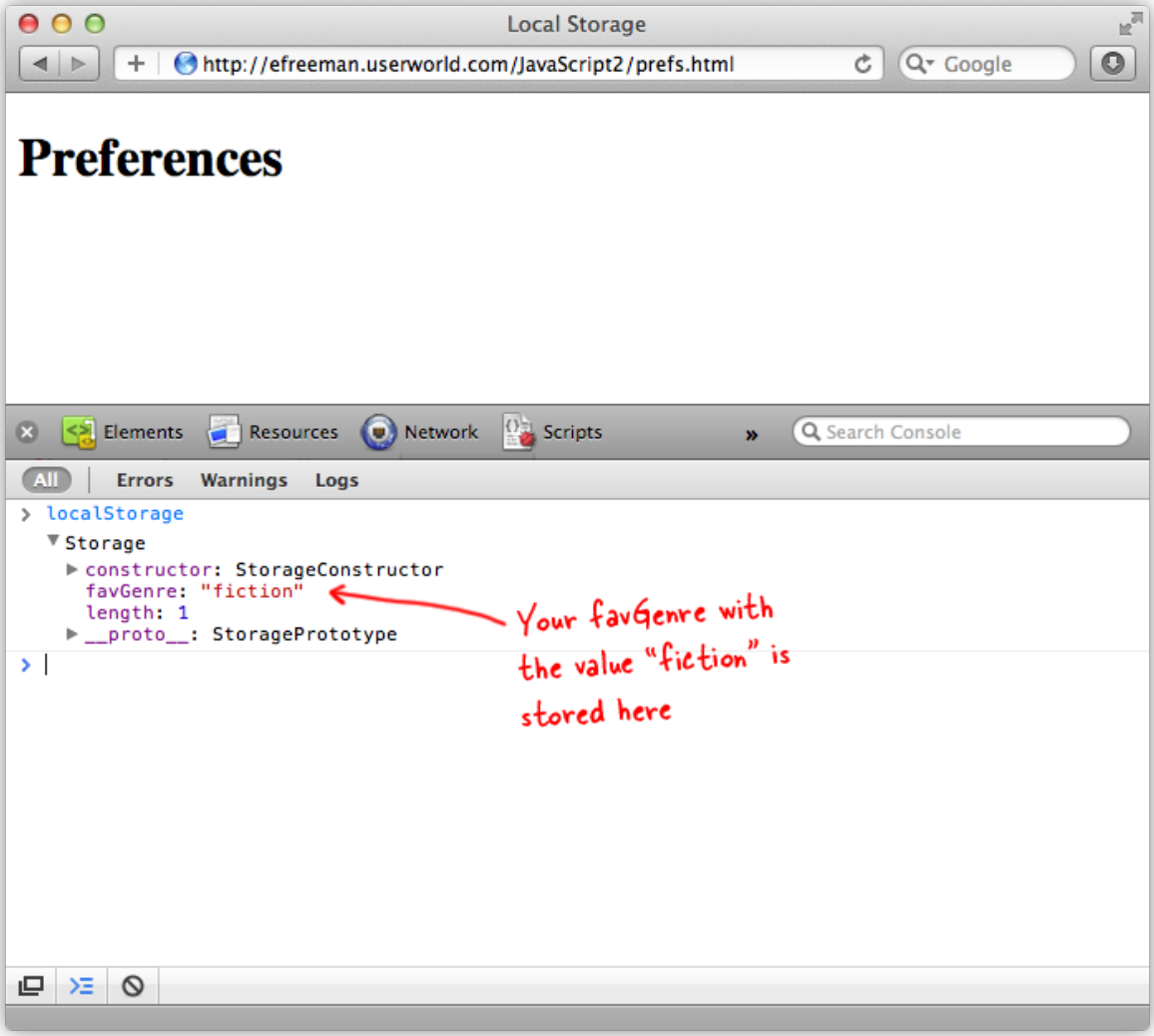
```
<!doctype html>
<html>
<head>
<title>Local Storage</title>
  <meta charset="utf-8">
  <script src="prefs.js"></script>
</head>
<body>
   <h1>Preferences</h1>
   <ul id="items">
   </ul>
</body>
</html>
```

Save it in your **/javascript2** folder as **prefs.html**. The <script> tag includes a link to **prefs.js**; let's create that now:

CODE TO TYPE:

```
window.onload = init;

function init() {
    localStorage.setItem("favGenre", "fiction");
}
```

Save it in your **/javascript2** folder as **prefs.js**. Open your **prefs.html** file and click Preview. You won't see anything special in the HTML page, so open your developer console and type **localStorage** at the JavaScript console prompt:



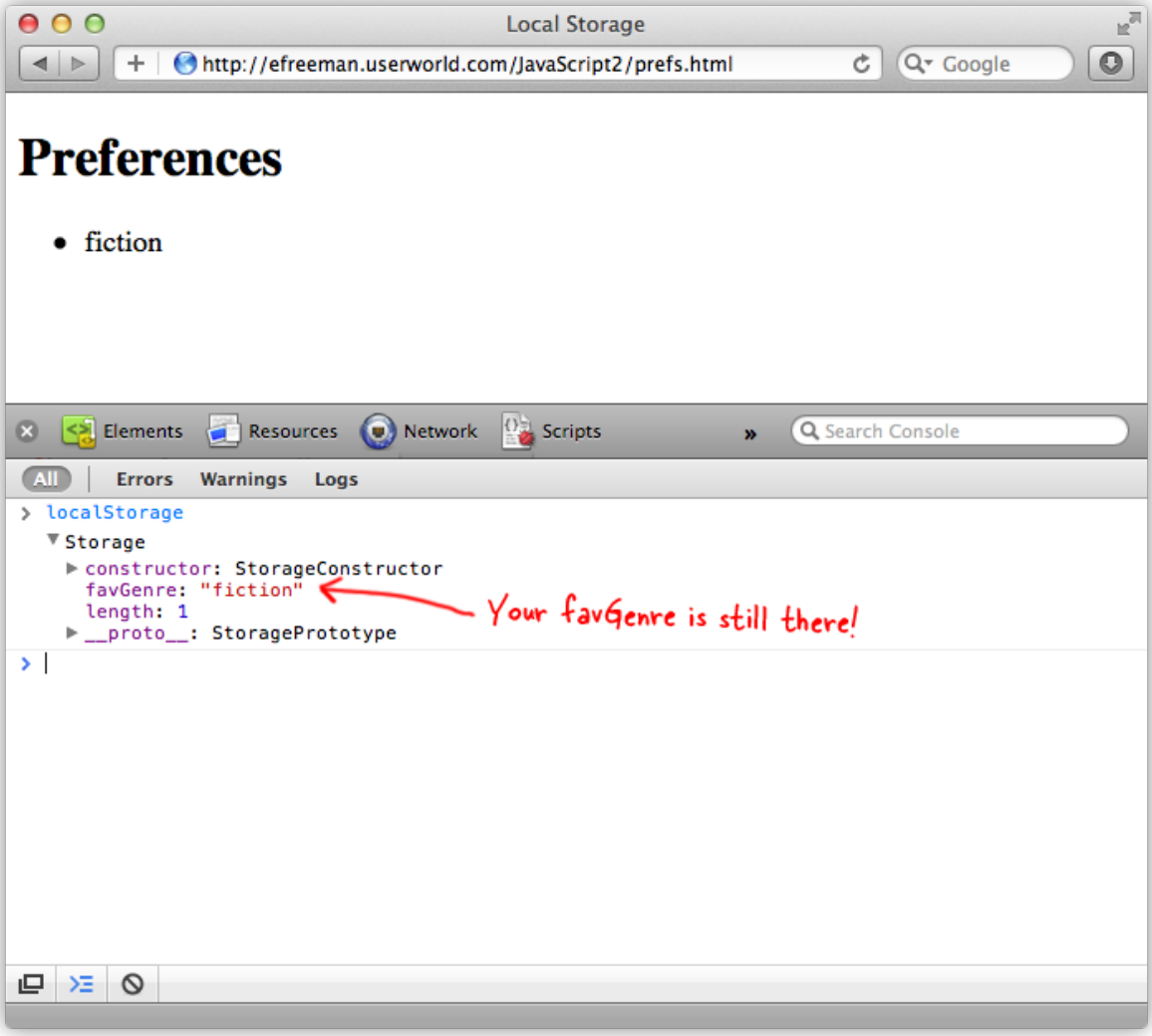
We created a little bit of data, with the name **"favGenre"** and the value **"fiction"**, and stored this data in the object named **localStorage**. Before we dig into Local Storage, modify **prefs.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    localStorage.setItem("favGenre", "fiction");
    var favoriteGenre = localStorage.getItem("favGenre");

    var ul = document.getElementById("items");
    var li = document.createElement("li");
    li.innerHTML = favoriteGenre;
    ul.appendChild(li);
}
```

Save it. In this version of **prefs.js**, we delete the first line which *creates* the **"favGenre"** bit of data, and replace that line with new code that *retrieves* that bit of data, using the same name (**"favGenre"**) to save it, and then add the data to the web page by creating a new list item and adding the list item to the "items" list (which already exists in your HTML).

Before you preview, make sure you *close* the browser window you had open when you ran the program before. We want to make sure we're starting from scratch. Now, open **prefs.html**, and click Preview. Do you see "fiction" in the list in the page? Open the developer console, and again, type **localStorage** at the JavaScript console prompt:



So even though you *closed* the browser window, when you retrieve the data you stored in the **localStorage** object, it is still there! So, what's going on? How does the data in the **localStorage** object survive, even though you closed the browser window, and reloaded the **prefs.html** page?!

Here's the secret: the **localStorage** object is a built-in JavaScript object that saves data *in the browser itself*. It works in all browsers. As you know, the browser is a program that you use to browse the web and run web applications. When you run your web application in the browser, and use **Local Storage** to store data, the browser reserves some space just for your web application's data, and makes it accessible to your web application through the **localStorage** object.

Note We use the term "Local Storage" to refer to the feature offered by browsers, and "localStorage" to refer specifically to the **localStorage** object in JavaScript.

Have you ever heard of *browser cookies*? Well, **Local Storage** is similar to cookies, except that in many ways, it's *way better*. We'll come back to cookies later, for now we'll focus on Local Storage.)

Exploring Local Storage in the Browser

All modern browsers include tools you can use to inspect Local Storage. Before we go further using JavaScript programming with Local Storage, let's learn how to find out what's in your browser's Local Storage.

These next few videos show how to explore Local Storage using browser tools:

Firefox, Chrome, and Opera

Safari 5

Safari 6

If you've upgraded to Safari 6 (from Safari 5, the current version at the time this course was written), you'll need to watch this video:

Safari 6

Microsoft Internet Explorer 9

Internet Explorer 9 works much like Firefox, Chrome, and Opera. Select **Tools | F12 Developer Tools** (or press **F12**) and select **Console**. Then, type **localStorage** and any other commands at the >> prompt.

After watching the videos, make sure you try accessing Local Storage from at least one or more browsers. Try adding new items to Local Storage from the JavaScript console. Make sure you know how to determine what's in Local Storage; it will come in handy for debugging your programs later.

Note Browser Developer Tools usually change with each new version of a browser. While the functionality should be the same or better with each new version, user interfaces sometimes changes dramatically. So if your browser Developer Tools look different from the versions shown in the videos or screen shots, don't worry. You'll still be able to do the same things, you just might have to work a bit to figure out the differences in the UI.

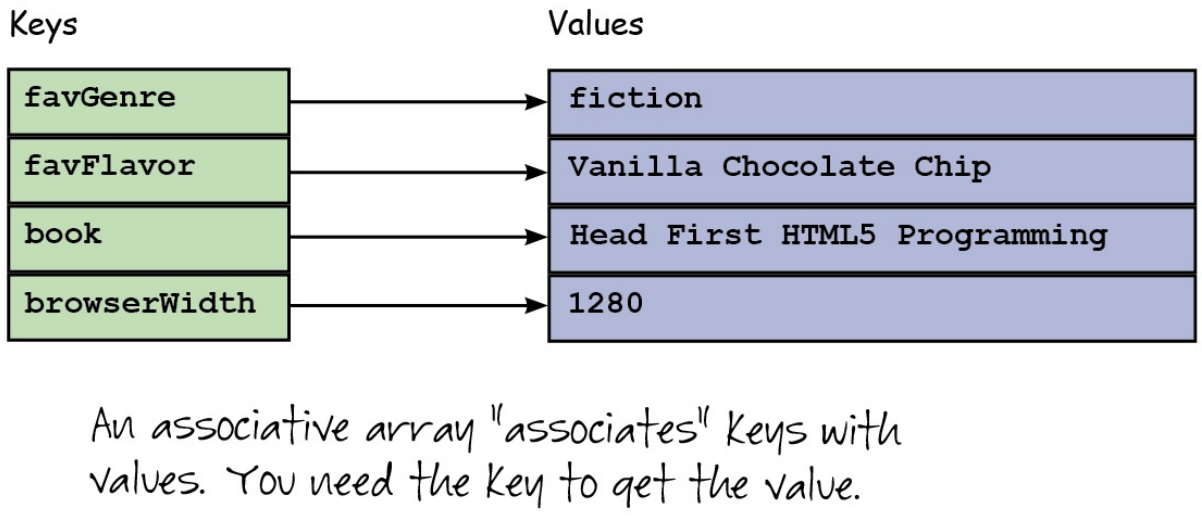
A Closer Look at the localStorage Object

Let's take a closer look at the **localStorage** object and the **setItem()** method first.

OBSERVE:

```
localStorage.setItem("favGenre", "fiction");
```

The **localStorage** object is similar to objects you create in JavaScript, except that it's built in to JavaScript, and has access to native browser code that can store and retrieve data from the browser local storage area. The **setItem()** method allows you to store data. Each item stored in Local Storage has a **key** and a **value**. This type of computer storage is called an *associative array*; instead of using a numerical index (and having a specific order based on that index) like in a regular array, an associative array uses keys as the index values (and has no inherent ordering).



The **key** you use to store the data must be a string; in this case, we used the string **"favGenre"**. Each key must be unique—if you use the *same* key to store another piece of data, you'll overwrite the previous piece of data you stored in the bucket for that key.

When you want to retrieve a value from Local Storage, you need to use the same **key**:

OBSERVE:

```
var favoriteGenre = localStorage.getItem("favGenre");
```

The **localStorage** has a **getItem()** method that you use to retrieve a value by its **key**. Here, we're using the **"favGenre" key** to retrieve the bit of data we just stored. The **value returned** is the value you stored; we're putting that value in the variable **favoriteGenre**, which you can then use in your program. In the **prefs** example, we added that value to a list in the web page.

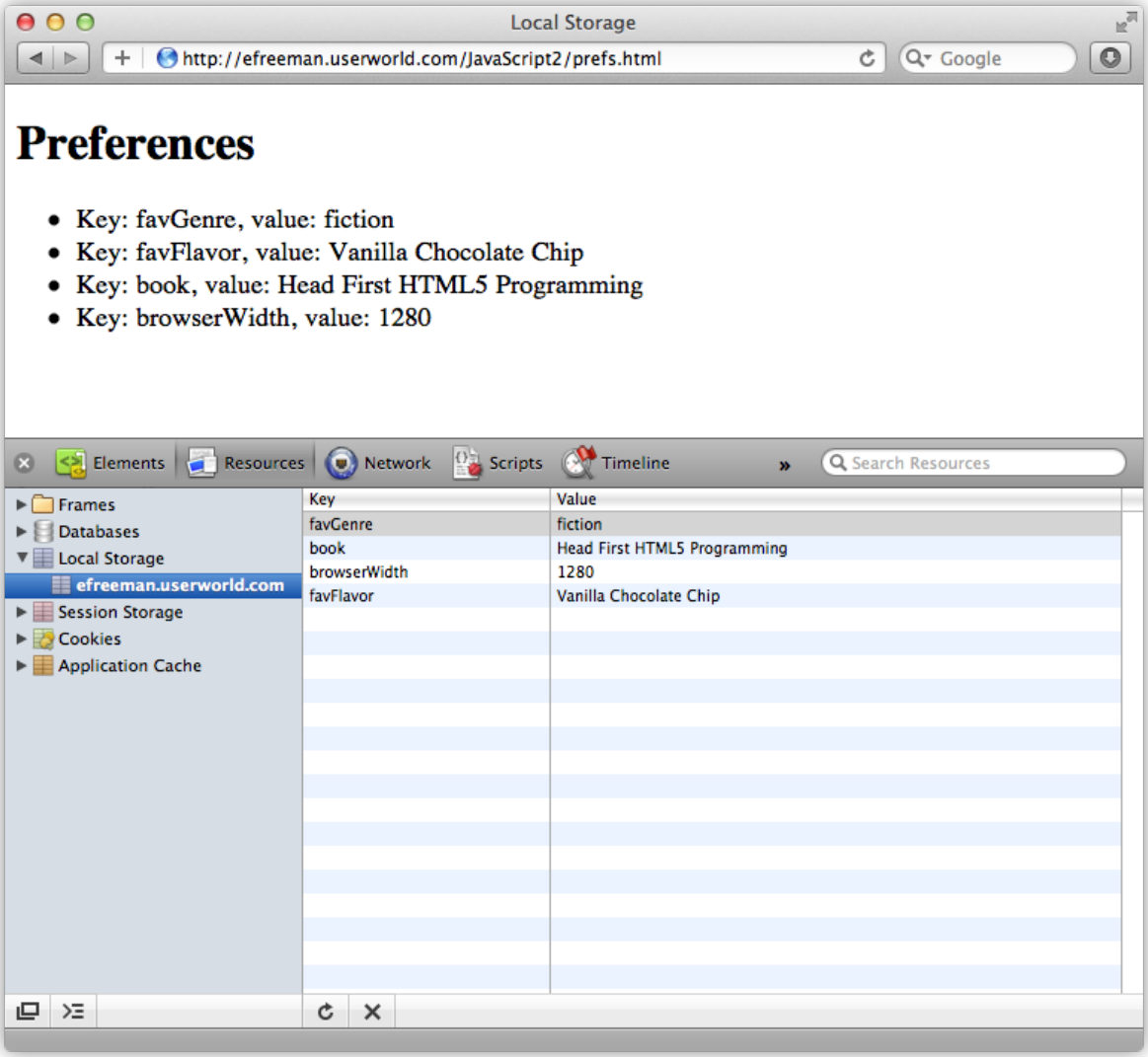
Using the localStorage Object with JavaScript

Let's expand our example a bit, so you can see a couple more ways to use the **localStorage** object with JavaScript. Modify **prefs.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var favoriteGenre = localStorage.getItem("favGenre");

    addItem("favGenre", "fiction");
    addItem("favFlavor", "Vanilla Chocolate Chip");
    addItem("book", "Head First HTML5 Programming");
    addItem("browserWidth", 1280);
}

function addItem(key, value) {
    localStorage.setItem(key, value);
    addToList(key, value);
}
function getItem(key) {
    var value = localStorage.getItem(key);
    alert("Item: " + key + ": " + value);
}
function addToList(key, value) {
    var ul = document.getElementById("items");
    var li = document.createElement("li");
    li.innerHTML = favoriteGenre"Key: " + key + ", value: " + value;
    ul.appendChild(li);
}
```

Save it, open your **prefs.html** file, and click Preview. You see four items in the list, and if you check your browser's Local Storage, you see the same four items there:



Try adding a few more items using this program by adding more calls, with different values, to the **addItem()** function. Reload the page. Do you see your new items? Try changing the values. Reload. Do you notice anything about the ordering of the items in the list, and the ordering in Local Storage?

In this program, we create an **addItem()** function to add a key and a value to Local Storage, and then call another function, **addToList()**, to add it to the page.

We also create a **getItem()** function to get an item from Local Storage, but we're not using it yet. Let's do that next. Now that you have a bunch of items in Local Storage, let's comment out the lines to add the items, and add a call to **getItem()** so we can see the values that are in Local Storage that way. Modify **prefs.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    //addItem("favGenre", "fiction");
    //addItem("favFlavor", "Vanilla Chocolate Chip");
    //addItem("book", "Head First HTML5 Programming");
    //addItem("browserWidth", 1280);
    getItem("book");
}

function addItem(key, value) {
    localStorage.setItem(key, value);
    addToList(key, value);
}
function getItem(key) {
    var value = localStorage.getItem(key);
    alert("Item: " + key + ": " + value);
}
function addToList(key, value) {
    var ul = document.getElementById("items");
    var li = document.createElement("li");
    li.innerHTML = "Key: " + key + ", value: " + value;
    ul.appendChild(li);
}
```

 Save it, open your **prefs.html** file, and click 

. You see an alert showing the book item you got from Local Storage using **getItem()**. Notice that you will *not* see the items in the list in the web page. Why? Because you're no longer calling **addToList()** which is called from **addItem()**. Try changing "book" to another key, like "browserWidth". You see an alert showing you the key and the value for each key you try. What happens if you try a key that doesn't exist in Local Storage?

Local Storage Stores Strings

Did you notice that the "browserWidth" item that we added has a numeric value, rather than a string value?

Currently, browser implementations of Local Storage only store string values. So how is our "browserWidth" value working? Make this change in **prefs.js** and then see if you can guess how it's working:

CODE TO TYPE:

```
window.onload = init;

function init() {
    //addItem("favGenre", "fiction");
    //addItem("favFlavor", "Vanilla Chocolate Chip");
    //addItem("book", "Head First HTML5 Programming");
    //addItem("browserWidth", 1280);
    getItem("book""browserWidth");
}

function addItem(key, value) {
    localStorage.setItem(key, value);
    addToList(key, value);
}
function getItem(key) {
    var value = localStorage.getItem(key);
    alert("Item: " + key + ": " + value + " (" + (typeof value) + ")");
}
function addToList(key, value) {
    var ul = document.getElementById("items");
    var li = document.createElement("li");
    li.innerHTML = "Key: " + key + ", value: " + value;
    ul.appendChild(li);
}
```

Save it, open your **prefs.html** file, and click Preview.

The localStorage object converted the number, **1280**, that we used as the value for the "browserWidth" key, to a string to store it in Local Storage. Then, when we get the value for "browserWidth" using **getItem()**, the value returned is a string, not a number, because that's how it was stored. (Note that we used a built-in JavaScript operator, **typeof**, to get the type of the primitive value returned from the call to **getItem()**. **typeof** is a unary operator: it takes one value, and returns its type. This operator works on all JavaScript *primitive* types, including numbers, strings, Booleans and undefined).

Try adding an item (using **addItem()**) that contains a Boolean value. For instance:

OBSERVE:
`addItem("isItCold", false);`

and then use **getItem()** to get the value and alert it. What is the type?

So, Local Storage uses strings for both its keys and its values. That means, if you store another value, like a number or a Boolean, you need to be prepared to convert it back to that type from a string when you retrieve the value from Local Storage later!

Iterating Through Local Storage

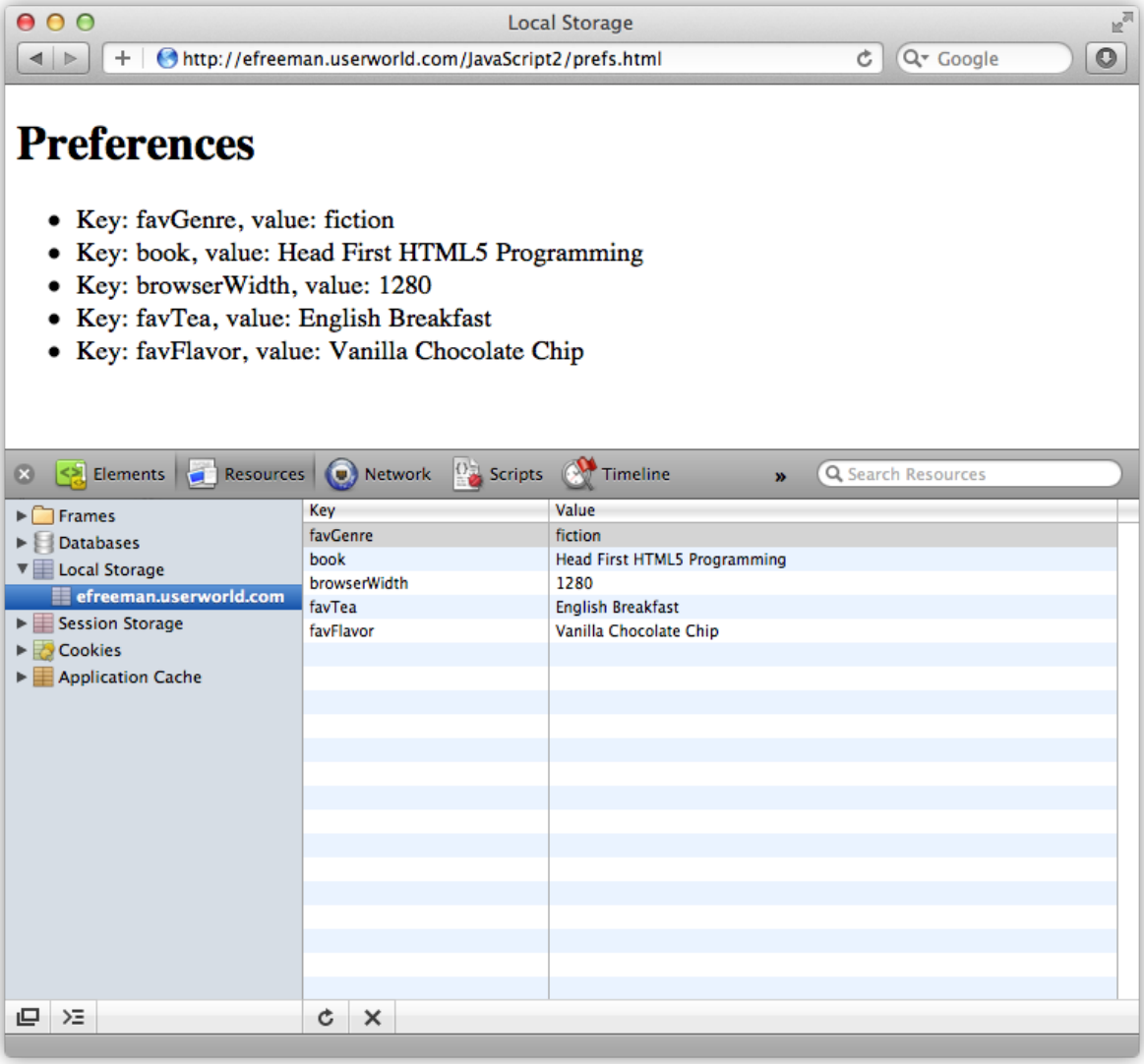
Have you thought about this: What if you want to write a program to list all the key/value pairs in Local Storage? What if there are hundreds of keys and values? Wouldn't it be a pain to have to identify each key individually?!

Indeed it would, and there is a better way. We can iterate through all the keys in Local Storage, without knowing in advance what those keys are. Modify **prefs.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    //addItem("favGenre", "fiction");
    //addItem("favFlavor", "Vanilla Chocolate Chip");
    //addItem("book", "Head First HTML5 Programming");
    //addItem("browserWidth", 1280);
    getItem("browserWidth");

    addItem("favTea", "English Breakfast");
    showAllPrefs();
}

function addItem(key, value) {
    localStorage.setItem(key, value);
    addToList(key, value);
}
function getItem(key) {
    var value = localStorage.getItem(key);
    alert("Item: " + key + ": " + value + " (" + (typeof value) + ")");
}
function addToList(key, value) {
    var ul = document.getElementById("items");
    var li = document.createElement("li");
    li.innerHTML = "Key: " + key + ", value: " + value;
    ul.appendChild(li);
}
function showAllPrefs() {
    for (var i = 0; i < localStorage.length; i++) {
        var key = localStorage.key(i);
        var value = localStorage.getItem(key);
        addToList(key, value);
    }
}
```

Save it, open **prefs.html**, and click Preview. You see every item you've stored in Local Storage in your list.



We added a new function, **showAllPrefs()**, to show us every item in Local Storage. This function looks through everything in Local Storage and adds it to the web page by calling **addToList()**. We call **showAllPrefs()** in **init()**, so we no longer need to call **addToList()** from **addItem()**. We just removed that call (otherwise, items we add to Local Storage will be shown twice on the page!).

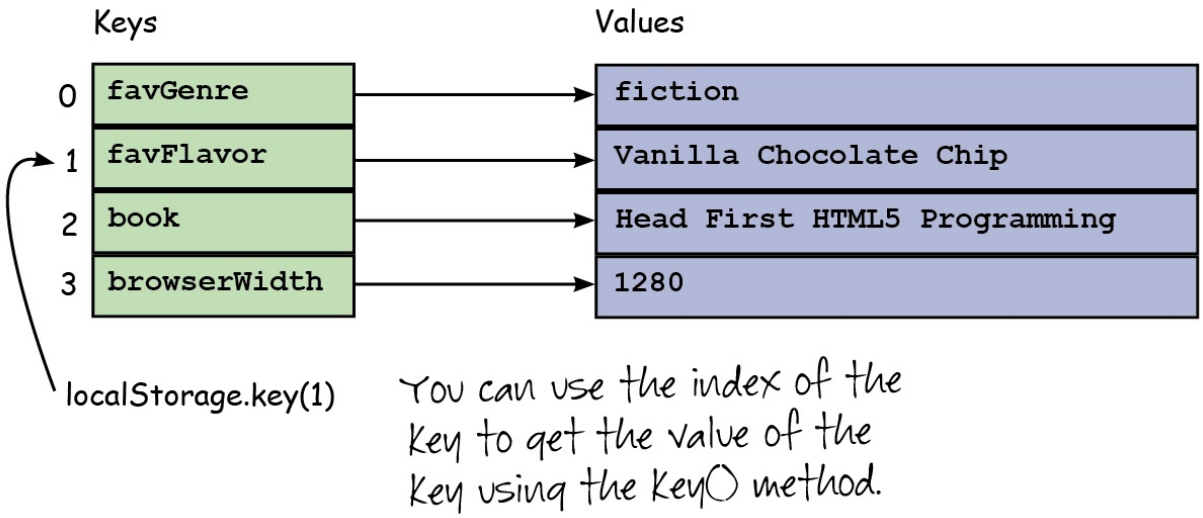
Let's step through **showAllPrefs()**, because we're using a couple of new things about the **localStorage** object we haven't talked about yet: the **length** property and the **key()** method.

OBSERVE:

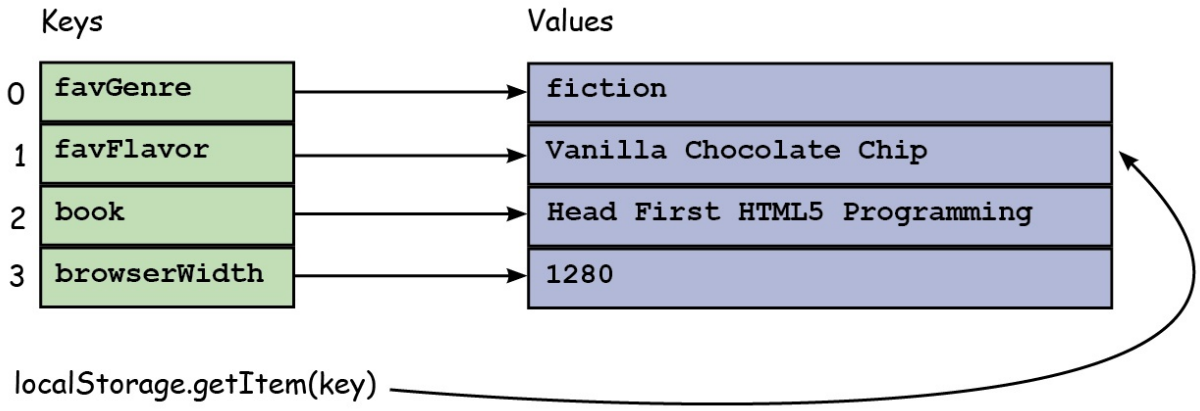
```
function showAllPrefs() {
    for (var i = 0; i < localStorage.length; i++) {
        var key = localStorage.key(i);
        var value = localStorage.getItem(key);
        addToList(key, value);
    }
}
```

First, remember that the **localStorage** object uses an *associative array* to store items in the browser's Local Storage. While an associative array doesn't have numerical indices like a regular array does, it does have a **length** property, which is just the number of key/value pairs in **localStorage**.

We can loop through all the keys in **localStorage** and access each key using the **key** method. Even though **localStorage** doesn't have an index like regular arrays, the **key** method makes it act a bit like it does, because the **key** method takes a number and returns a key.



localStorage.key(i) returns the string value of a key, like "book" or "browserWidth." Once you have that key, stashed in the variable **key**, you can use it to **retrieve the value associated with that key**, using **localStorage.getItem(key)**.



Finally, we pass both the **key** and the **value** to **addToList()** to add them to the page.

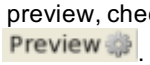
You're going to find that you'll frequently use this technique of iterating through all the keys in Local Storage to retrieve the values because you won't always know precisely what keys you have or how many you have.

Removing Items from Local Storage

Okay, you know how to add items to Local Storage, retrieve items from Local Storage, and even iterate through all the items in Local Storage. But how do you remove items from Local Storage using JavaScript? You know how to do it with the browser developer tools, but there is also a way to do it using code. You can remove one item at a time using **localStorage.removeItem()**, or you can remove everything at once, using **localStorage.clear()**. Modify **prefs.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    //addItem("favGenre", "fiction");
    //addItem("favFlavor", "Vanilla Chocolate Chip");
    //addItem("book", "Head First HTML5 Programming");
    //addItem("browserWidth", 1280);
    //addItem("favTea", "English Breakfast");

    removeItem("favFlavor");

    showAllPrefs();
}

function addItem(key, value) {
    localStorage.setItem(key, value);
}
function removeItem(key) {
    localStorage.removeItem(key);
}
function getItem(key) {
    var value = localStorage.getItem(key);
    alert("Item: " + key + ": " + value + " (" + (typeof value) + ")");
}
function addToList(key, value) {
    var ul = document.getElementById("items");
    var li = document.createElement("li");
    li.innerHTML = "Key: " + key + ", value: " + value;
    ul.appendChild(li);
}
function showAllPrefs() {
    for (var i = 0; i < localStorage.length; i++) {
        var key = localStorage.key(i);
        var value = localStorage.getItem(key);
        addToList(key, value);
    }
}
```

 Save it. Before you preview, check the items in your Local Storage using the browser developer tools. Then open **prefs.html** and click 

. If you check your Local Storage, the item with the key "favFlavor" should be gone. Try removing a couple more items from Local Storage by changing the key you pass to **removeItem()**, and reloading the page. What happens if you try to remove a key that doesn't exist?

Our **removeItem()** function uses the **localStorage.removeItem()** method to remove the item from Local Storage. The method takes one argument, the key to be deleted, and removes the key and its value from Local Storage.

To remove *all* the items in Local Storage, use the **localStorage.clear()** method:

CODE TO TYPE:

```
window.onload = init;

function init() {
    //addItem("favGenre", "fiction");
    //addItem("favFlavor", "Vanilla Chocolate Chip");
    //addItem("book", "Head First HTML5 Programming");
    //addItem("browserWidth", 1280);
    //addItem("favTea", "English Breakfast");

    removeItem("favFlavor");
    clearAllItems();
    showAllPrefs();
}

function addItem(key, value) {
    localStorage.setItem(key, value);
}
function removeItem(key) {
    localStorage.removeItem(key);
}
function clearAllItems() {
    localStorage.clear();
}
function getItem(key) {
    var value = localStorage.getItem(key);
    alert("Item: " + key + ": " + value + " (" + (typeof value) + ")");
}
function addToList(key, value) {
    var ul = document.getElementById("items");
    var li = document.createElement("li");
    li.innerHTML = "Key: " + key + ", value: " + value;
    ul.appendChild(li);
}
function showAllPrefs() {
    for (var i = 0; i < localStorage.length; i++) {
        var key = localStorage.key(i);
        var value = localStorage.getItem(key);
        addToList(key, value);
    }
}
```

Save it, open **prefs.html**, and click Preview. Now there are no items in your list, and if you check your Local Storage using the browser's developer tool, you'll see no items in your Local Storage.

Cookies (Not the Kind You Can Eat)

You may have heard of *browser cookies* before, and you may even be familiar enough with them to see that cookies and Local Storage have some similarities. Cookies are another mechanism for storing data in the browser. To create a cookie, you create a string that looks like this:

Cookie: favFlavor=VanillaChocolateChip; favTea=EnglishBreakfast; browserWidth=800

As you can see, a cookie has one or more keys and one or more values associated with those keys. So in that way, a cookie is quite similar to Local Storage. Applications that interact with a web server use cookies to store temporary data or personalize an experience, in much the same way you use Local Storage to do those things.

However, cookies have a few of big disadvantages:

- Cookies are sent back and forth to and from the server each time you make a request! So if your application communicates with a web server at oreillystudent.com, every cookie associated with oreillystudent.com will be sent to the server, and retrieved from the server, with each request. This adds overhead to each request (slowing down your application).
- Cookies are limited to 4K of data. Compare this to 5MB of data that each domain gets to store data in Local Storage. That's a huge difference!

- The JavaScript interface for interacting with cookies is a lot more difficult to use than the localStorage object.
- It's often more difficult for users to have visibility into what cookies are stored on their computers.

Cookies are still useful for certain types of applications, but it's likely that Local Storage will take over a lot of the functions that cookies have been used for until now.

We hope you've had some fun storing, retrieving, and removing items to and from your browser's Local Storage using the JavaScript **localStorage** object.

Two more things to know about Local Storage: first, Local Storage is part of the **Web Storage** part of the HTML specification. Most developers, however, just call it "Local Storage" rather than "Web Storage." But if you hear "Web Storage," it's likely to be referring to the Local Storage feature.

Along with the **localStorage** object, there is another built-in object you can use named **sessionStorage**. Session Storage is just like Local Storage except that all data that is stored in the browser goes away when you close the tab or window you've been using to interact with application. Session Storage is useful for storing very temporary items that you want to make sure go away when the user leaves the page.

In the next lesson, we'll update the Todo List Application to use Local Storage.

Updating the To-Do List Application for Local Storage

Lesson Objectives

When you complete this lesson, you will be able to:

- store objects in local storage.
- save to-do items in local storage.
- add an ID to a to-do item.
- get to-do items from local storage.
- use the substring() method to take any String and create a new String from it.

Storing Objects in Local Storage

In the previous lesson, you learned the basics of storing data in the browser, using Local Storage. In this lesson, we're going update our To-Do List application to store to-do items in Local Storage, rather than on the server.

You know how to store simple strings in Local Storage, and you know that you can store only strings in Local Storage (meaning you can't store numbers, objects, or other types in Local Storage). A to-do item is an object. Here's an example of one:

OBSERVE:

```
{
  task: "get milk",
  who: "Scott",
  dueDate: "today",
  done: false
}
```

Knowing that you can only store strings in Local Storage, can you think of how we might store to-do items? And what should we use as the keys for the to-do items?

We can use JSON to store objects such as our to-do items in Local Storage. Recall that JSON allows us to represent JavaScript objects as strings. Once we have a string representation of an object, we can store it in Local Storage just like any other string. Similarly, we can use JSON to retrieve the string from Local Storage and turn it back into an object.

Before we dive back into the To-Do List application, let's create a small example to store and retrieve JSON objects from Local Storage. Create a new HTML file as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
    <title>Store an Object in Local Storage</title>
    <meta charset="utf-8">
    <script src="storeObj.js"></script>
</head>
<body>

<h1>Store an object in Local Storage</h1>

</body>
</html>
```

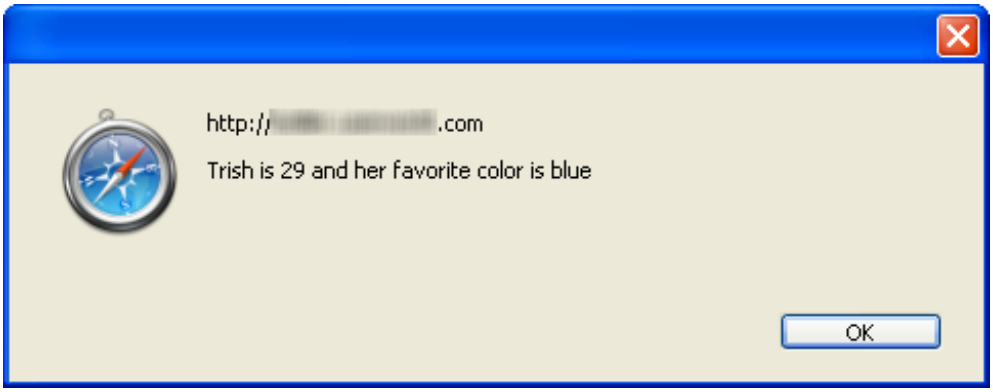
Save this file in your **/javascript2** folder as **storeObj.html**. We link to the file **storeObj.js** in the header—that's where all the real action is! Now we need some HTML so we can preview and look Local Storage using the browser developer console.

Create a new file and add the JavaScript as shown:

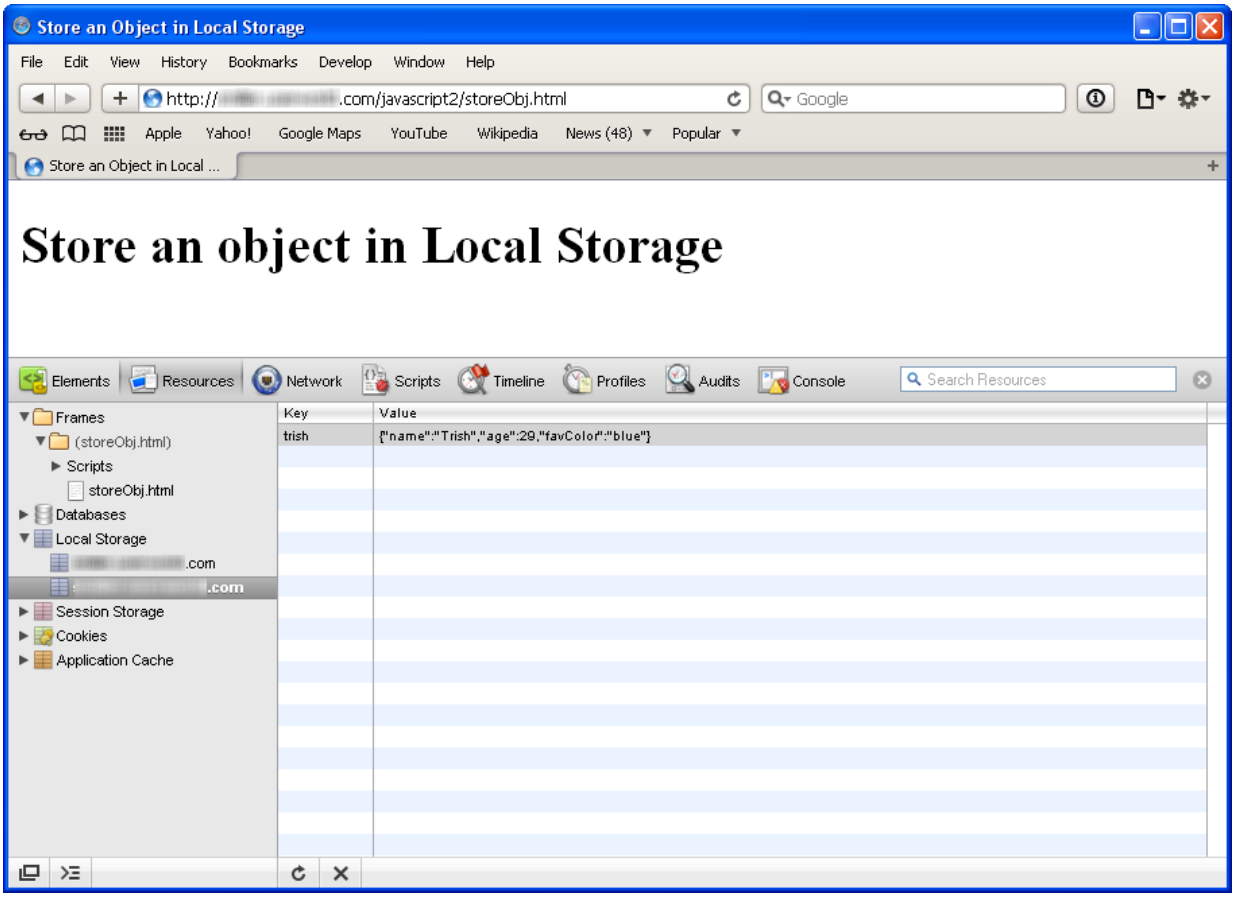
CODE TO TYPE:

```
window.onload = init;

function init() {
    var myObject = {
        name: "Trish",
        age: 29,
        favColor: "blue"
    };
    var myObjectJson = JSON.stringify(myObject);
    localStorage.setItem("trish", myObjectJson);
    var newMyObjectJSON = localStorage.getItem("trish");
    var newMyObject = JSON.parse(newMyObjectJSON);
    alert(newMyObject.name + " is " + newMyObject.age +
        " and her favorite color is " + newMyObject.favColor);
}
```

Save this file in your **/javascript2** folder as **storeObj.js**, open your **storeObj.html** file, click Preview. You see an alert like this:



Use your browser's developer console to inspect Local Storage. The Local Storage includes the item you just stored using this program:



Note You might see additional items in Local Storage if you've added anything to the Local Storage at the oreillystudent.com domain. Just make sure you see the "trish" object you just added!

Let's look at the JavaScript:

OBSERVE:

```
function init() {
    var myObject = {
        name: "Trish",
        age: 29,
        favColor: "blue"
    };
    var myObjectJson = JSON.stringify(myObject);
    localStorage.setItem("trish", myObjectJson);
    var newMyObjectJSON = localStorage.getItem("trish");
    var newMyObject = JSON.parse(newMyObjectJSON);
    alert(newMyObject.name + " is " + newMyObject.age +
        " and her favorite color is " + newMyObject.favColor);
}
```

This init() function:

- creates an object named myObject.
- converts that object to JSON (a string) using the JSON object's stringify() method.
- stores the JSON version of the object in Local Storage, with the key "trish", using the localStorage object's setItem() method.
- retrieves a JSON string from Local Storage using the same key, "trish," using the localStorage object's getItem() method, and saving it in a new variable, newMyObjectJSON.
- creates newMyObject from that JSON string using the JSON object's parse() method.
- displays the values in newMyObject using dot notation, and alerts the values so you can see them (newMyObject has exactly the same structure as our original myObject, so we can do

this).

So now that you know how to store and retrieve an object from Local Storage, let's go back to the To-Do List application and modify it to use Local Storage instead of Ajax to store to-do items.

Saving To-Do Items in Local Storage

Even though we use keys that relate to the content of the items we're storing (for example, "favGenre" and "browserWidth"), it's still difficult to figure out which items in Local Storage belong to which application. So, unless you use a string in your key that indicates that a particular item belongs to a specific application, how will your application know which items it can use? Remember that more than one application can use Local Storage at any given domain, and all these applications will be mixed together in Local Storage; we need a way to identify the items that belong to the To-Do List application.

For now, let's create a key for each to-do item from a unique id and a string that represents the application. We'll create the unique id for each to-do item as that item is created (that is, when you submit a new to-do item using the To-Do List form), store that id (along with the other to-do list information) in the Todo object, and then use that id when we create the key to store the item in Local Storage. Let's update the code and then we'll go over each change.

Note We'll continue to update the **todo.html** and **todo.js** files. You might want to make copies of the files to save your previous work.

Modify **todo.js** as shown:

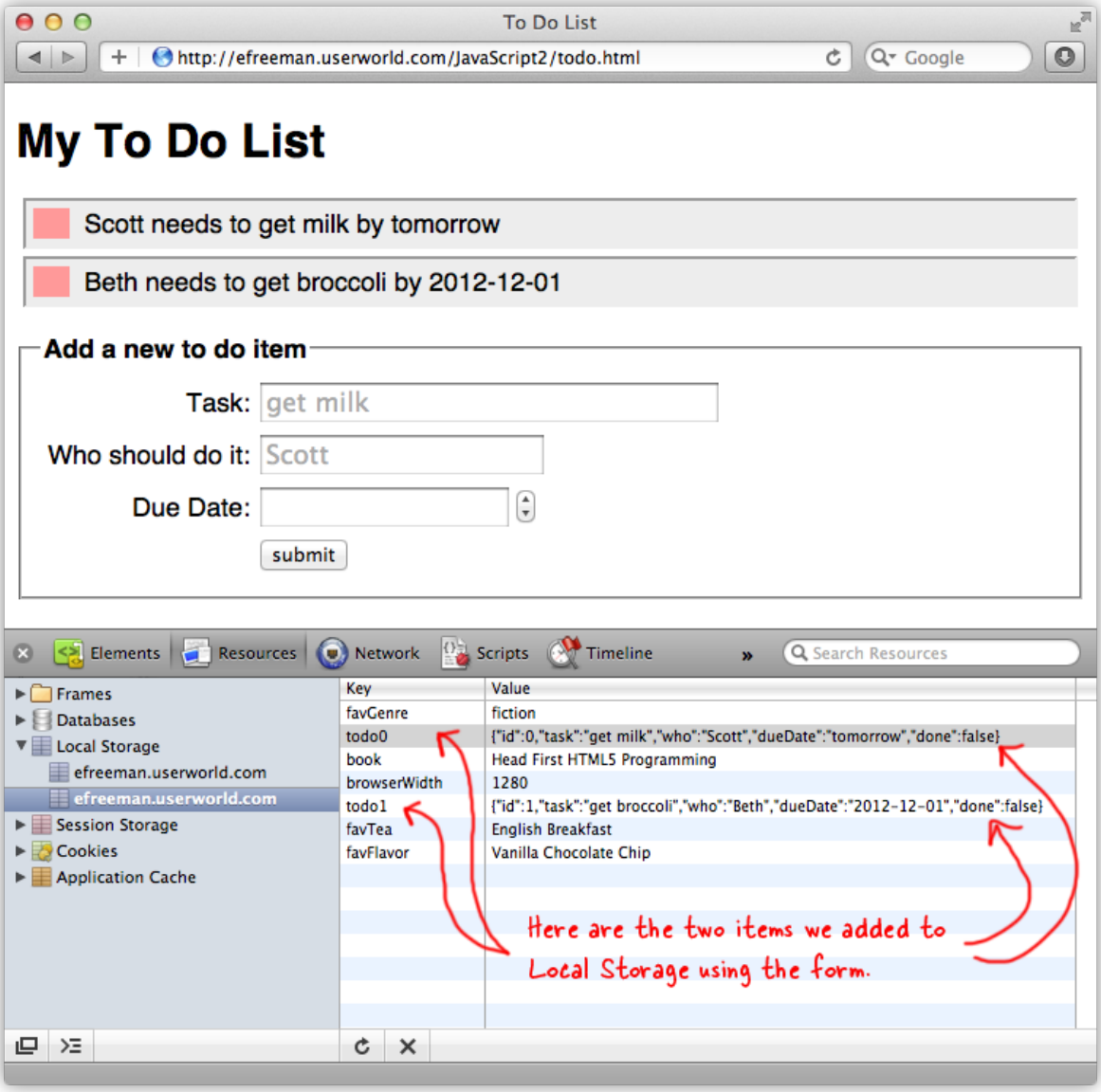
CODE TO TYPE:

```
function Todo(id, task, who, dueDate) {
    this.id = id;
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;

    getTodoData();
}

function getTodoData() {
    var request = new XMLHttpRequest();
    request.open("GET", "todo.json");
    request.onreadystatechange = function() {
        if (this.readyState == this.DONE && this.status == 200) {
            if (this.responseText) {
                parseTodoItems(this.responseText);
                addTodosToPage();
            }
            else {
                console.log("Error: Data is empty");
            }
        }
    };
    request.send();
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
}
function addTodosToPage() {
    var ul = document.getElementById("todoList");
    var listFragment = document.createDocumentFragment();
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = createNewTodo(todoItem);
        listFragment.appendChild(li);
    }
    ul.appendChild(listFragment);
}
function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = createNewTodo(todoItem);
    ul.appendChild(li);
    document.forms[0].reset();
}
```

```
function createNewTodo(todoItem) {
    var li = document.createElement("li");
    var spanTodo = document.createElement("span");
    spanTodo.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;

    var spanDone = document.createElement("span");
    if (!todoItem.done) {
        spanDone.setAttribute("class", "notDone");
        spanDone.innerHTML = "     ";
    }
    else {
        spanDone.setAttribute("class", "done");
        spanDone.innerHTML = " &#10004; ";
    }

    li.appendChild(spanDone);
    li.appendChild(spanTodo);
    return li;
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    var id = todos.length;
    var todoItem = new Todo(id, task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
    saveTodoData();
    saveTodoItem(todoItem);
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}

function saveTodoItem(todoItem) {
    if (localStorage) {
        var key = "todo" + todoItem.id;
        var item = JSON.stringify(todoItem);
        localStorage.setItem(key, item);
    }
    else {
        console.log("Error: you don't have localStorage!");
    }
}

function saveTodoData() {
    var todoJSON = JSON.stringify(todos);
    var request = new XMLHttpRequest();
    var URL = "save.php?data=" + encodeURI(todoJSON);
    request.open("GET", URL);
    request.setRequestHeader("Content-Type",
                             "text/plain;charset=UTF-8");
    request.send();
}
```

Save it. We removed the functions to save and get the to-do items (**saveTodoData()** and (**getTodoData()**) from the server. We replaced the function to save the data with a new function, **saveTodoItem()**, which takes one to-do item and saves it to Local Storage.

Because we are still using the same JSON format we used before (in the Ajax version), we don't have to change much of the other code. That's convenient. Before we walk through the details, open **todo.html** and click Preview. Try adding a few to-do items using the form:



When you look at Local Storage using your browser, you see the new items you added. If you had other items in Local Storage already, the new to-do items will be mixed in with those other items. You'll be able to tell which items belong to the To-Do List application because the keys all begin with "todo."

Right now we are *saving* the items we enter using the form in Local Storage (we're not retrieving those items when we first load the page), so you'll see new items being added to Local Storage and to the page as you add them using the form. If you reload the page, you won't see those items in your list yet. We'll get to that shortly.

Adding an ID to a Todo Item

We had to add an **id** property to the **Todo** object, so each to-do item could store a unique id:

OBSERVE:

```
function Todo(id, task, who, dueDate) {
    this.id = id;
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}
```

Once we have this new **id** property, we need to update the code where we create the **Todo** object, to pass in a unique id. But what do we use for that **unique id**?

OBSERVE:

```
function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    var id = todos.length;
    var todoItem = new Todo(id, task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
    saveTodoItem(todoItem);
}
```

Each time we create a new **Todo** object representating a to-do item, we're adding that item to the **todos** array. Each time we add a new item to the array, the length of the array increases. Because of that, we can use the **current length of the array** as a **unique id**. We retrieve the length of the array with **todos.length**. Once we have an **id**, which in this case is just a number, we can use that **id** when we **create a new Todo object**. Next, we **add the new Todo item to the array, and to the page**, and then **save the new item** using our new function **saveTodoItem()**:

OBSERVE:

```
function saveTodoItem(todoItem) {
    if (localStorage) {
        var key = "todo" + todoItem.id;
        var item = JSON.stringify(todoItem);
        localStorage.setItem(key, item);
    }
    else {
        console.log("Error: you don't have localStorage!");
    }
}
```

The **saveTodoItem()** function takes one argument, **todoItem**, which is a **Todo** object with a unique id in the **id** property. The function **checks to make sure that the user's browser has a localStorage object**, and if so, creates a **key** using the **string "todo" concatenated with the id of the todoItem**. So, if we pass in a **todo** item with an id of 1, then the **key** will be "todo1".

Now the **todoItem** is a **Todo** object, so we have to convert it to JSON before we can store it in Local Storage. We do that using the **JSON.stringify()** method. We now have a key and an item, so we can **store the item in Local Storage** using **localStorage.setItem()**.

If the user's browser doesn't have localStorage, we show an error message in the console. (In a more robust application you might want to have another option.)

Getting To-Do Items from Local Storage

Okay, we've got to-do items stored in Local Storage and they're being displayed on the page as you add them. We put the items in Local Storage so they'll be there when you reload the page (or want to access your items the next day or even a month from now).

We can use **localStorage.getItem()** to retrieve items from Local Storage, but we want to retrieve only items with keys that contain the string "todo." Because we get just the items from Local Storage when we load the page, we need to add all the items in Local Storage to the list you see on the page as well. We already have a function that does that: **addTodosToPage()**. It adds all the items in the **todos** array to the page. So, as we retrieve each item from Local Storage, we'll add it to the **todos** array, then we can call **addTodosToPage()** to add them to the page.

Modify **todo.js** as shown:

CODE TO TYPE:

```
function Todo(id, task, who, dueDate) {
    this.id = id;
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;

    getTodoItems();
}

function getTodoItems() {
    if (localStorage) {
        for (var i = 0; i < localStorage.length; i++) {
            var key = localStorage.key(i);
            if (key.substring(0, 4) == "todo") {
                var item = localStorage.getItem(key);
                var todoItem = JSON.parse(item);
                todos.push(todoItem);
           }
        }
        addTodosToPage();
    }
    else {
        console.log("Error: you don't have localStorage!");
    }
}

function parseTodoItems(todoJSON) {
    if (todoJSON == null || todoJSON.trim() == "") {
        return;
    }
    var todoArray = JSON.parse(todoJSON);
    if (todoArray.length == 0) {
        console.log("Error: the to-do list array is empty!");
        return;
    }
    for (var i = 0; i < todoArray.length; i++) {
        var todoItem = todoArray[i];
        todos.push(todoItem);
    }
}
function addTodosToPage() {
    var ul = document.getElementById("todoList");
    var listFragment = document.createDocumentFragment();
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = createNewTodo(todoItem);
        listFragment.appendChild(li);
    }
    ul.appendChild(listFragment);
}
function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = createNewTodo(todoItem);
    ul.appendChild(li);
    document.forms[0].reset();
}
```

```
function createNewTodo(todoItem) {
    var li = document.createElement("li");
    var spanTodo = document.createElement("span");
    spanTodo.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;

    var spanDone = document.createElement("span");
    if (!todoItem.done) {
        spanDone.setAttribute("class", "notDone");
        spanDone.innerHTML = "     ";
    }
    else {
        spanDone.setAttribute("class", "done");
        spanDone.innerHTML = " &#10004; ";
    }

    li.appendChild(spanDone);
    li.appendChild(spanTodo);
    return li;
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    var id = todos.length;
    var todoItem = new Todo(id, task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
    saveTodoItem(todoItem);
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}

function saveTodoItem(todoItem) {
    if (localStorage) {
        var key = "todo" + todoItem.id;
        var item = JSON.stringify(todoItem);
        localStorage.setItem(key, item);
    }
    else {
        console.log("Error: you don't have localStorage!");
    }
}
```

Save it, open **todo.html**, and click Preview. You see all the items that you had previously added in the page; you'll be able to add new ones. Give it a try. Remove some items from Local Storage using the browser console tool, and then reload and make sure that what you see in Local Storage matches what you see in the page.

Note You may need to refresh the view of Local Storage in your browser (using the browser tools) or use the JavaScript console to display the **localStorage** object after you add an item to see the updated values.

Let's walk through the changes we made. We add a function **getTodoItems()**, and call that function from the **init()** function, so it runs as soon as the page loads and you see all the items you have stored, as soon as you load the page.

OBSERVE:

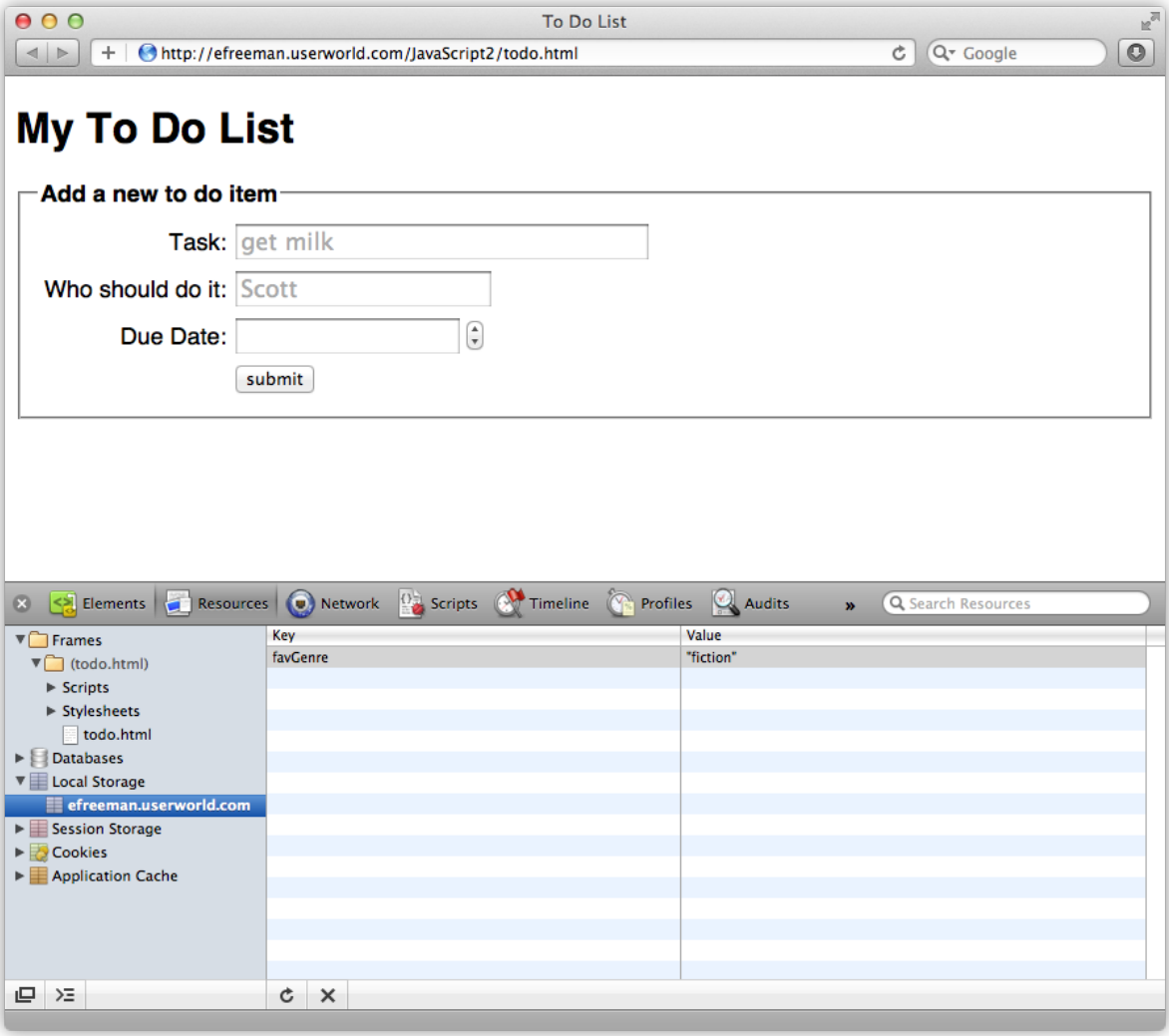
```
function getTodoItems() {
    if (localStorage) {
        for (var i = 0; i < localStorage.length; i++) {
            var key = localStorage.key(i);
            if (key.substring(0, 4) == "todo") {
                var item = localStorage.getItem(key);
                var todoItem = JSON.parse(item);
                todos.push(todoItem);
            }
        }
        addTodosToPage();
    }
    else {
        console.log("Error: you don't have localStorage!");
    }
}
```

In **getTodoItems()**, we check to make sure the **localStorage** object object exists— if it doesn't, display a message in the console.

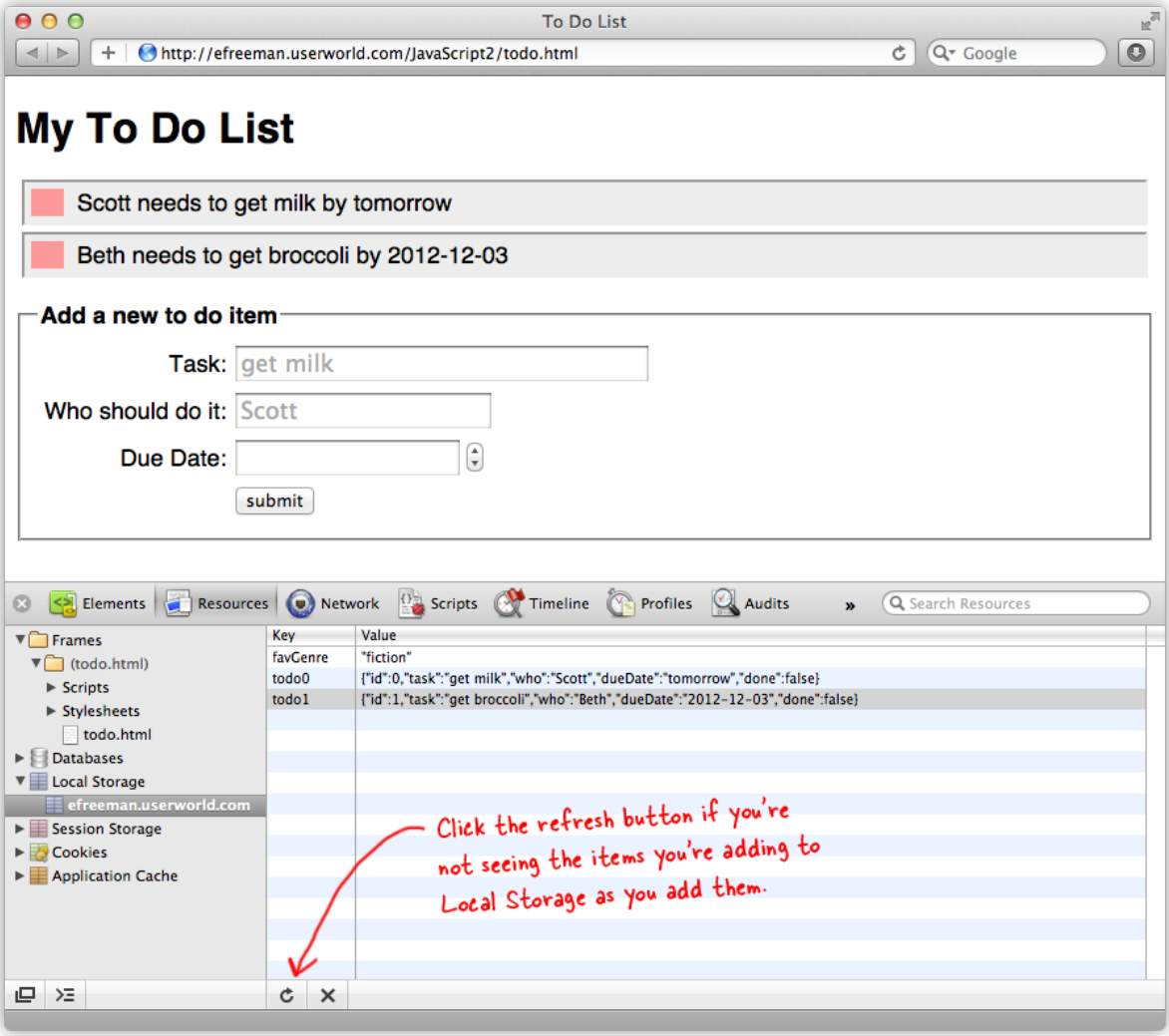
Next, we loop through all the items in Local Storage using the **length** property of the **localStorage** object, but this time, instead of adding every item to the page, we want to add only the items that have the string "todo" in the key. We **check to make sure that the key string contains the string "todo"**, using the String method **substring()** (we'll come back to this method shortly). If it does, then we get the item from Local Storage, and use **JSON to parse()** the string back into a Todo object, which we store in the variable **todoItem**. Then we **add the todoItem to the todos array**. Finally, after we've added all of the to-do items in Local Storage to the **todos array**, we call the **addTodosToPage()** function, which adds all the to-do items to the page.

We delete the **parseTodoItems()** function from the code. We don't need this function any more because we **parse each item in the loop** as we get the items from Local Storage, and we **add each item to the array** right there. Remember, we used **parseTodoItems()** in the Ajax version of the To-Do List application to parse the JSON we got back from the XMLHttpRequest.

If you have no "todo" items in Local Storage, when you load your page, you'll see:



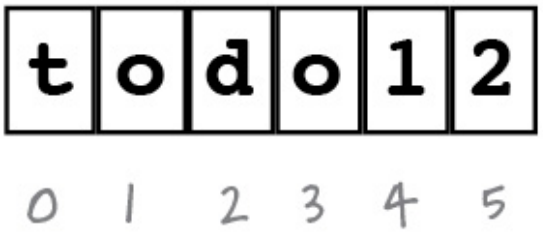
After adding some "todo" items, you'll see:



The String substring() method

In the function we just added, **getTodoItems()**, we used a String method, **substring()**. We haven't talked much about String methods yet; we have a whole lesson devoted to Strings later in this course. For now, let's take a closer look at the **substring()** method to see how it works, and how to use it to pull out only to-do items for the To-Do List application.

Think of a String as a set of multiple characters. It's a little bit like an array (but it's definitely NOT an array, so don't confuse the two), in that a character in a string holds a position, like this:



So the string "todo12" has a length of six; six separate characters that together make up the String. We capitalize **String** here because **String** is a special object in JavaScript. It's not quite like the objects you create in JavaScript, but similar in the sense that a String has properties (such as **length**) and methods (such as **substring()**).

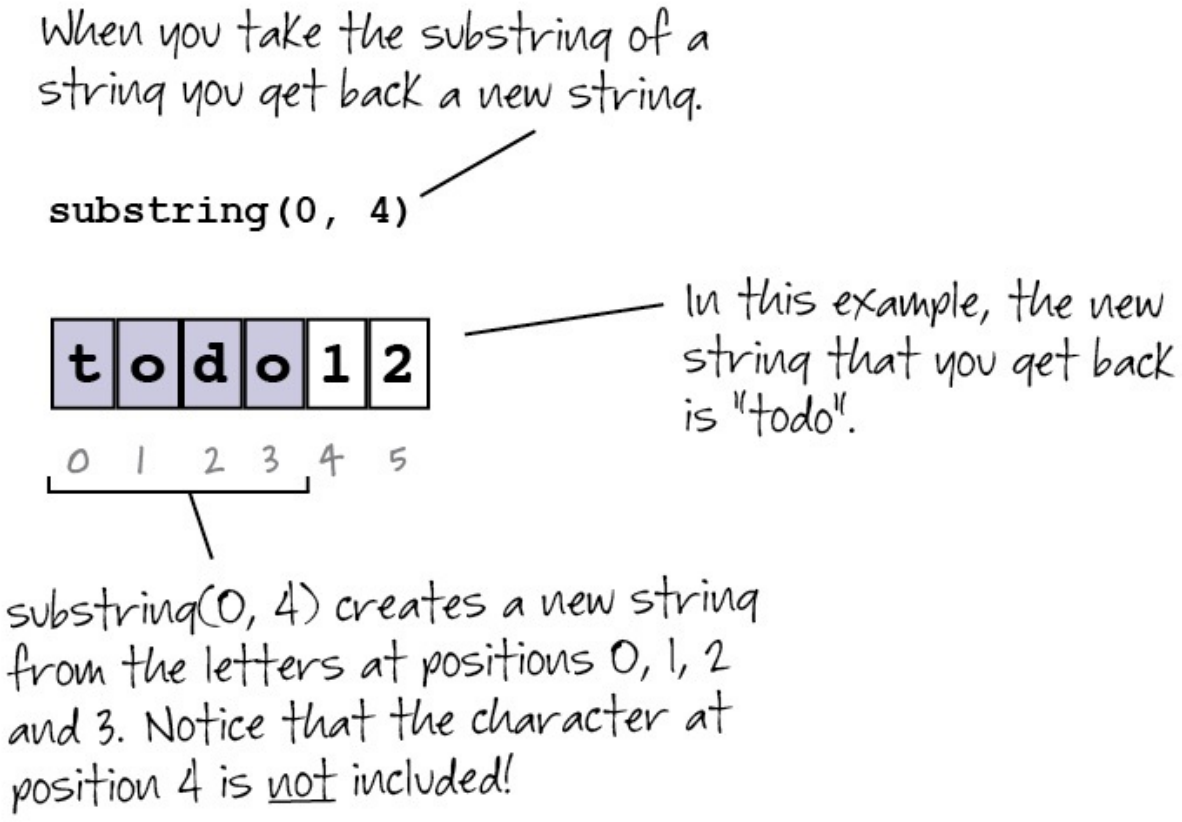
The **substring()** method allows you to take any String and create a new String from it by using a range of characters. The two arguments you pass to the **substring()** method are a **from** value and a **to** value, where **from** is the position

of the beginning character you want and **to** is the position of the next character *after* the ending character you want. For example, if you have the string "Hello World!" and you want to get the string "lo w", your code would look like this:

OBSERVE:

```
var str="Hello world!";
var mySubString = str.substring(3,7);
```

The variable **mySubString** will contain "lo w": all the characters beginning at position 3 and ending at (but *not including the character at*) position 7. (Remember that string positions start at 0, so "H" is at 0, "e" is at 1, and so on). For our Todo application, we can extract the "todo" portion of the key items using **substring()**, like this:



You can use the **substring()** method on any string you create in JavaScript. So, if we have a variable **key** that contains the string "todo12," we can call **substring()** on the **key** variable—**key.substring(0, 4)**—which returns the first four characters of the string (from positions 0, 1, 2, and 3), unless the length of the value in the key is less than four characters. In that case, you get all the letters in the string—so if the **key** contains the string "to," you get "to" as the result.

We can compare the result of calling **substring()** on **key** to a literal string, "todo," to see if they are equal, using the **==** operator:

OBSERVE:

```
if (key.substring(0, 4) == "todo") {
   ...
}
```

If they are, we know we've found a key for a to-do item. Because the **substring()** method returns a *new* string, we haven't changed the value of **key** at all. In our example, **key** still contains the string "todo12."

In this lesson, we updated the To-Do List application to use Local Storage instead of Ajax. That means the user's to-do items are stored in the browser they used to add items to their to-do list, using the To-Do List application.

We also learned the importance of choosing good keys for your applications that use Local Storage. In this example, we used the string "todo" and a unique id to create a key for to-do list items. It seems to be working well, but can you think of anything

that might cause our id / key scheme to fail? What would happen if you deleted a todo item from Local Storage, and then added a new item using the form without reloading the page first? Will the ids always match the position in the todos array when you get all the items from Local Storage, when you first load the page, using the function **getTodoItems()**? If not, could that cause a problem?

Think about those questions in preparation for the next lesson, take a crack at the quiz and project, and we'll see you in the next lesson!

Deleting To-Do List Items

Lesson Objectives

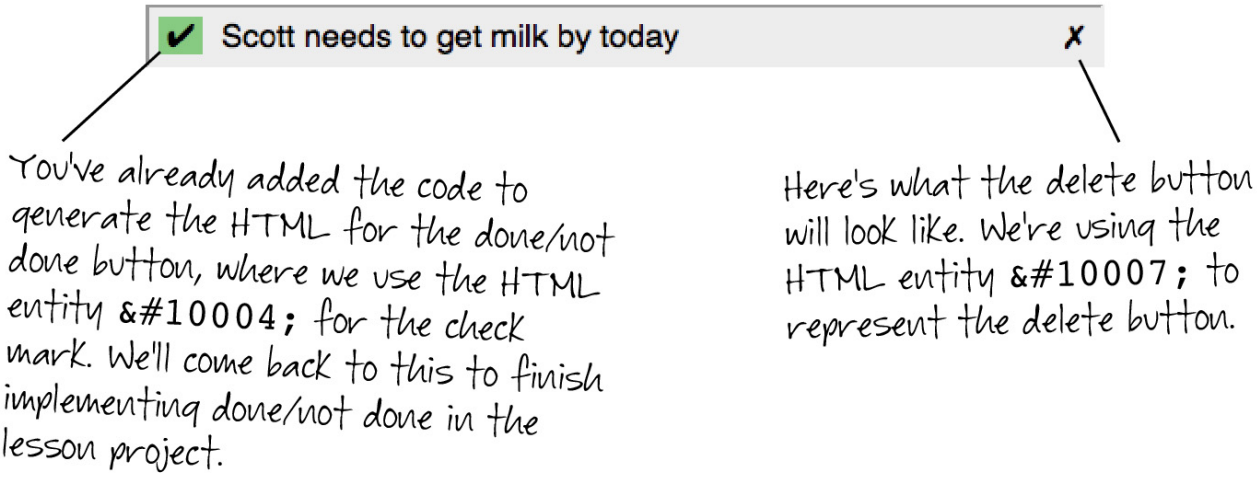
When you complete this lesson, you will be able to:

- add a delete button to a to-do item.
- generate an id that is guaranteed to be unique.
- delete items from local storage and to-do lists.
- move an ID to a parent element.

We've made a lot of progress with the To-Do List application, but wouldn't it be nice if we could delete items as well as add them? We also need a way to mark them as done. We'll add that functionality in this lesson and in the project later. Let's get started.

Adding a Way to Delete a To-Do Item

To add a delete button to a to-do item, we need to modify the code that generates the elements that represent each to-do item. We'll be modifying the <li> that contains the done/not done button we added in a previous lesson (using a element) and the text of the to-do item. We'll add a delete button at the right side of a list item and again, use a element to simulate a button, and use the HTML ✗ entity that looks like an "x" for delete. You could also use a small image for this if you wanted, but we'll keep it simple and just use the HTML entity.



We generate the HTML to represent a to-do item in the web page in the function **createNewTodo()**, so we'll update that code to add the delete button. We'll also add a function to handle the click on the delete button. This function is a click handler that will eventually implement the delete functionality; for now we'll just display some information in the console about the click.

Note We'll continue to update the **todo.html** and **todo.js** files. You might want to make copies of the files to save your previous work.

Ready? Here we go! Modify **todo.js** as shown:

CODE TO TYPE:

```
function Todo(id, task, who, dueDate) {
    this.id = id;
    this.task = task;
    this.who = who;
    this.dueDate = dueDate;
    this.done = false;
}

var todos = new Array();

window.onload = init;

function init() {
    var submitButton = document.getElementById("submit");
    submitButton.onclick = getFormData;

    getTodoItems();
}

function getTodoItems() {
    if (localStorage) {
        for (var i = 0; i < localStorage.length; i++) {
            var key = localStorage.key(i);
            if (key.substring(0, 4) == "todo") {
                var item = localStorage.getItem(key);
                var todoItem = JSON.parse(item);
                todos.push(todoItem);
            }
        }
        addTodosToPage();
    }
    else {
        console.log("Error: you don't have localStorage!");
    }
}

function addTodosToPage() {
    var ul = document.getElementById("todoList");
    var listFragment = document.createDocumentFragment();
    for (var i = 0; i < todos.length; i++) {
        var todoItem = todos[i];
        var li = createNewTodo(todoItem);
        listFragment.appendChild(li);
    }
    ul.appendChild(listFragment);
}
function addTodoToPage(todoItem) {
    var ul = document.getElementById("todoList");
    var li = createNewTodo(todoItem);
    ul.appendChild(li);
    document.forms[0].reset();
}

function createNewTodo(todoItem) {
    var li = document.createElement("li");
    var spanTodo = document.createElement("span");
    spanTodo.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;

    var spanDone = document.createElement("span");
    if (!todoItem.done) {
        spanDone.setAttribute("class", "notDone");
        spanDone.innerHTML = "     ";
    }
    else {
        spanDone.setAttribute("class", "done");
```

```
        spanDone.innerHTML = " &#10004; ";
    }

    var spanDelete = document.createElement("span");
    spanDelete.setAttribute("id", todoItem.id);
    spanDelete.setAttribute("class", "delete");
    spanDelete.innerHTML = " &#10007; ";

    spanDelete.onclick = deleteItem;

    li.appendChild(spanDone);
    li.appendChild(spanTodo);
    li.appendChild(spanDelete);

    return li;
}

function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    var id = todos.length;
    var todoItem = new Todo(id, task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
    saveTodoItem(todoItem);
}

function checkInputText(value, msg) {
    if (value == null || value == "") {
        alert(msg);
        return true;
    }
    return false;
}

function saveTodoItem(todoItem) {
    if (localStorage) {
        var key = "todo" + todoItem.id;
        var item = JSON.stringify(todoItem);
        localStorage.setItem(key, item);
    }
    else {
        console.log("Error: you don't have localStorage!");
    }
}

function deleteItem(e) {
    var id = e.target.id;
    console.log("delete an item: " + id);
}
```

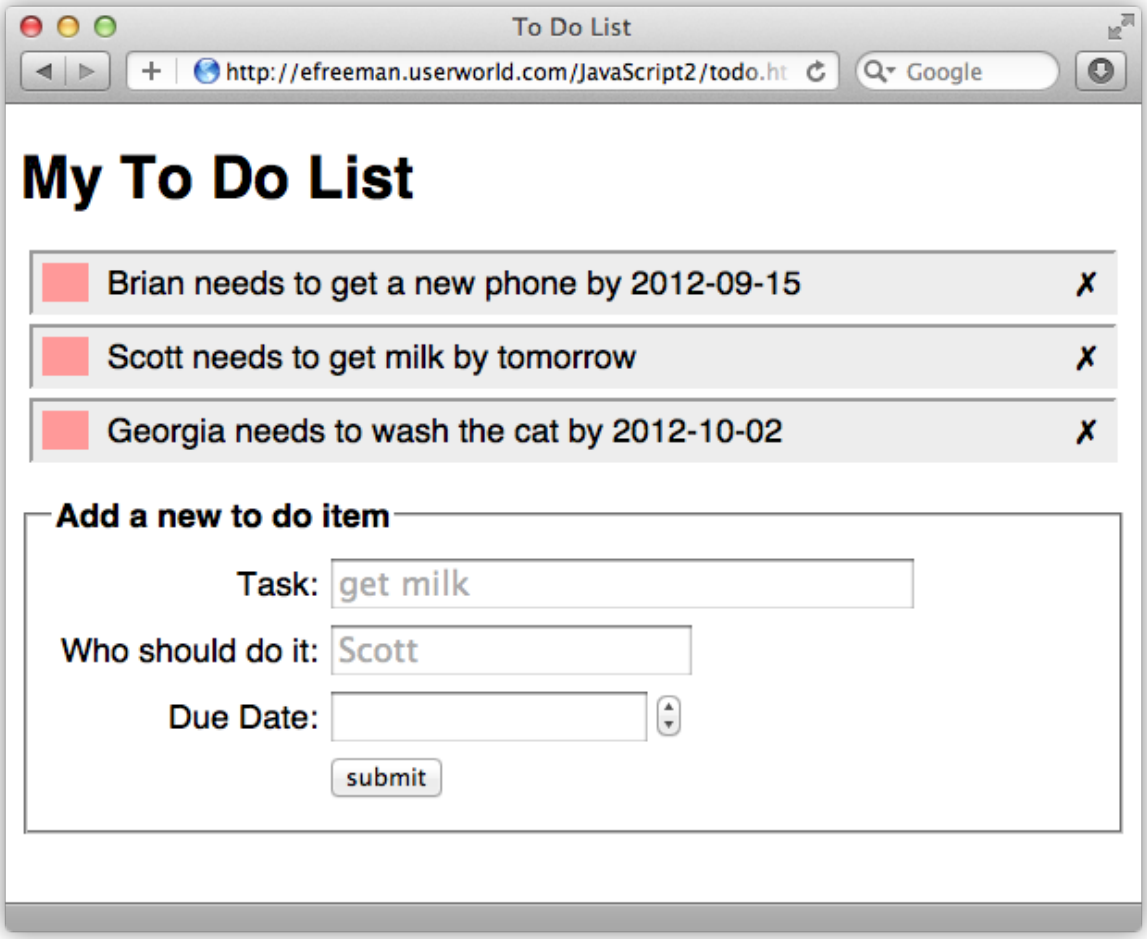
Save it, but don't preview yet; we need to add the CSS. Let's go ahead and do that, and then we'll come back to the code to step through it in detail.

CODE TO TYPE:

```
body {
    font-family: Helvetica, Ariel, sans-serif;
}
legend {
    font-weight: bold;
}
div.tableContainer {
    display: table;
    border-spacing: 5px;
}
div.tableRow {
    display: table-row;
}
div.tableRow label {
    display: table-cell;
    text-align: right;
}
div.tableRow input {
    display: table-cell;
}

ul#todoList {
    list-style-type: none;
    margin-left: 0px;
    padding-left: 0px;
}
ul#todoList li {
    position: relative;
    padding: 5px;
    margin: 5px;
    background-color: #ededed;
    border: 2px inset #ededed;
}
ul#todoList li span.notDone {
    margin-right: 10px;
    background-color: #FF9999;
    cursor: pointer;
}
ul#todoList li span.done {
    margin-right: 10px;
    background-color: #80CC80;
    cursor: pointer;
}
ul#todoList li span.delete {
    display: inline-block;
    position: absolute;
    right: 5px;
    cursor: pointer;
}
```

 Save it, open **todo.html**, and click 

. You'll see a delete "button" next to each to-do item in the page.



In the JavaScript, we update the **createNewTodo()** function to generate HTML to represent the delete "button" (which isn't a button in the sense of a form button, but rather an element you can click on to take some action):

OBSERVE:

```
var spanDelete = document.createElement("span");
spanDelete.setAttribute("id", todoItem.id);
spanDelete.setAttribute("class", "delete");
spanDelete.innerHTML = " &#10007; ";
```

We **create another element** just like we did for the done/not done button, and **use the entity ✗ to generate the ✗ character.**

As we did for the done/not done button, **we set the "class" attribute of the element, in this case, to "delete"**, so we can style it (we'll get to the style in just a moment).

Notice that we also **set the "id" attribute of the element**; that's so we know which element we've clicked on when we implement the **deleteItem()** function (you'll see how this works a bit later). The id of the item we want to delete is just the **todoItem.id** property. So if an item has the id 1, the element will look like this:

OBSERVE:

```
<span id="1" class="delete">&#10007;</span>
```

Now, the list item representing the to-do item has three elements in it. The generated HTML will look something like this:

OBSERVE:

```
<li>
  <span class="notDone">     </span>
  <span>TEXT OF TO DO ITEM</span>
  <span id="1" class="delete">&#10007;</span>
</li>
```

To style the delete button so it sits all the way to the right of the to-do item, we position it using CSS positioning:

OBSERVE:

```
ul#todoList li {
    position: relative;
    padding: 5px;
    margin: 5px;
    background-color: #ededed;
    border: 2px inset #ededed;
}
ul#todoList li span.notDone {
    margin-right: 10px;
    background-color: #FF9999;
    cursor: pointer;
}
ul#todoList li span.done {
    margin-right: 10px;
    background-color: #80CC80;
    cursor: pointer;
}
ul#todoList li span.delete {
    display: inline-block;
    position: absolute;
    right: 5px;
    cursor: pointer;
}
```

We can position the delete button using **absolute positioning**, which allows us to specify the number of pixels from the **right** of the list item that we want the delete button to appear. We used **5px** to leave a little space between the button and the edge of the list item.

But, to position an element absolutely relative to another element (rather than the page), we need to position its *parent element* as well. The parent element of the that represents the delete button is the <li> element. We can position the <li> element so that it goes with the flow of the page using **position: relative**. So the <li> element is **positioned relative** to where it would be normally, and then the delete is **positioned absolutely** within the <li> element, **5px** from the right.

Finally, we added a **property** that changes the **cursor to a pointer**. Now when you mouse over the done/not done button or the delete button, you'll see a hand pointer () rather than the regular mouse pointer, which provides visual feedback that indicates that you're hovering over something on which you can click.

The Delete Button Click Handler

Now that you know how the delete button is created with HTML and styled with CSS, let's take another look at the JavaScript code where we added a click handler for the delete button:

OBSERVE:

```
var spanDelete = document.createElement("span");
spanDelete.setAttribute("id", todoItem.id);
spanDelete.setAttribute("class", "delete");
spanDelete.innerHTML = " &#10007; ";

spanDelete.onclick = deleteItem;
```

After creating the element that holds the delete button, we **add a click handler to it** by setting the

onclick property to the function **deleteItem()**. For now, the **deleteItem()** function is pretty straightforward:

OBSERVE:

```
function deleteItem(e) {
    var id = e.target.id;
    console.log("delete an item: " + id);
}
```

The **deleteItem()** function has one parameter, **e**, that is the **click event** data. When you click on an element with a click handler, the event data is always passed into the click handler function; it contains data about the click, including the **target**, which is the **element that was clicked on**. In our case, that element is the "delete" . We know that the element has an **id** attribute (because we added it!) set to the id of the to-do item, so we can access that id using the **id** property of the element that is stored in the **target** property.

Note We could also access the id attribute using the **getAttribute()** method, like this: **var id = e.target.getAttribute("id");**.

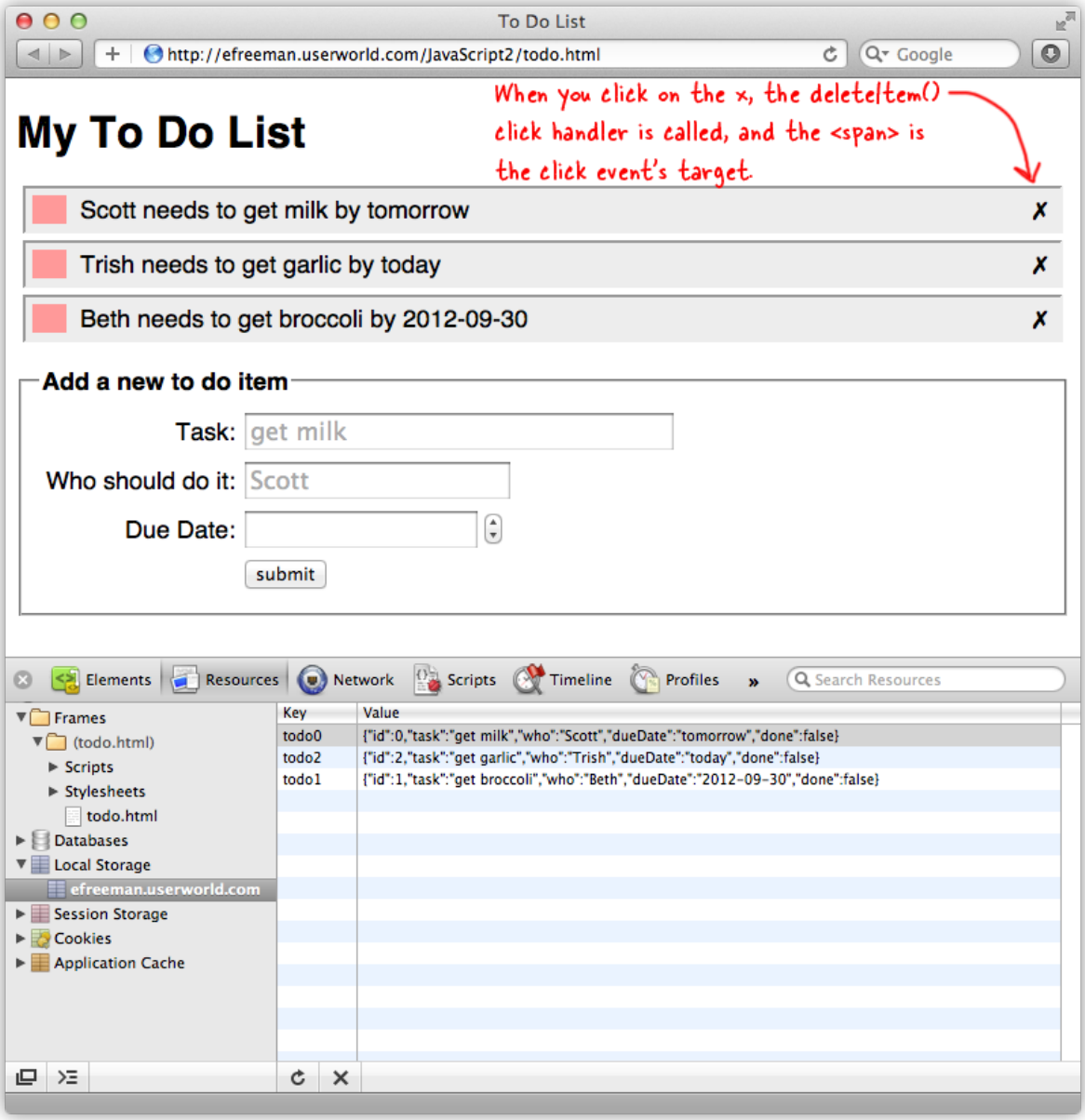
Once we have the **id** of the to-do item we want to delete, we display the id to the console so we can verify that it's correct.

Try it!

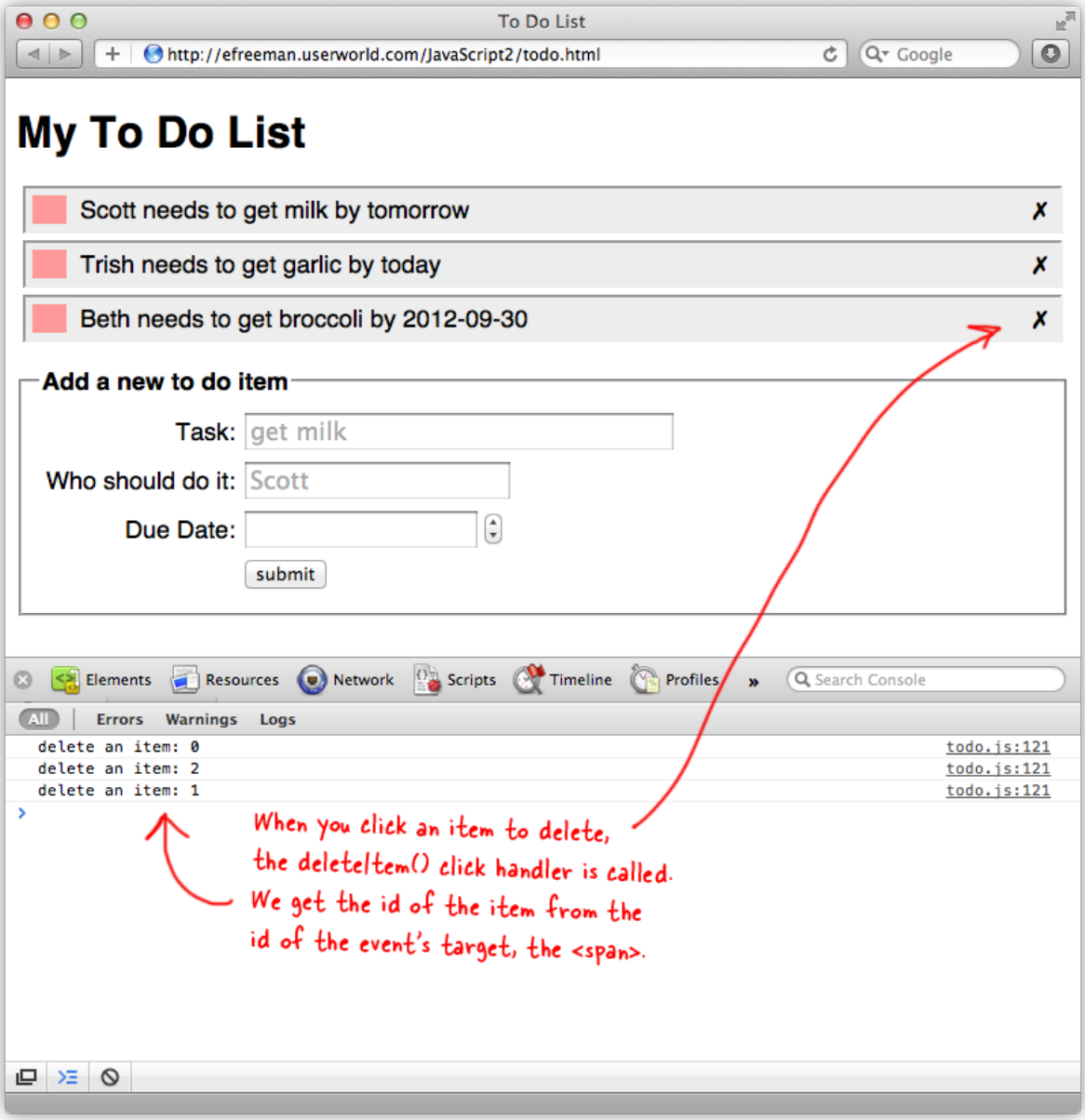
Preview **todo.html** again, open the Developer tools for your browser so you can see the JavaScript console, and try deleting a to-do item. It doesn't go away, because we haven't implemented **deleteItem()** yet, but the message in the console will indicate which item was clicked. Try clicking on several different items. Make sure the ids you see in the console are correct by comparing them with the ids you see in Local Storage.

So, what did we do?

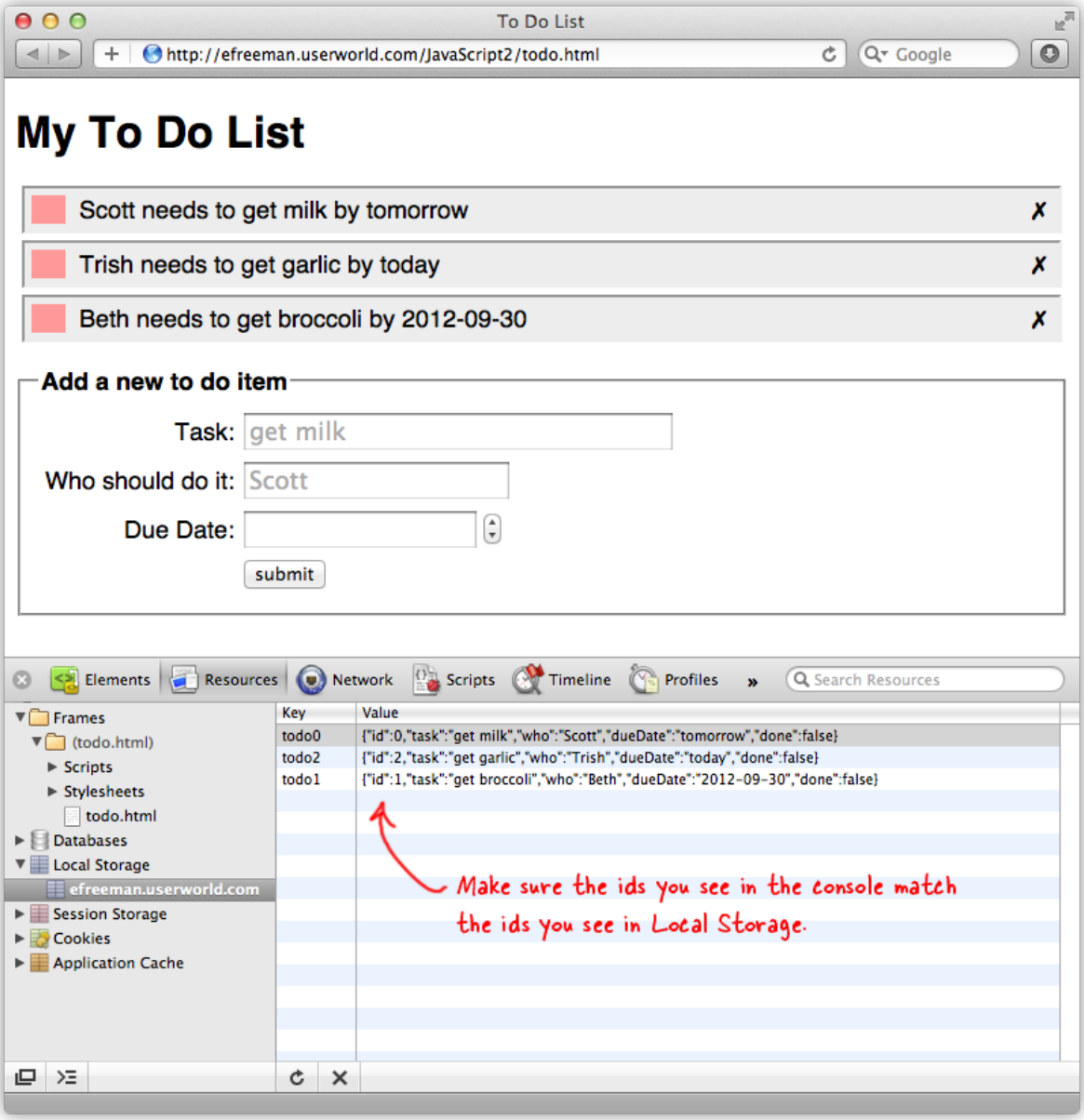
First, we added the HTML for the delete "button" to the HTML, and set the id attribute of the element to the id of the to-do item:



We also added a click handler so that when you click on the delete button, the click handler is called. The target of the event that's passed to the click handler is the element you clicked on, which is the representing the delete button. You can get the id of the to-do item to delete from the id of the element:



That id should match the id of the element in Local Storage:



We're *almost* ready to implement **deleteItem()**.

A Problem With Our ID Scheme

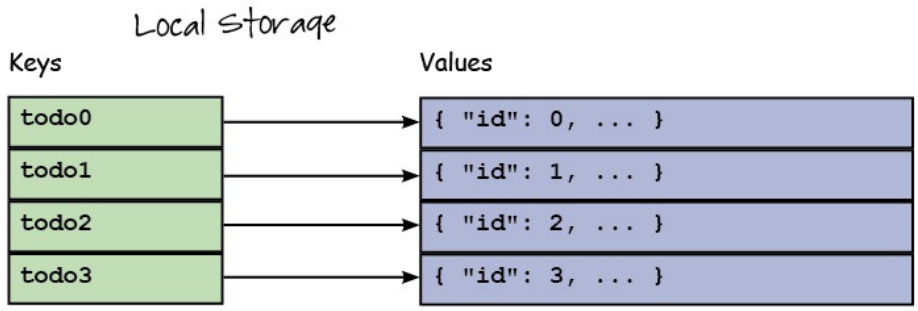
Before we implement the **deleteItem()** function, we need to look over our id scheme; it's going to have some problems once we add delete functionality. Did you think about this at the end of the previous lesson? Did you figure out why it won't work when we can delete items from the to-do list?

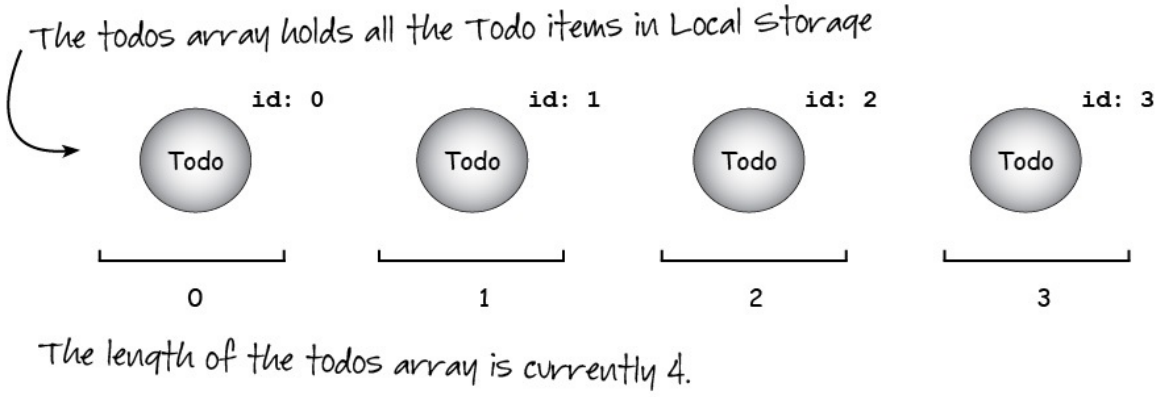
Remember that we add all to-do items to the **todos** array, when we first load the page (items already in Local Storage), and also as we add new items using the form. We use the length of the **todos** array to create a unique id for each **Todo** object that we create, like this (in the **getFormData()** function):

OBSERVE:

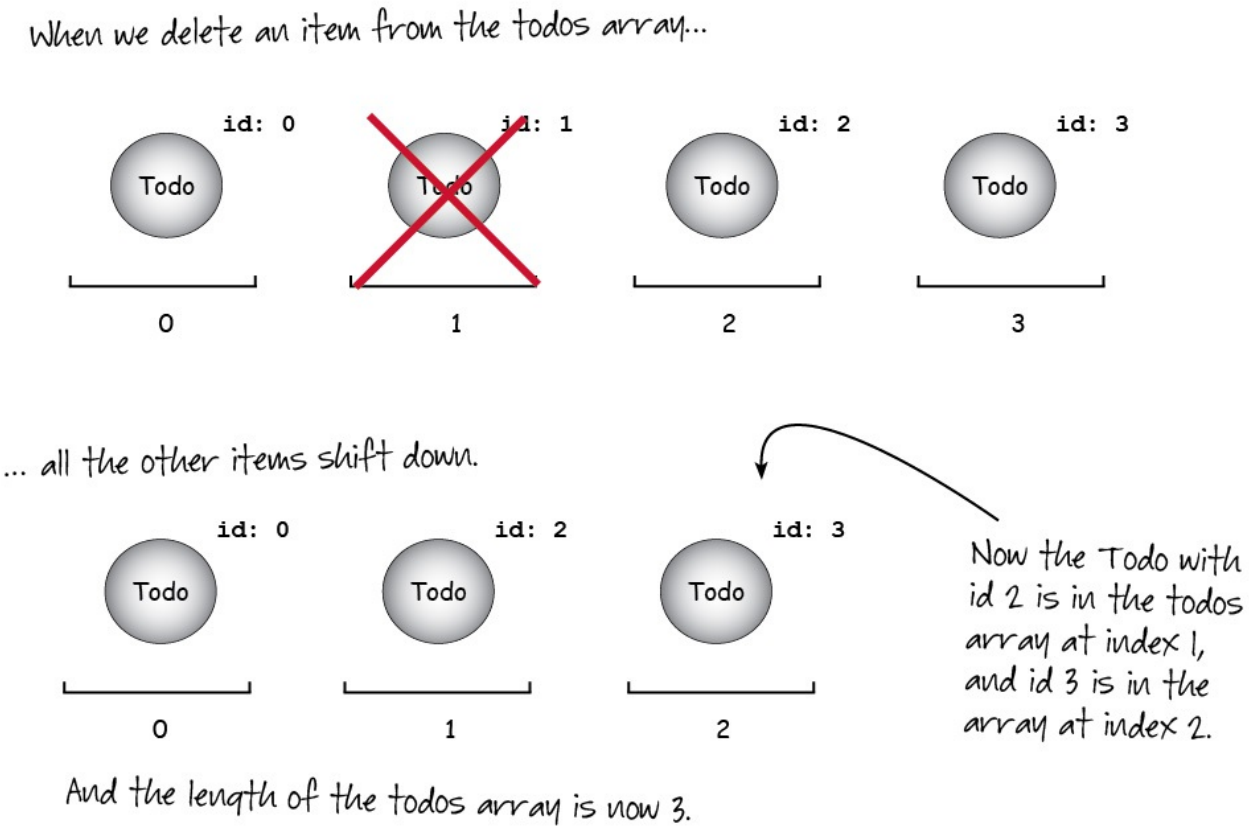
```
var id = todos.length;
```

...so that the id of the **Todo** matches the index of the item in the **todos** array:



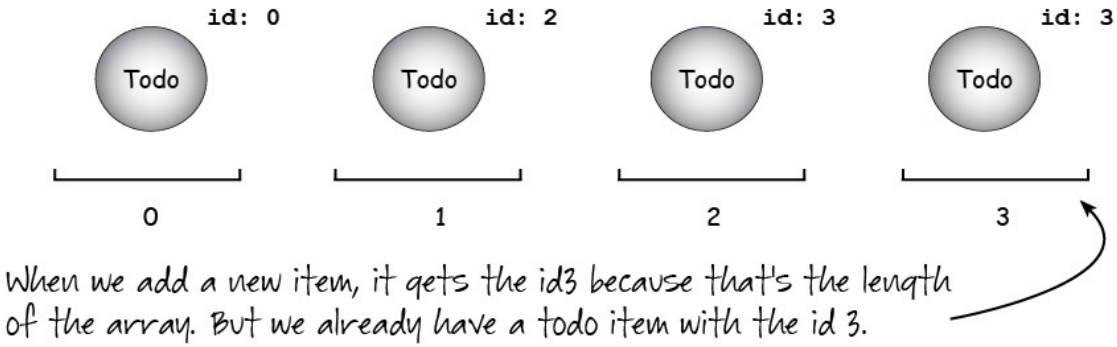


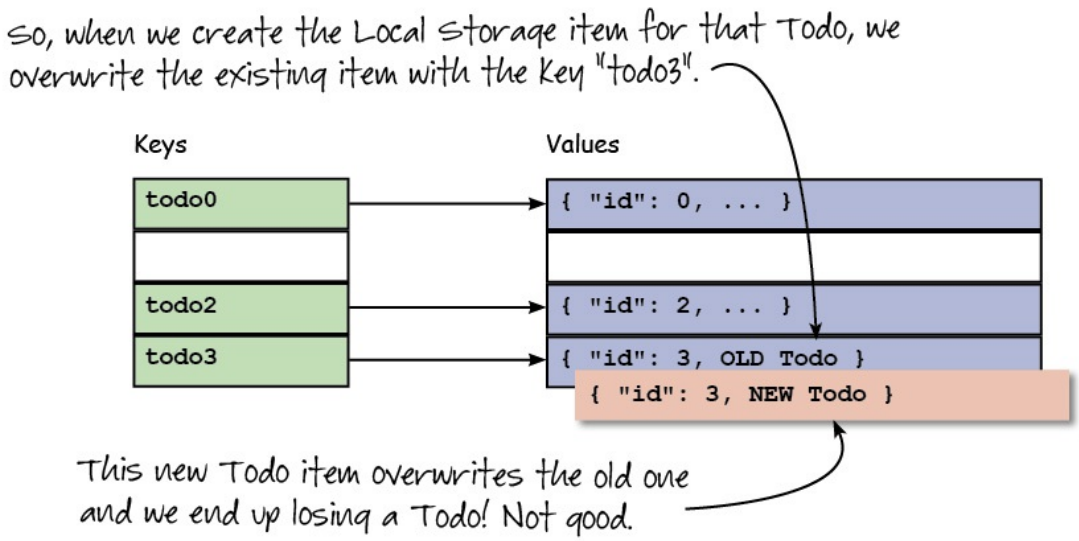
When we delete an item, we'll delete it from three places: Local Storage, the web page, and the **todos** array. Let's say we decide to delete the item with id 1. When we delete it from the array, the other items in the array shift down, so now the **Todo** with id 2 is in the array at index 1, the **Todo** with id 3 is in the array at index 2, and the length of the array is reduced by one; in this example, the length is now 3 (before the delete, the length was 4):



The trouble arises when we go to add a new **Todo**. Because the length of the array is 3, the id of the new item is set to 3, but we already have an item with the id 3. Even so, we add it to the array with no problem; when we add it to Local Storage, we create the key to store it in Local Storage using the **Todo** item's id and the string "todo", getting "todo3" in this example. However, there is already a "todo3" key with a value in Local Storage, and you know when happens

when you use the same key to store a new value in Local Storage. Yup, you *overwrite* the existing value, which means you just lost a to-do item on your list!





So, we need another way to create unique ids for our **Todo** items so that we have unique keys for each item in Local Storage. Can you think of how we might do that?

An ID Scheme Using Time

We need a way to generate an id that is guaranteed to be unique. To truly guarantee uniqueness is a complex task; but we can get close enough for this application by using the time expressed in milliseconds.

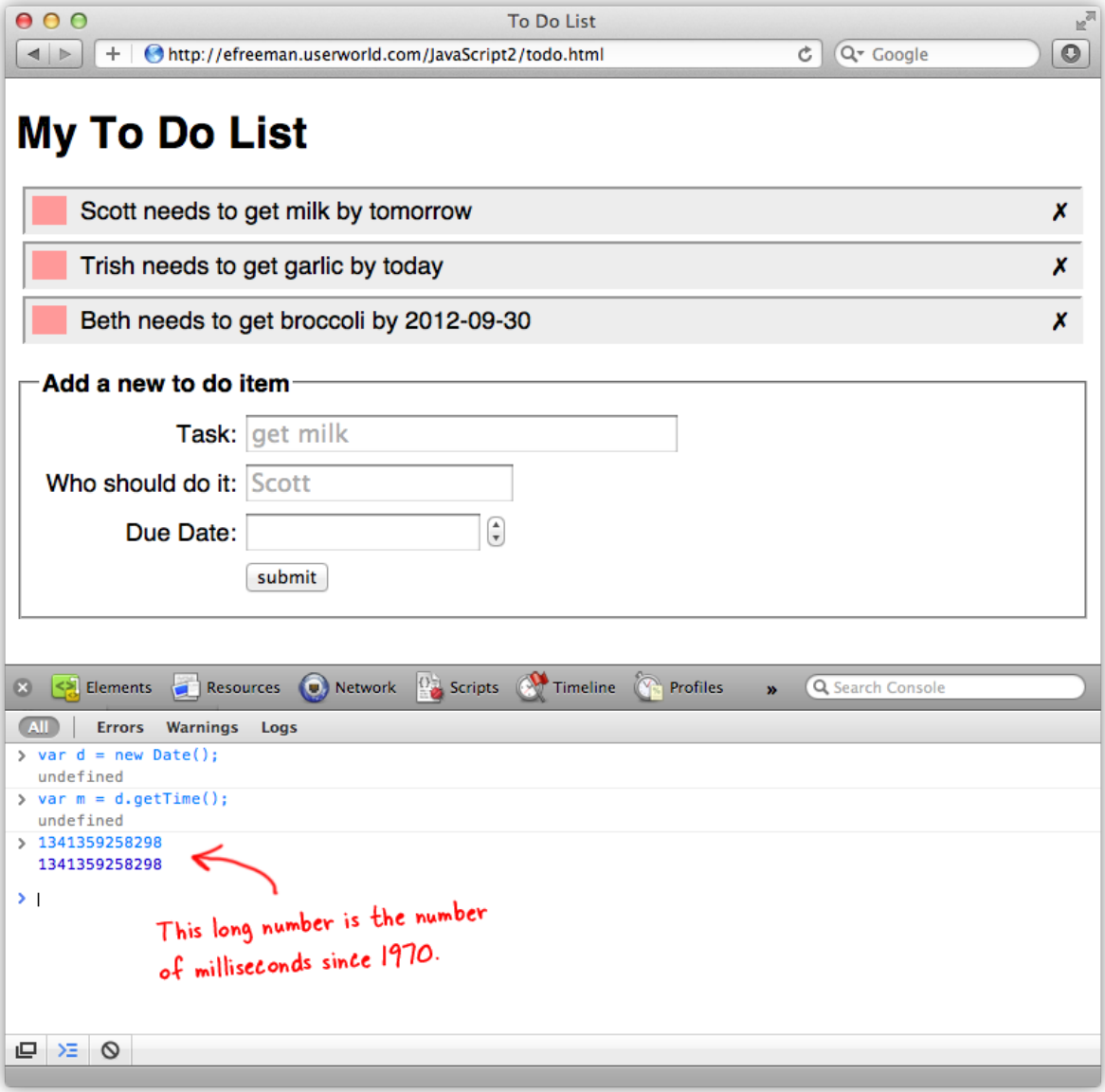
You might think of this as an odd choice for creating a unique id, but it turns out to work well on today's computers, and using JavaScript in particular, because we can get a value for time that is **the number of milliseconds since 01/01/1970**. Why 1970? That's just a date that was picked a long time ago by computer scientists and we must accept it. The JavaScript **Date** object's **getTime()** method still uses it (and this technique is used by many computers and many computer languages, so you'll see it pop up in other types of applications too).

Let's try an example. Go to the JavaScript console in a web page and type this:

CODE TO TYPE:

```
var d = new Date();
var m = d.getTime();
console.log(m);
```

It should look similar to this in your browser's JavaScript console:



Of course, your big long number representing the number of milliseconds will be different because you're doing it later than when I ran this code.

The code **new Date()** creates a new **Date** object that represents "right now." (Try typing **console.log(d)** to see). We convert that date into milliseconds using the **getTime()** method. We'll talk more about the **Date** object and the **Date** object constructor later in the course.

We can combine the two lines to create a **Date** object and get the time with the **getTime()** method into one, like this:

CODE TO TYPE:

```
(new Date()).getTime();
```

Try that in your console and make sure it works. Make sure you put the parentheses in the right places! The parentheses ensure that we get the **Date** object for "right now" first, and then call **getTime()** on that object. You'll see this syntax used often in JavaScript, both with the **Date** object and other objects too.

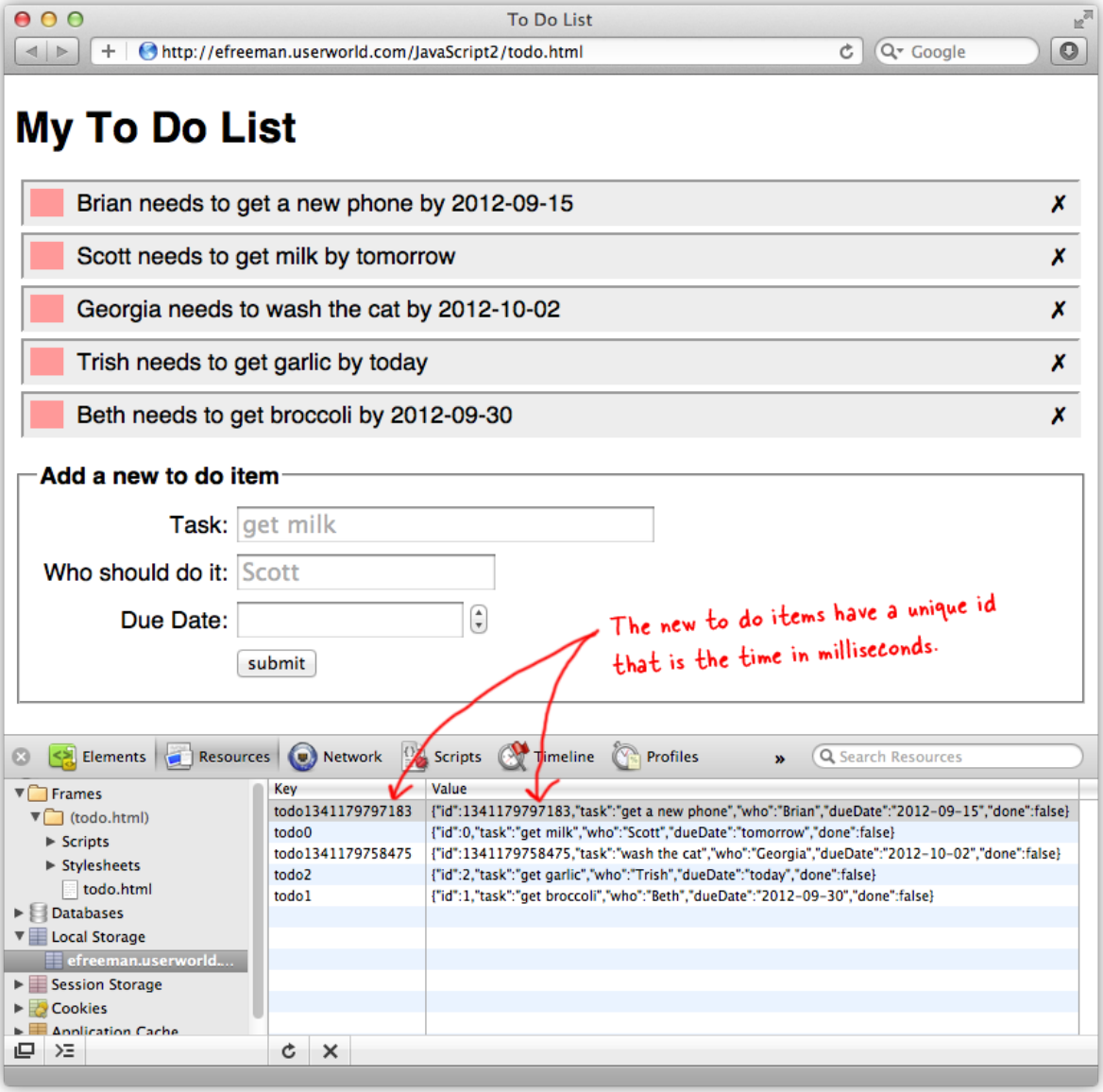
Okay, now that we have a new scheme for our ids, let's add it to our code. We just have to change one line in the function **getFormData()** in our **todo.js** file:

CODE TO TYPE:

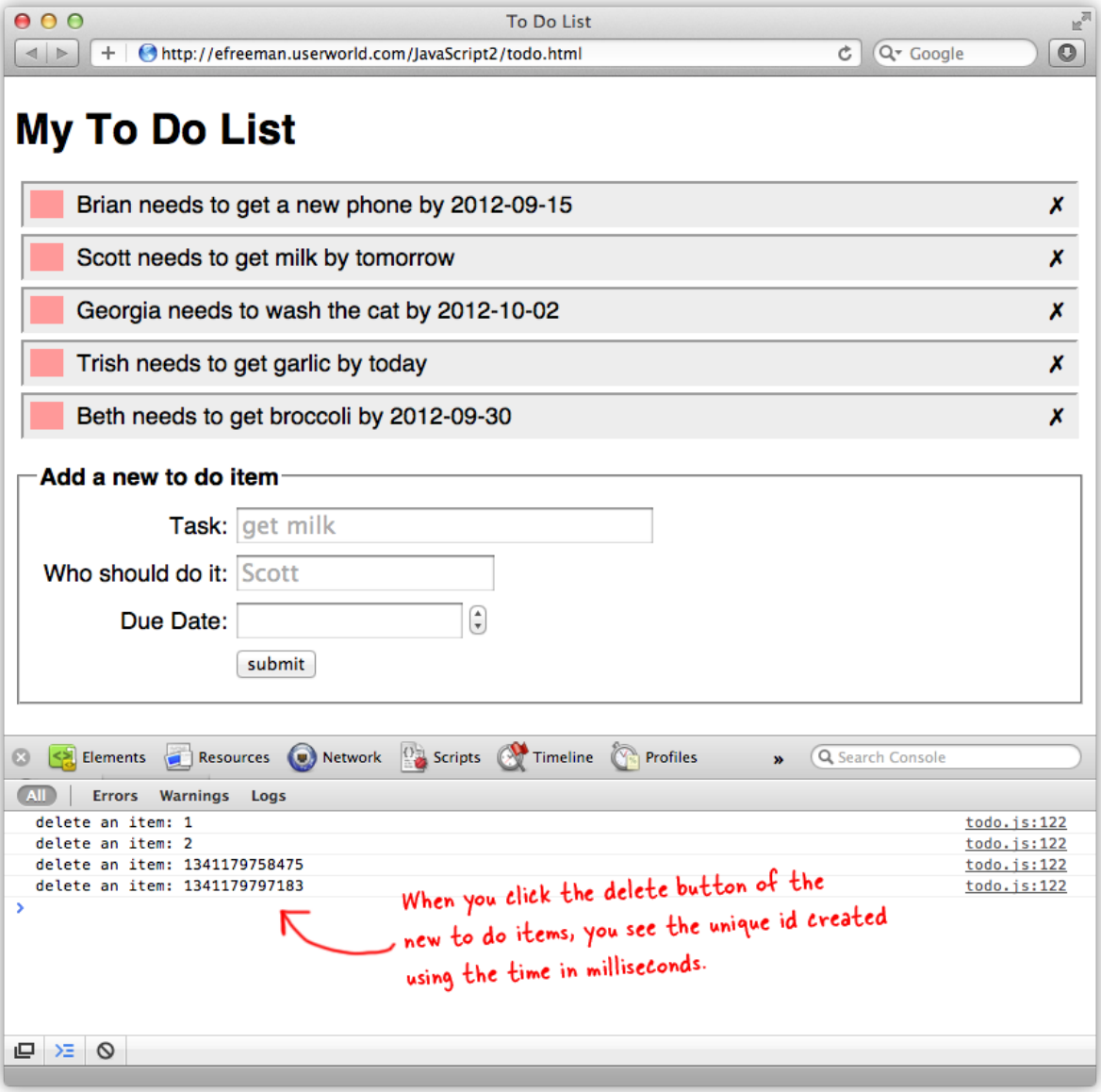
```
function getFormData() {
    var task = document.getElementById("task").value;
    if (checkInputText(task, "Please enter a task")) return;

    var who = document.getElementById("who").value;
    if (checkInputText(who, "Please enter a person to do the task")) return;

    var date = document.getElementById("dueDate").value;
    if (checkInputText(date, "Please enter a due date")) return;

    var id = todos.length;
    var id = (new Date()).getTime();
    var todoItem = new Todo(id, task, who, date);
    todos.push(todoItem);
    addTodoToPage(todoItem);
    saveTodoItem(todoItem);
}
```

 Save it, open **todo.html**, and click 

. Make sure you have your Developer console open and display the contents of Local Storage, either using the JavaScript console, or using the Resources tab (depending on your browser). If you have any existing to-do items, those will have the old ids. That's fine. Try adding some new ones; refresh your view of Local Storage and make sure they have new ids that use the time in milliseconds.



Click on an item that has one of the new ids. In the JavaScript console, you'll see the console.log message that shows the id of the item you want to delete, as well as the big, long id.



Okay, now that we have a new id, we're *really* ready to implement **deleteItem()**, so let's get to it!

Deleting Items from Local Storage and the To-Do List

Now that we have an id scheme that won't cause our To-Do List app to fail when we delete items, we're finally ready to implement delete. When we click on an item to delete, we need to remove the item from Local Storage, from the array of to-do items, and from the list in the page. Update the **deleteItem()** function in **todo.js** as shown:

CODE TO TYPE:

```
function deleteItem(e) {
    var id = e.target.id;
    console.log("delete an item: " + id);

    // find and remove the item in localStorage
    var key = "todo" + id;
    localStorage.removeItem(key);

    // find and remove the item in the array
    for (var i = 0; i < todos.length; i++) {
        if (todos[i].id == id) {
            todos.splice(i, 1);
            break;
        }
    }

    // find and remove the item in the page
    var li = e.target.parentElement;
    var ul = document.getElementById("todoList");
    ul.removeChild(li);
}
```

Save it, open **todo.html**, and click Preview. Try adding and deleting some items. Check your Local Storage using the developer console and make sure the items are gone. Reload the page and make sure, again, that they are gone.

Let's walk through the code. First, we remove the to-do item from Local Storage. We use the **localStorage** method, **removeItem()** and pass in the **key to delete**. The key is a string created by **combining "todo" with the id**, just like we did when we created the key to add the item to Local Storage. Remember, we get id from the element that we click on to delete an item:

OBSERVE:

```
// find and remove the item in localStorage
var key = "todo" + id;
localStorage.removeItem(key);
```

Next, we delete the to-do item from the **todos** array:

OBSERVE:

```
// find and remove the item in the array
for (var i = 0; i < todos.length; i++) {
    if (todos[i].id == id) {
        todos.splice(i, 1);
        break;
    }
}
```

To delete a to-do item from the **todos** array, we **loop through the array**, looking for the **Todo** object with the same id as the id of the item on which we clicked. If we **find the Todo with that id**, then we can remove it from the array using the array method, **splice()**. We pass in the **position of the item we're deleting (i)**, and the **number of items to delete (in this case, just one)**, then **splice()** removes that item from the array, and shifts everything else in the array down by one so that there's no empty spot in the array. You can use **splice()** to remove multiple elements from an array, but in this case, we just need to delete the one to-do item with the id that matches the one we clicked on to delete.

If we've **found the item to delete**, we don't need to keep looping. It would be a waste of time to loop over the rest of the elements, because we know none of the other items will match the id we're seeking. So, we can use the **break** statement to break out of the loop. When you use **break**, the loop stops, and the code following the **for loop** will run next.

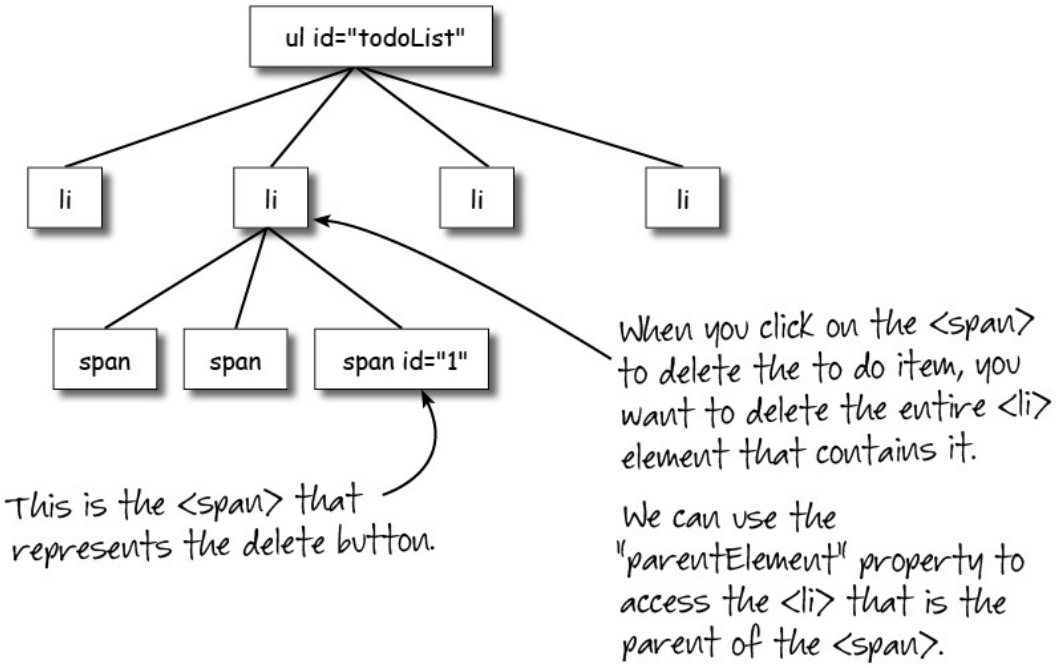
In this case, that next code is the code to remove the to-do item from the page. We've removed it from Local Storage, and removed it from the **todos** array, but the item still appears on the page:

OBSERVE:

```
// find and remove the item in the page
var li = e.target.parentElement;
var ul = document.getElementById("todoList");
ul.removeChild(li);
```

To remove the item from the page, we need to remove the ** element that represents (and displays) the to-do item in the page** from the DOM. Remember that the DOM is an internal browser structure that represents what you see on the page. It contains all the elements and content of the page.

When you click on the delete , that is passed into the click handler function, **deleteItem()**, in the event object, and we access it with the property **target**. Because the is a *child* of the <li> element we want to remove, we can get the <li> element using the **parentElement** property of the . Once we have the right **<li>** element, we can remove it from the DOM by using the **removeChild()** method of the "todoList" element that contains that **<li>** element. Here's how it works:



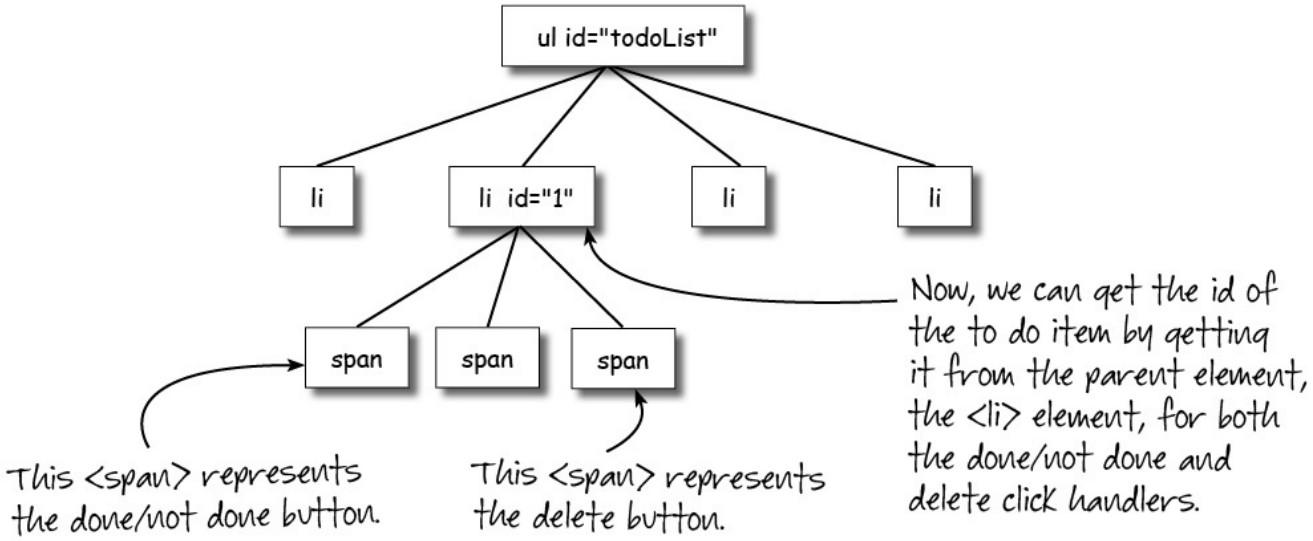
Calling **removeChild()** and passing in the **<li>** element immediately removes that list item from the page, and then you see the to-do item disappear when you click delete.

Moving the ID to the Parent Element

Before we end the lesson, we need to do one more thing. In the project for this lesson, you're going to implement the done/not done button. however, right now, the element that represents the done/not done button has no id; no way to know the to-do item id you want to update when you click on the button.

We can't just add another id to that element with the same id we're using for the delete button, because id attributes must be unique in HTML. Hmm....

A good solution in this case is to move the id up to the parent element: the <li> that contains *both* the done/not element and the delete element. Now that you know how to access an element's parent element, you can get the id of the list item to modify from the parent element, the <li>, rather than the , so that single id will work for both the delete click handler function, and the done/not done click handler function that you're going to write in the project. Here's how that will work:



Let's go ahead and move the id from the delete up to the <li> element for the to-do item when we create that element, in **createNewTodo()**. We'll also need to update the **deleteItem()** function so that it gets the id from the parent element (the <li>) rather than from the target element (the). Modify **todo.js** as shown:

CODE TO TYPE:

```
.
.
.
function createNewTodo(todoItem) {
    var li = document.createElement("li");
    li.setAttribute("id", todoItem.id);

    var spanTodo = document.createElement("span");
    spanTodo.innerHTML =
        todoItem.who + " needs to " + todoItem.task + " by " + todoItem.dueDate;

    var spanDone = document.createElement("span");
    if (!todoItem.done) {
        spanDone.setAttribute("class", "notDone");
        spanDone.innerHTML = "     ";
    }
    else {
        spanDone.setAttribute("class", "done");
        spanDone.innerHTML = " &#10004; ";
    }

    //spanDone.onclick = updateDone;

    // add the delete link
    var spanDelete = document.createElement("span");
    spanDelete.setAttribute("id", todoItem.id);
    spanDelete.setAttribute("class", "delete");
    spanDelete.innerHTML = " &#10007; ";

    // add the click handler to delete
    spanDelete.onclick = deleteItem;

    li.appendChild(spanDone);
    li.appendChild(spanTodo);
    li.appendChild(spanDelete);

    return li;
}

function deleteItem(e) {
    var id = e.target.id;
    var span = e.target;
    var id = span.parentElement.id;
    console.log("delete an item: " + id);

    // find and remove the item in localStorage
    var key = "todo" + id;
    localStorage.removeItem(key);

    // find and remove the item in the array
    for (var i = 0; i < todos.length; i++) {
        if (todos[i].id == id) {
            todos.splice(i, 1);
            break;
        }
    }

    // find and remove the item in the page
    var li = e.target.parentElement;
    var ul = document.getElementById("todoList");
    ul.removeChild(li);
}
.
.
.
```

Save it, open **todo.html**, and click 

. Your delete button will work just as it did before, but now you know that you're getting the id to delete from the <li> containing the to-do item, rather than from the element containing the delete button.

All we did is change the **createNewTodo()** function so that instead of adding the id attribute to the delete , we add it to the <li> element. Then in **deleteItem()**, we use the element's **parentElement** property to get the id from the <li> element—the rest of the code stays the same.

Make sure you understand how this works because you're going to need to use it to implement a new click handler, **updateDone()** in the project.

Wow, you got through a lot in this lesson! We implemented the delete button and figured out how to make an id scheme using time in milliseconds that works well with our application. We introduced the **Date** object, the **Date()** constructor and the **getTime()** method briefly; we'll be coming back to **Date** in a later lesson.

Take a break, and then tackle the project to implement the done/not done functionality. Once you've completed that, you'll have a very nice To-Do List Application you can use!

Strings and String Methods

Lesson Objectives

When you complete this lesson, you will be able to:

- build a string search application.
- explore several of the different methods we have for manipulating strings in JavaScript.
- use methods and chain those methods to improve your searches.
- use the substring() and split() methods.
- use regular expressions to create a pattern to match in some text.

It's time for a well-deserved break from the To-Do List Application. In this lesson, we'll build a string search application and, in the process, explore several of the different methods we have for manipulating strings in JavaScript. You've used strings many times in this course, but we haven't done much with them except to put them into web pages. There is a lot more you can do with strings, as you'll soon discover.

String Basics

You already know that to create a string in JavaScript, you write some text in quotes, like this:

OBSERVE:

```
var myString = "I'm a string!";
```

It's fine for a string to contain a single quote, as long as you're using double quotes to delimit the string, but if you need a double quote in a string, you need to *escape* it, like this:

OBSERVE:

```
var myQuoteString = "He said, \"Give me the ice cream!\" but I didn't.";
```

Strings in JavaScript are a bit special. When you write a string, you're actually creating a **String object**. Just like other objects, String objects have methods and properties. You already know about one property, **length**. Type this into your browser's JavaScript console:

CODE TO TYPE:

```
var myString = "I'm a string!";
var len = myString.length;
console.log(len);
```

```
> var myString = "I'm a string!";
  var len = myString.length;
  console.log(len);
  13
< undefined
>
```

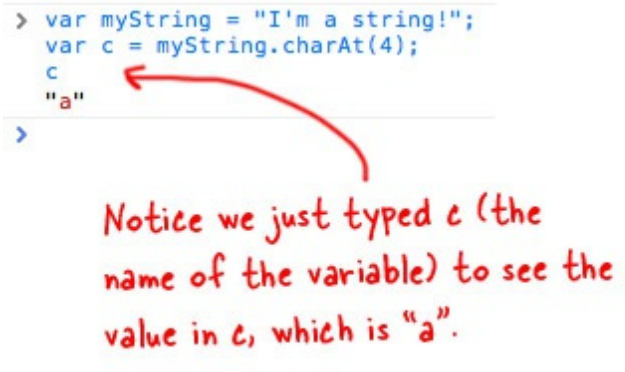
Try working through a few other strings for a little more practice.

Note When you want to see the value of a variable in the JavaScript console, you don't actually have to use **console.log()**; you can just type the name of the variable. For example, if you want the value of **len**, you can just type **len** at the console prompt.

Let's try a String method, **charAt()**:

CODE TO TYPE:

```
var myString = "I'm a string!";
var c = myString.charAt(4);
c
```



If "a" is the fifth character in the string, how did charAt() get "a"? It retrieved the character at position (also called **index**) 4. Just like arrays, strings start with position 0, so if you count 0, 1, 2, 3, 4 characters (including the space!), you'll find "a" in position 4. But just because you can access a character in a String using a position, or index, don't confuse it with an Array because they are two entirely different objects.

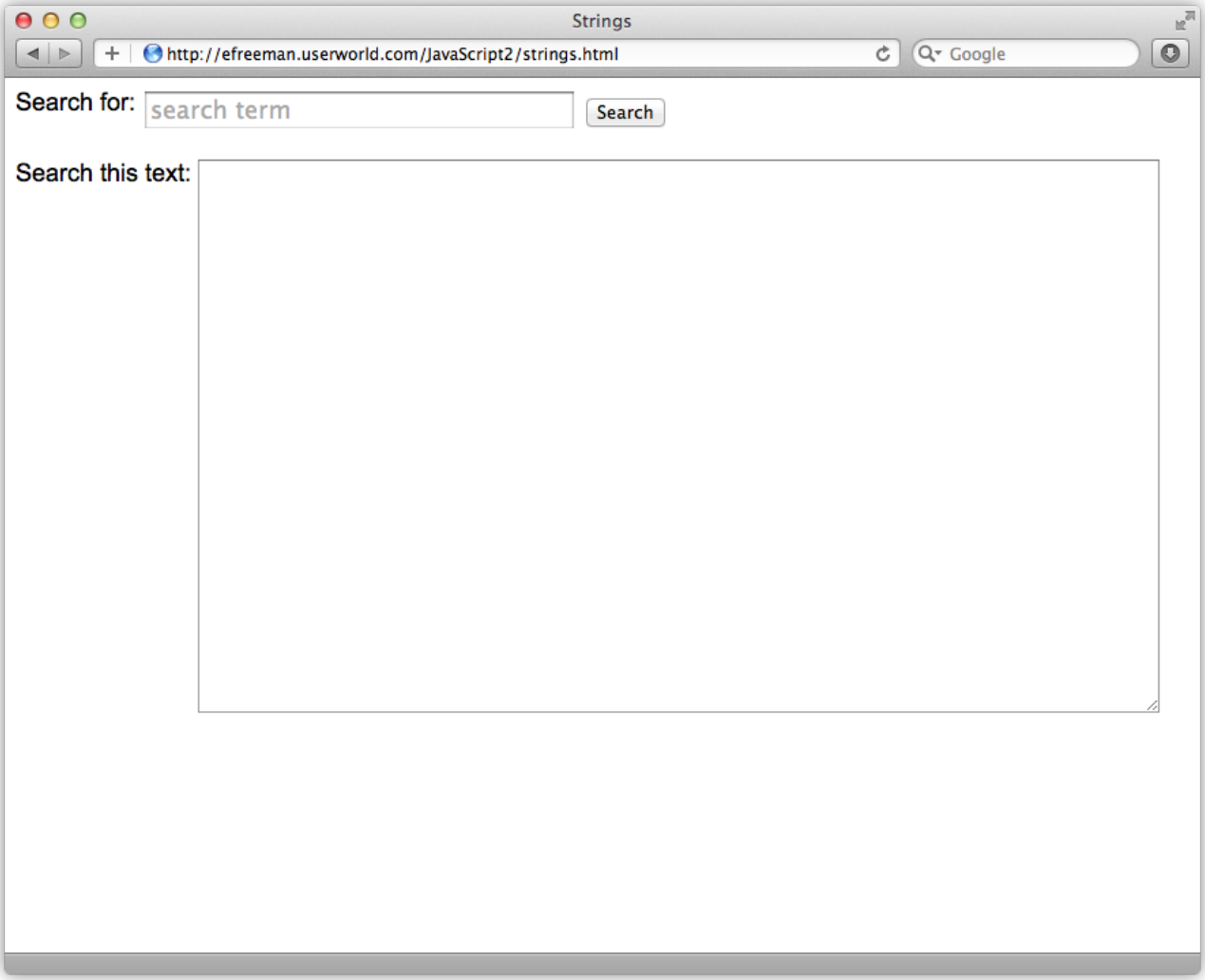
Basic String Comparision and Searching

Now that you know some string basics, let's start writing our string search application. We'll start with some basic HTML. Create this file:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>Strings</title>
  <meta charset="utf-8">
  <script src="strings.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    textarea {
      width: 700px;
      height: 400px;
    }
    label {
      vertical-align: top;
    }
  </style>
</head>
<body>
   <form>
     <label for="searchTerm">Search for: </label>
     <input type="text" id="searchTerm" size="35"
            placeholder="search term">
     <input type="button" id="searchButton" value="Search"><br><br>
     <label for="textToSearch">Search this text:</label>
     <textarea id="textToSearch"></textarea>
   </form>
</body>
</html>
```

Save the file in your **/javascript2** folder as **strings.html** and click Preview. You see this:



Nothing works yet because we haven't written the JavaScript. The plan is that when you click the Search button, we'll begin the search process and search for the string you enter in the top search area within the string in the textarea at the bottom. Notice that we're linking to the file **strings.js** from the HTML. That's where we'll add the JavaScript to make this work; let's do that now.

Open a new file and enter this JavaScript:

CODE TO TYPE:

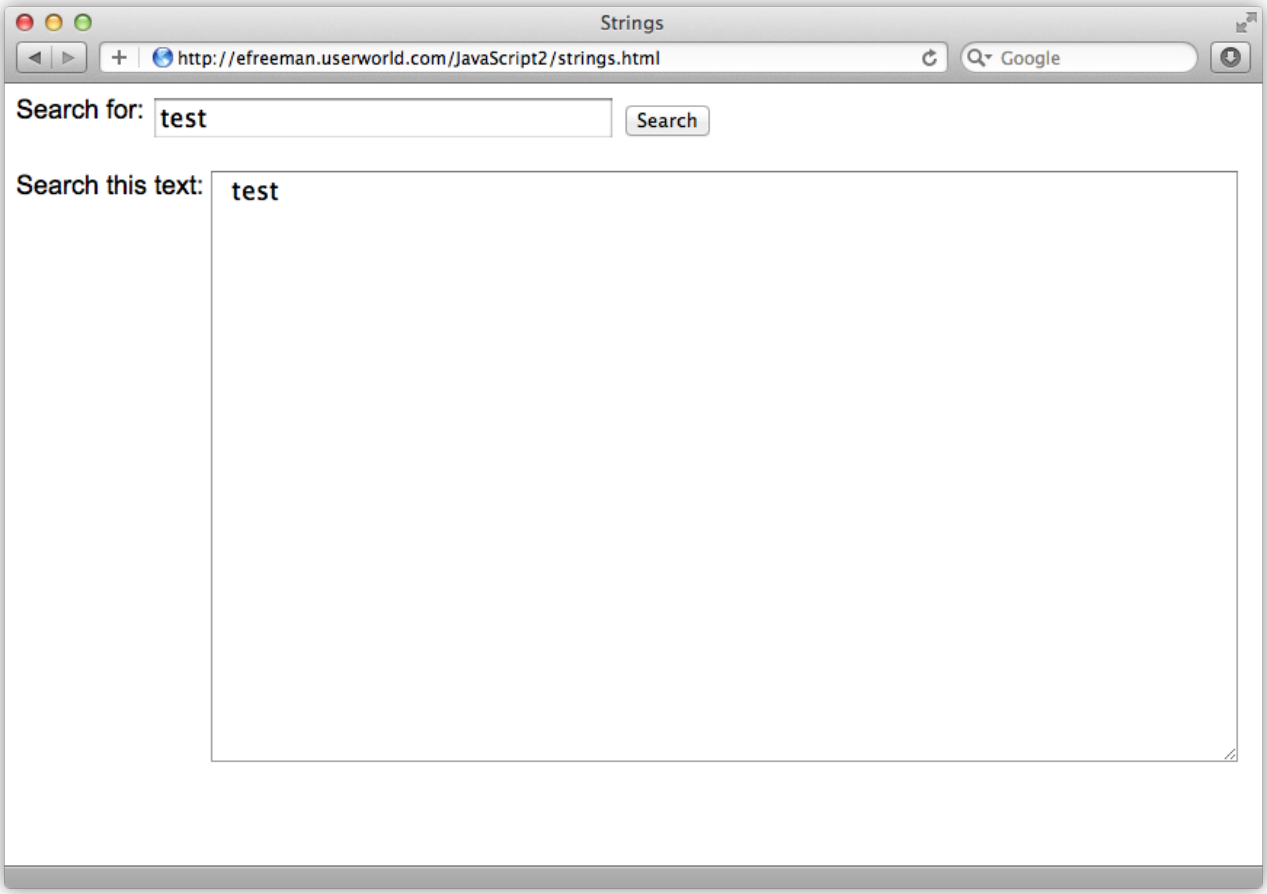
```
window.onload = init;

function init() {
    var searchButton = document.getElementById("searchButton");
    searchButton.onclick = searchText;
}

function searchText() {
    var searchTerm = document.getElementById("searchTerm").value;
    var textToSearch = document.getElementById("textToSearch").value;
    if (searchTerm == null || searchTerm == "") {
        alert("Please enter a string to search for");
        return;
    }
    if (textToSearch == null || textToSearch == "") {
        alert("Please enter some text to search");
        return;
    }
    if (searchTerm == textToSearch) {
        alert("Found 1 instance of " + searchTerm);
    }
    else {
        alert("No instance of " + searchTerm + " found!");
    }
}
```

 Save it as **strings.js**, open **strings.html**, and click 

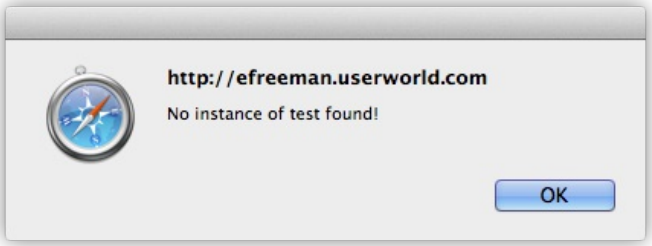
. Try entering a search string and some text to search. You'll probably notice two things right away:

- If either of the strings you enter has extra white space at the beginning or the end, you might find your search doesn't work, even if it appears that the text in both boxes is exactly the same.
- Both strings must be *exactly* the same in order for the search to work, which isn't particularly useful.

Don't worry, we'll fix both of these problems. First things first. Try entering "test" in the search area, and " test " in the text area:



These strings won't match when you click the Search button:



The code that tests to see if the two strings are the same:

OBSERVE:

```
if (searchTerm == textToSearch) {
    alert("Found 1 instance of " + searchTerm);
}
```

The **==** operator checks to see if they are *exactly* the same, spaces included; when comparing two strings using **==**, JavaScript compares the two strings, character by character, to see if they are the same, and that includes any spaces in the quotes. "test" does not equal " test ", so the message we get is correct.

The **trim()** method removes leading and trailing spaces from a string. This function is handy; adding extra spaces is a common error and typically people don't *really* want those extra spaces when they are comparing strings. (In some situations they might, however, so you'll need to take this on a case-by-case basis).

Modify **strings.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var searchButton = document.getElementById("searchButton");
    searchButton.onclick = searchText;
}

function searchText() {
    var searchTerm = document.getElementById("searchTerm").value;
    var textToSearch = document.getElementById("textToSearch").value;
    searchTerm = searchTerm.trim();
    textToSearch = textToSearch.trim();
    if (searchTerm == null || searchTerm == "") {
        alert("Please enter a string to search for");
        return;
    }
    if (textToSearch == null || textToSearch == "") {
        alert("Please enter some text to search");
        return;
    }
    if (searchTerm == textToSearch) {
        alert("Found 1 instance of " + searchTerm);
    }
    else {
        alert("No instance of " + searchTerm + " found!");
    }
}
```

 Save the file (**strings.js**), open **strings.html** and click 

. Now two strings will match if they have the same letters, even if you use leading or trailing spaces. Try it! Try "test" and " test " again.

The built-in trim function is only available in recent versions of most browsers: Firefox 3.5+, Safari 5+, IE9+, Chrome 5+, and Opera 10.5+. For browsers that don't support the built-in function, you can substitute your own implementation:

OBSERVE:

```
function trim(str) {
    return str.replace(/^\s+|\s+$/g,"");
}
```

And then instead of calling **myString.trim()**, you'd write **trim(myString)**.

Improving the Search

So far, our search is rather underwhelming, but there are lots of things we can do to improve it. Let's start by using two additional String methods, **indexOf** and **toUpperCase()**.

The **indexOf()** method takes a string to find in the string it's called on, and returns the position of the first instance it finds. So, if you write:

OBSERVE:

```
var myString = "I scream, you scream, we all scream for ice cream";
var pos = myString.indexOf("scream");
pos
```

...you'll get the value 2 in the variable **pos**, because the **first instance of "scream"** starts at position 2 in the variable **myString**. **indexOf()** can also take an optional argument, a starting position, so that you can start looking for a string at that position (if you don't supply this argument, the default is to start looking at position 0, the beginning of the string):

OBSERVE:

```
var myString = "I scream, you scream, we all scream for ice cream";
var pos = myString.indexOf("scream", 3);
pos
```

Now you'll get 14, the position of the **first instance of "scream", starting at or after position 3**.

If **indexOf()** doesn't find any instances of the string you pass it, then it returns -1.

The function **toUpperCase()** converts the characters in a string to upper case:

OBSERVE:

```
var myString = "I scream, you scream, we all scream for ice cream";
var myStringUpper = myString.toUpperCase();
myStringUpper
```

Our code produces the string "I SCREAM, YOU SCREAM, WE ALL SCREAM FOR ICE CREAM", in the variable **myStringUpper**. There is an analogous **toLowerCase()** method also. Notice that these methods do not change the original string, **myString**, rather, the methods return a *new* string that you can store in a different variable if you want.

Let's use these two methods to improve our search. Modify **strings.js** as shown:

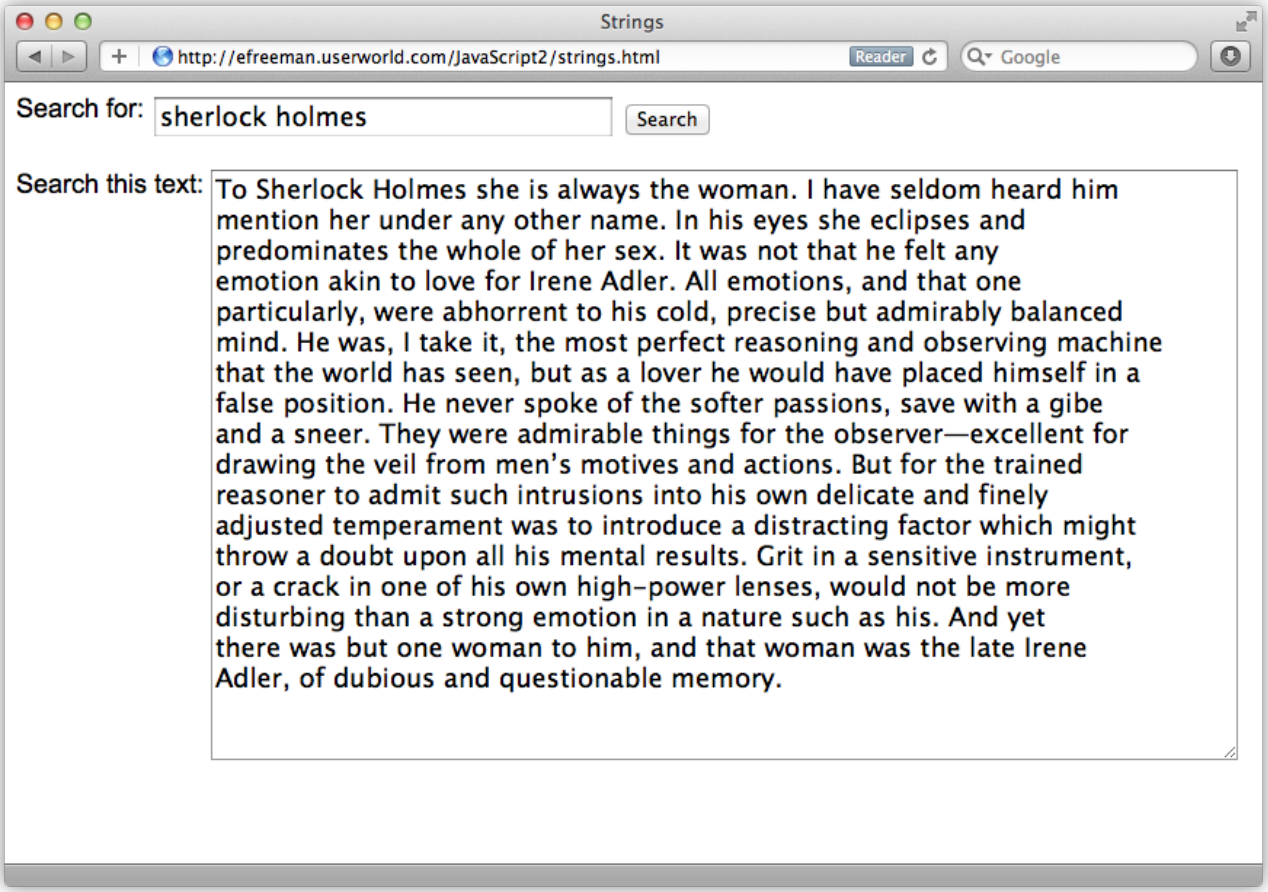
CODE TO TYPE:

```
window.onload = init;

function init() {
    var searchButton = document.getElementById("searchButton");
    searchButton.onclick = searchText;
}

function searchText() {
    var searchTerm = document.getElementById("searchTerm").value;
    var textToSearch = document.getElementById("textToSearch").value;
    searchTerm = searchTerm.trim();
    textToSearch = textToSearch.trim();
    if (searchTerm == null || searchTerm == "") {
        alert("Please enter a string to search for");
        return;
    }
    if (textToSearch == null || textToSearch == "") {
        alert("Please enter some text to search");
        return;
    }
    if (searchTerm == textToSearch) {
        alert("Found 1 instance of " + searchTerm);
    }
    else {
        alert("No instance of " + searchTerm + " found!");
    }

    var pos = 0;
    var count = 0;
    while (pos >= 0) {
        pos = textToSearch.toUpperCase().indexOf(searchTerm.toUpperCase(), pos);
        if (pos >= 0) {
            count++;
            pos++;
        }
    }
    alert("Found " + count + " instances of " + searchTerm);
}
```

Save the file, open **strings.html** and click Preview. Try a few test strings. Make sure you try:

- lower and upper case words
- words with spaces in them
- words with multiple instances in the text you're searching

We pasted in some text from The Adventures of Sherlock Holmes, and tried some searches on the first paragraph from the book.



How many times does the word "and" appear in the paragraph? Here's what you get if you try:

OBSERVE:

```
var pos = 0;
var count = 0;
while (pos >= 0) {
    pos = textToSearch.toUpperCase().indexOf(searchTerm.toUpperCase(), pos);
    if (pos >= 0) {
        count++;
        pos++;
    }
}
alert("Found " + count + " instances of " + searchTerm);
```

Let's break it down. We use a loop so we can find *all* instances of the **searchTerm** string that appear in the **textToSearch** string, and keep track of how many we find using the **count** variable. We use the **pos** variable to keep track of the position where we find each instance of **searchTerm**. As long as **pos is greater than or equal to 0** we know we've found another instance (remember, **indexOf()** returns -1 when it can't find an instance of the string for which you're searching). We convert both strings to upper case with **toUpperCase()** so that we can find "and" as well as "AND" or "And" (or even "aND"). We use **indexOf()** with the second, optional, argument, **pos**, so that we can start searching at a position that is *after* the position where we found the previous instance (so we don't keep finding the same instance over and over). As long as we keep finding a new instance of the **searchTerm**, we increment the position, and we increment the count. When **pos** is -1, the loop stops because we've found all the instances of **searchTerm** there are to find in **textToSearch**.

Chaining

We used a JavaScript technique in this code called **chaining**. Chaining is combining multiple method calls together in one line of code. For instance, if you write:

OBSERVE:

```
var myString = "This is the text we're searching to find the word 'and'.";
var anotherString = "AND";
var pos = myString.toUpperCase().indexOf(anotherString);
```

..it's the same as if you wrote:

OBSERVE:

```
var myString = "This is the text we're searching to find the word 'and'.";
var anotherString = "AND";
var myStringUpper = myString.toUpperCase()
var pos = myStringUpper.indexOf(anotherString);
```

By chaining expressions together with the "dot notation," you eliminate the intermediate variable you'd create if you used two statements instead. You can do this when you know that the result of the first method will yield a value you can use for the second method. In this case, **myString.toUpperCase()** returns an **upper-case string**, then we can call the method **indexOf()** on **that string**.

The Substring() and Split() Methods

Two other useful String methods you'll run into fairly often are **substring()** and **split()**.

substring() creates a string from a longer string, like this:

OBSERVE:

```
var myString = "I scream, you scream, we all scream for ice cream";
var myStringSub = myString.substring(0, 8);
myStringSub
```

The value of **myStringSub** is "I scream", a string made from the characters at positions 0-7 of **myString**. Notice that the character at position **8** is *not* included. **substring()** takes two values, **from** and **to**: **from** is the position of the first character you want included in the substring, and **to** is *one greater* than the position of the last character you want included in the substring.

The **split()** method is pretty handy. It splits a string into parts using a character as the divider for the split. So, let's say we want to split the text we entered in our string search application into words. We can split the input string using the " " (space) character. The result of **split()** is an array, so in this case, we'd get an array of all the words in the text. Let's modify our string search application a bit to count the number of words, and then search for a word by comparing the search text to each word we found in the text we searched. Modify **strings.js**:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var searchButton = document.getElementById("searchButton");
    searchButton.onclick = searchText;
}

function searchText() {
    var searchTerm = document.getElementById("searchTerm").value;
    var textToSearch = document.getElementById("textToSearch").value;
    searchTerm = searchTerm.trim();
    textToSearch = textToSearch.trim();
    if (searchTerm == null || searchTerm == "") {
        alert("Please enter a string to search for");
        return;
    }
    if (textToSearch == null || textToSearch == "") {
        alert("Please enter some text to search");
        return;
    }

    var pos = 0;
    var count = 0;
    while (pos >= 0) {
        pos = textToSearch.toUpperCase().indexOf(searchTerm.toUpperCase(), pos);
        if (pos >= 0) {
            count++;
            pos++;
        }
    }
    alert("Found " + count + " instances of " + searchTerm);

    var results = textToSearch.split(" ");
    var count = 0;
    for (var i = 0; i < results.length; i++) {
        if (searchTerm.toUpperCase() == results[i].toUpperCase()) {
            count++;
        }
    }
    alert("Found " + count + " instances of " + searchTerm + " out of a total of " + re
sults.length + " words!");
}
```

Save it, open **strings.html**, and click Preview. Try searching for a string. Now you'll see an alert that displays the number of times the string you searched for was found, as well as how many total words were found:

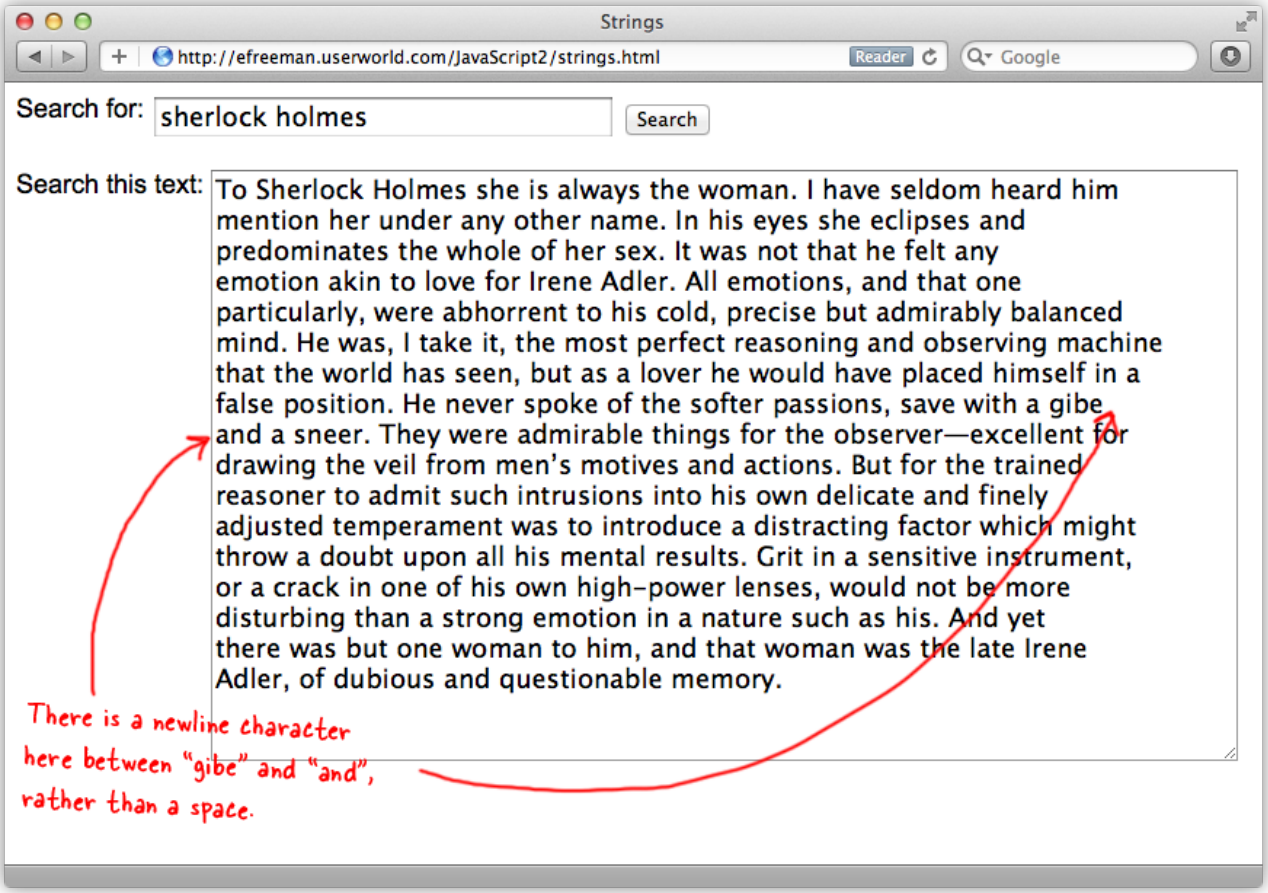
OBSERVE:

```
var results = textToSearch.split(" ");
    var count = 0;
    for (var i = 0; i < results.length; i++) {
        if (searchTerm.toUpperCase() == results[i].toUpperCase()) {
            count++;
        }
    }
```

First, we **split the textToSearch** into words using **split()**, and stored the results in **an array named results**. Then, we iterate over the entire array, and **compare each word in the array with the word we're searching for**, and **keep track of how many times we found it.**

If you're using the text from the Sherlock Holmes example (above), try the word "and," and you might see that "and" appears 8 times. If you tried "and" earlier (when we were using **indexOf()** rather than **split()**) you probably saw that it was found 9 times in the text. So why 8 this time? If you typed in the text to search on and pressed the "Enter" key on your keyboard between lines (rather than continuing to type and having the lines wrap around), then the words at the

end of each line and the beginning of the next line are not separated by a space; rather, they are separated by a **newline** character. So when you use **split()**, and you split the words up into an array using space (" ") as the split character, these words (the ones at the end of line/beginning of next line) aren't actually split up. So if "and" appears at the end or beginning of a line, it will not appear in the array as "and," but rather as "and" concatenated with another word. In my example, I pressed return between "gibe" and "and" so one of the words in the array was "gibe(newline)and." That word didn't match "and" and that's why I got 8 as the result rather than 9.



Typically, **split()** works best when you know for sure that a given character is used to separate text; for instance, in a CSV file, the "," character is used to delimit the columns in the file. It's also used to split smaller pieces of text, for example, a "firstname lastname" entry could be split into "firstname" and "lastname" using **split()**.

Regular Expressions

So far, we've been searching for exact matches for the search terms we've tried. For instance, we've tried searching for "and" to see how many times the word "and" appears in the text you enter to be searched. But what if you want to find, say, all the phone numbers in a bit of text, even if there is more than one and they are different?

For a task like that you need **Regular Expressions**. Regular Expressions are a powerful way to express patterns to match. You'll find Regular Expressions used frequently whenever text needs to be matched to a pattern; for instance, you can use Regular Expressions to verify that the pattern of text a user enters for a phone number in a form really does look like a phone number—or an email address. There are many uses of Regular Expressions; you'll see them in other programming languages, as well as in command line commands or shell scripts.

In JavaScript, there are a few ways to use Regular Expressions; we'll talk about one of these: **match()**.

Let's update our code to use a Regular Expression and then we'll come back to talk more about how Regular Expressions work. They're a bit mysterious at first, so don't worry if it takes you a while to get used to them. Modify **strings.js**:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var searchButton = document.getElementById("searchButton");
    searchButton.onclick = searchText;
}

function searchText() {
    var searchTerm = document.getElementById("searchTerm").value;
    var textToSearch = document.getElementById("textToSearch").value;
    searchTerm = searchTerm.trim();
    textToSearch = textToSearch.trim();
    if (searchTerm == null || searchTerm == "") {
        alert("Please enter a string to search for");
        return;
    }
    if (textToSearch == null || textToSearch == "") {
        alert("Please enter some text to search");
        return;
    }

    var results = textToSearch.split(" ");
    var count = 0;
    for (var i = 0; i < results.length; i++) {
        if (searchTerm.toUpperCase() == results[i].toUpperCase()) {
            count++;
        }
    }
    alert("Found " + count + " instances of " + searchTerm + " out of a total of " + results.length + " words!");

    var re = new RegExp(searchTerm, "ig");
    var results = textToSearch.match(re);
    if (results == null) {
        alert("No match found");
    }
    else {
        alert("Found " + results.length + " instances of " + searchTerm);
        // Show the matches in the page
        showResults(results);
    }

}
function clearResultsList(ul) {
    while (ul.firstChild) {
        ul.removeChild(ul.firstChild);
    }
}

function showResults(results) {
    var ul = document.getElementById("matchResultsList");
    clearResultsList(ul);
    var frag = document.createDocumentFragment();
    for (var i = 0; i < results.length; i++) {
        var li = document.createElement("li");
        li.innerHTML = results[i];
        frag.appendChild(li);
    }
    ul.appendChild(frag);
}
```

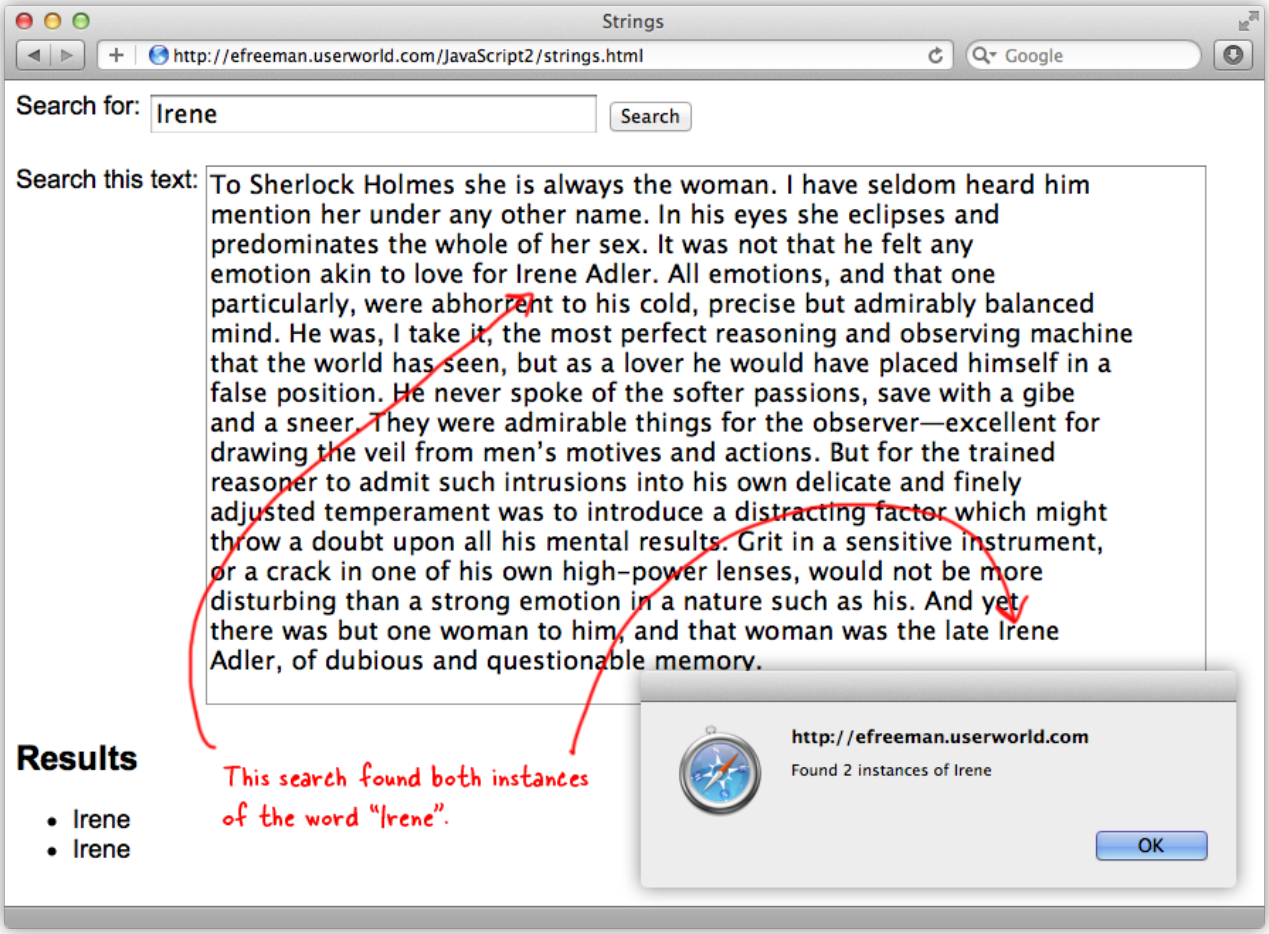
Save it. Before you try it though, we need to update **strings.html**, because we're going to update the HTML page with the results of our search match. We're just creating a list of matching words in the <ul> element with the id "matchResultsList," so we need to add this to the page:

CODE TO TYPE:

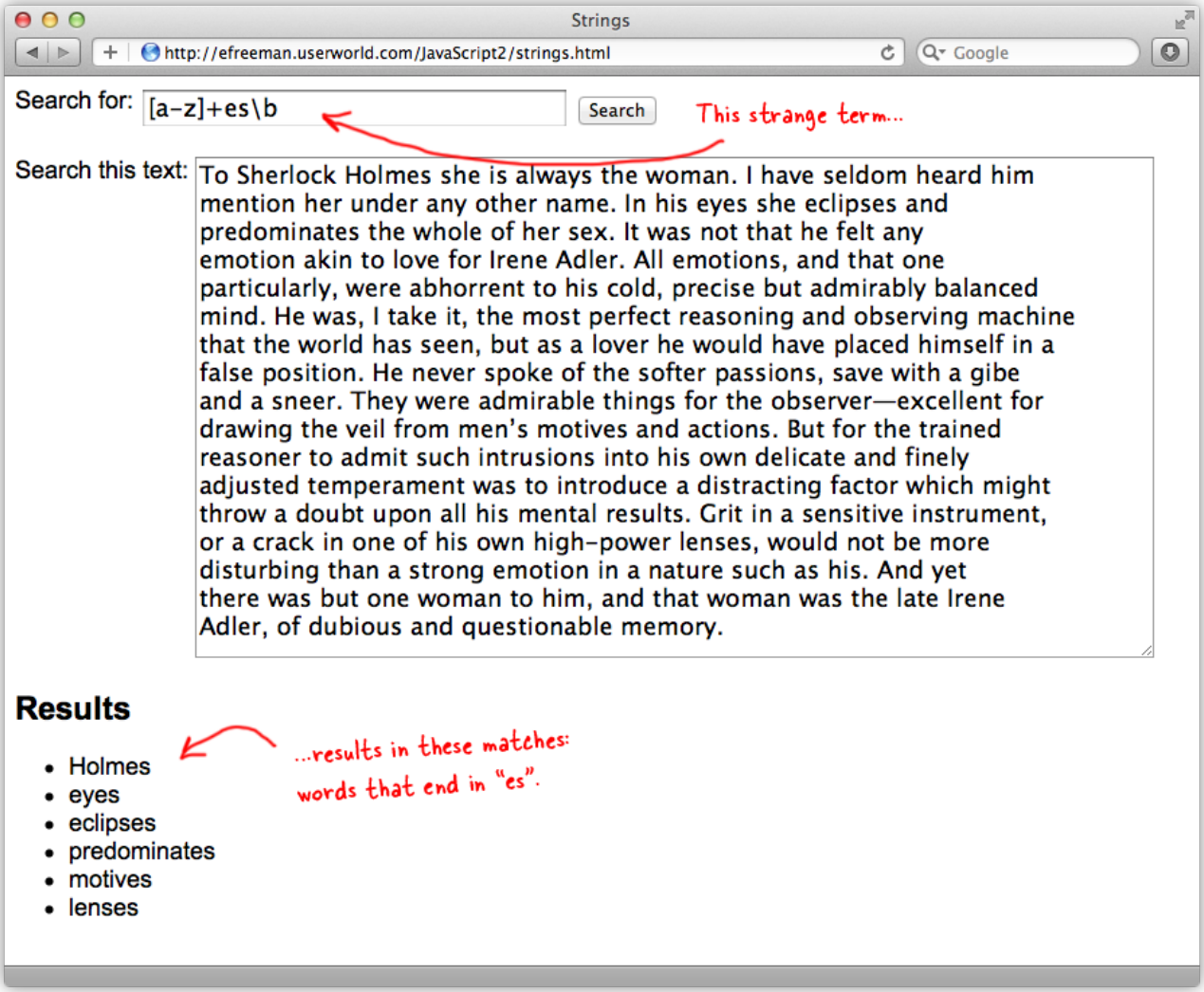
```
<!doctype html>
<html>
<head>
<title>Strings</title>
  <meta charset="utf-8">
  <script src="strings.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    textarea {
      width: 700px;
      height: 400px;
    }
    label {
      vertical-align: top;
    }
  </style>
</head>
<body>
   <form>
     <label for="searchTerm">Search for: </label>
     <input type="text" id="searchTerm" size="35"
            placeholder="search term">
     <input type="button" id="searchButton" value="Search"><br><br>
     <label for="textToSearch">Search this text:</label>
     <textarea id="textToSearch"></textarea>
   </form>
   <div>
     <h2>Results</h2>
     <ul id="matchResultsList">
     </ul>
   </div>
</body>
</html>
```

 Save the file, and click 

. Try searching for a string. For instance, if you enter the text from Sherlock Holmes again, search for "Irene". You'll find two matches; first, you'll see an alert that tells you how many matches you have, and then you'll see those matches in the results list in the page.



So what? We could do that before with **indexOf()**. Okay, try entering **[a-z]+es\b** in the Search for field:



Now you see a list of words that end in "es" in your page. That's pretty cool, right? But you're probably wondering, what on earth is **[a-z]+es\b**? It's a Regular Expression.

Regular Expressions Create a Pattern

Regular Expressions are used to create a pattern to match in some text. The simplest regular expression you can create is an exact word match, like we did above with "Irene." With this kind of regular expression, we just match the pattern "Irene" letter for letter; it works likes **indexOf()**, except that the method we used here, **match()**, returns an array of results, rather than a position.

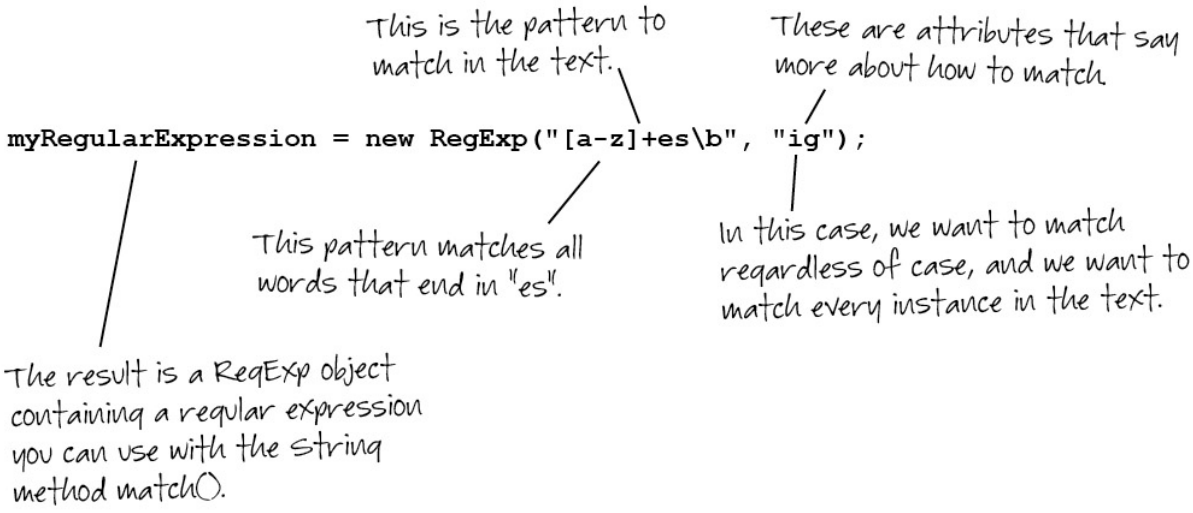
But you can also provide more complex regular expressions, ones that will match multiple strings. That's what we did with the regular expression "[a-z]+es\b", which matches all words that end in "es".

In our code, we create a **Regular Expression object**, using the **RegExp()** constructor. We pass in the **searchTerm**, which is a **pattern**—that is, the text you typed, like "Irene" or "[a-z]+es\b"—and a second argument, "ig." That second argument is a list of **attributes**. There are a few modifiers you can use; "i" and "g" are common. "i" says to ignore the case (so we'd match "Irene" to "IRENE" or "irene" or "irENe" or any combo like that), and "g" says to "globally" match every instance in the text, rather than just the first one:

OBSERVE:

```
var re = new RegExp(searchTerm, "ig");
```

This **RegExp** object can then be used to match strings in some text. The pattern can be exact, like "Irene", or it can be set up to match a variety of different words, like all words that end in "es", which is what the expression "[a-z]+es\b" does:



OBSERVE:

```
var re = new RegExp(searchTerm, "ig");
var results = textToSearch.match(re);
```

To match the pattern to some text, we use the **String method, match()**. **match()** takes a regular expression **re** and matches the pattern to the string, in this case **textToSearch**, that we called **match()** on. In the example above, **textToSearch** contains the text from Sherlock Holmes. We match that against the pattern "[a-z]+es\b" with the attributes "ig" (that is, find all instances in the text, regardless of their case). The **result is an array**, which we put into a variable, **results**.

Once we have the **results**, we can get the **length** of the array to count how many matches there were, or we can display each result in the page like we have:

OBSERVE:

```
var re = new RegExp(searchTerm, "ig");
var results = textToSearch.match(re);
if (results == null) {
    alert("No match found");
}
else {
    alert("Found " + results.length + " instances of " + searchTerm);
    // Show the matches in the page
    showResults(results);
}
```

We create a function **showResults()** that takes the **results array** and displays the results in the page. We also create a function **clearResultsList()** to clear the results list each time you submit a new search so you get new results each time. All of the code in **showResults()** and **clearResultsList()** is probably familiar to you.

Regular Expression Patterns

Let's take a closer look at the *attributes* and *patterns* you can use to create a *RegExp* object. Remember that in our search application, our Regular Expression object has the attributes "ig," so for all of these examples, we'll match regardless of case, and we'll search over all the instances in the text. If you want to, try removing one or both of these attributes to see the difference. Experiment!

Matching Characters in a Range [ ]

To match a character in a range, you use **[ ]** to specify the range. For instance, [a-z] says to match one letter between a and z; [0-9] will match one digit between 0 and 9. Try this: create some numbers in the text to search, varying the digits. For example, you could use 12, 333, 5985, 2, and so on. Now write a pattern that will match only digits 1, 2 and 3, like this: [1-3]. Enter the pattern in the search field. You should see all the

digits you typed that match 0-3, but notice, you're matching only single digits.

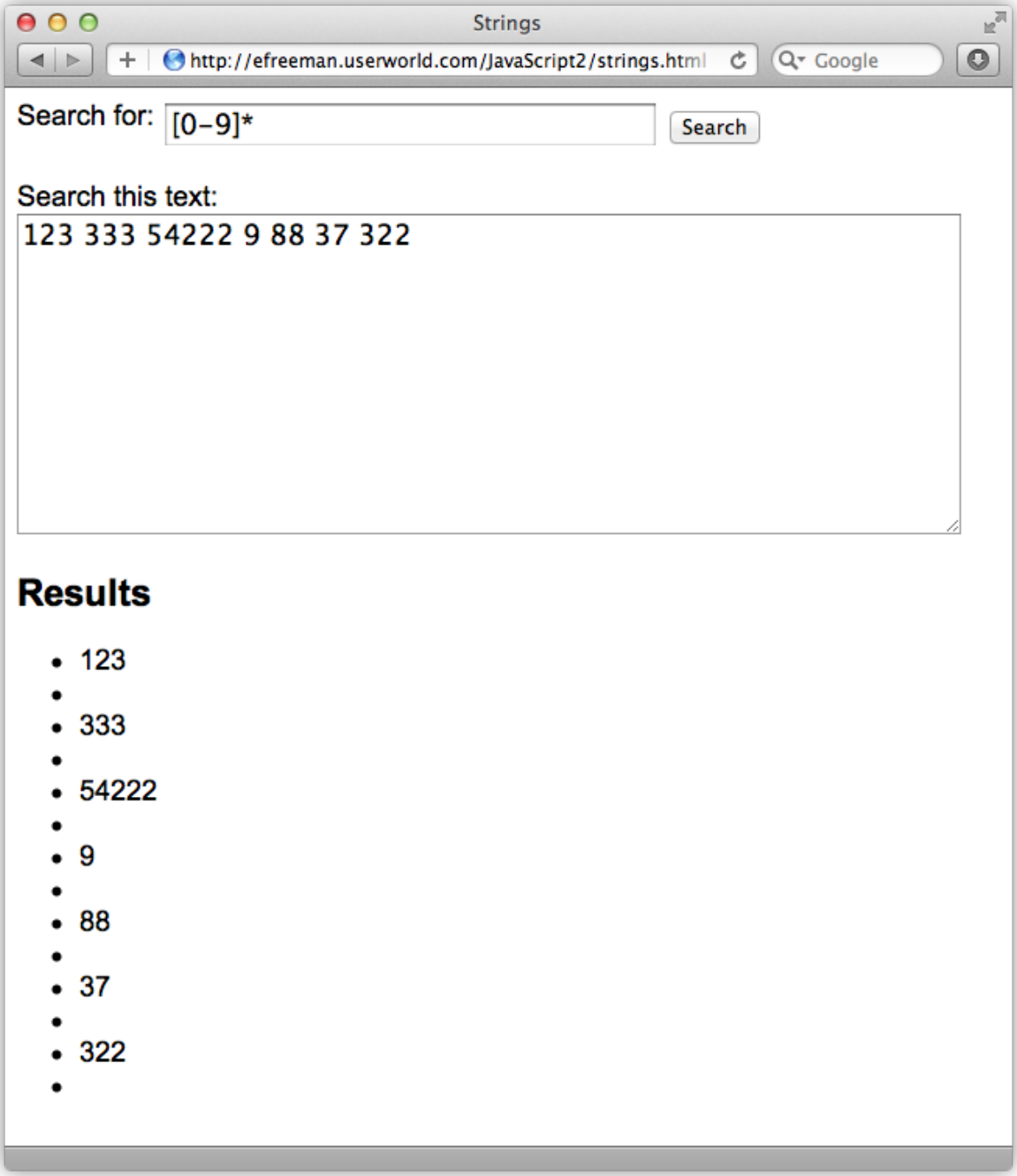
Matching Multiple Characters with *, +, and { }

SO, what if you want to match more than one character? You can use ***** to match any number of characters, including zero; use **+** to match one or more characters, and **{ }** to match a specific number of characters. Using your search application, try matching these patterns to the same numbers you entered before:

OBSERVE:

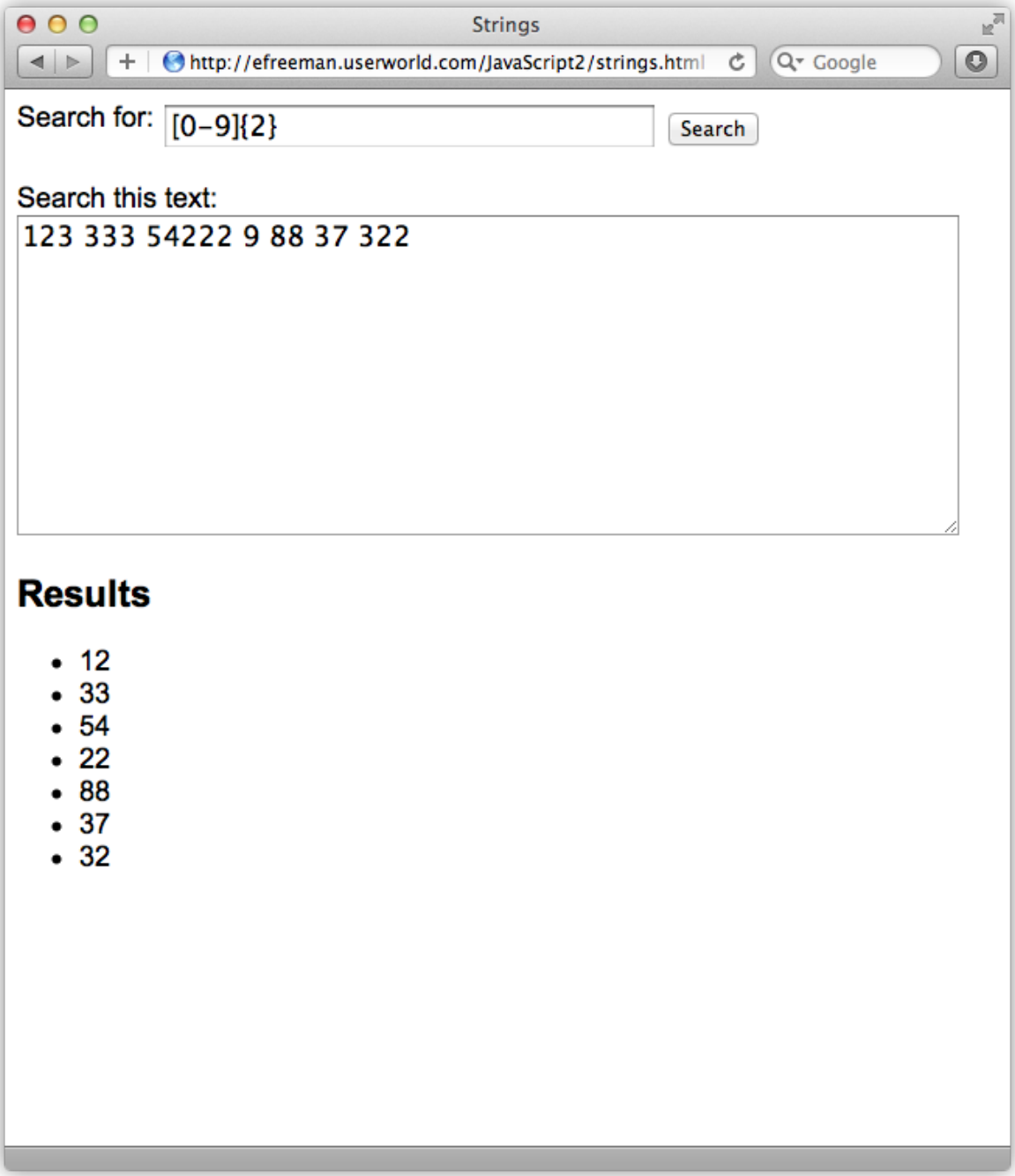
`[0-9]*`
`[0-9]+`
`[0-9]{2}`

[0-9]* says to match any number of characters (or none) as long as they are between 0 and 9. Your results will show all the numbers you typed, as well as matches of no numbers at all for the spaces in between the numbers. Why? Because the * matches *zero or more* characters. So for the first number, say 123, it will match all those digits. Then, for the space between 123 and the second number, say, 333, it will match zero digits. It will include that in the search results, and go on until it reaches the end of the numbers you input.



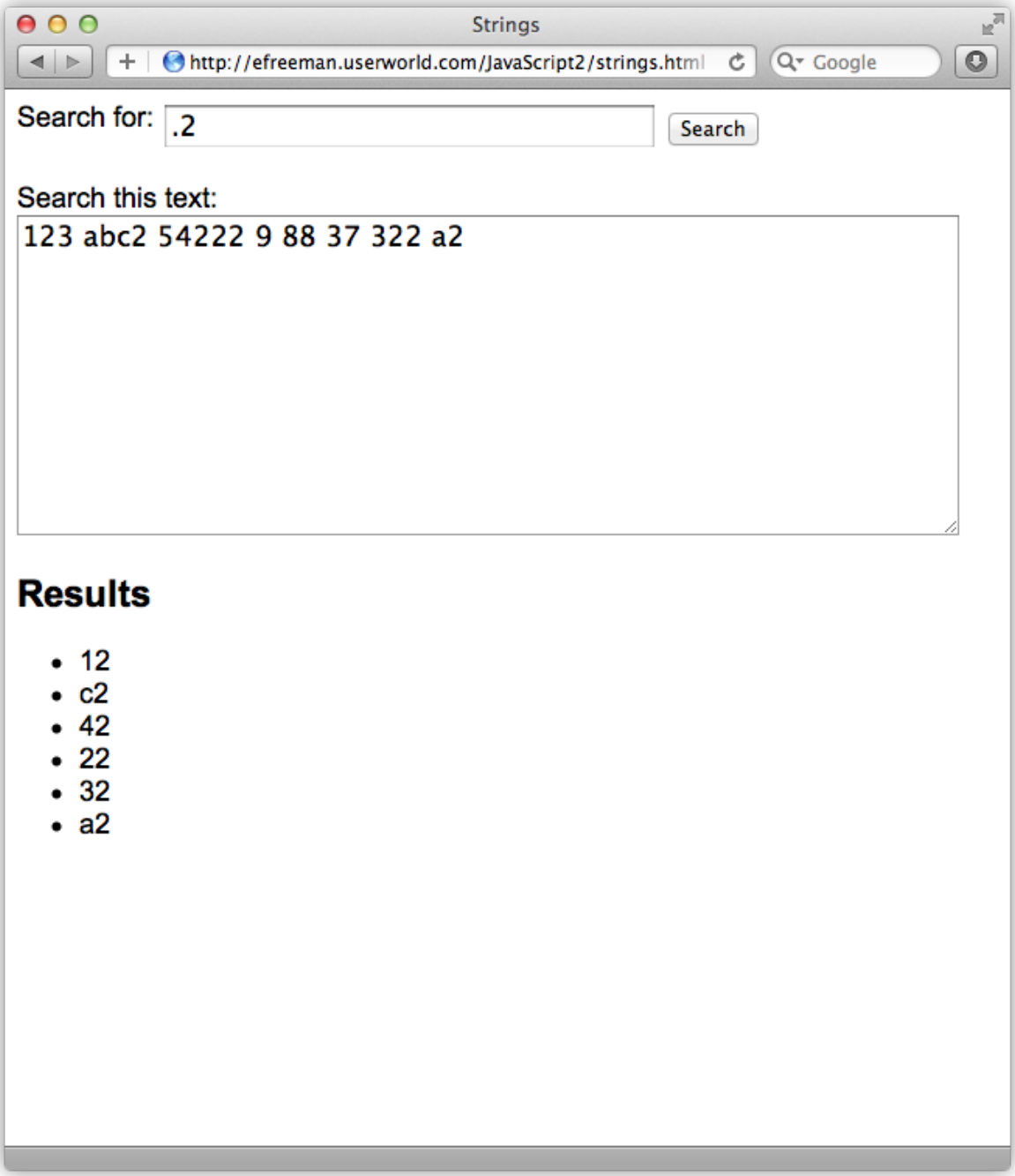
[0-9]+ says match one or more characters as long as they are between 0 and 9. Compare your results to [0-9]*. You'll see that your results are limited to the numbers you typed only; no "zero" results for the spaces in between, right?

[0-9]{2} matches 2 numbers precisely. Notice that it matches the sequences of two numbers only once, so if a number has been matched (like 12 in 123) the same number won't be matched again for a different sequence of two numbers (like 23, where the 2 is from the same number, 123).



Matching Characters at a Specific Position

What if you want to match one or more characters at a specific position? Try entering some letters next to a number, like I did here:



.2 matches *any non-white-space character* next to the number 2. Try **..2**; do you see three-character results? Now you're matching *two* non-white-space characters next to the number 2.

You can match characters or sequences of characters at the end of a string. Try entering: "I scream, you scream, we all scream for ice cream" in the text to search (don't put a period at the end of the sentence). Now search for:

OBSERVE:
cream$** **scream$

cream$** matches one "cream" in that sentence, the one at the very end, because **$ says match the pattern at the end of the string.

scream$ doesn't match with any results, even though "scream" appears three times in the sentence, because "scream" does not appear at the end of the string we're searching.

You can also match on a word boundary (rather than the whole string boundary, like we just did). Using the same sentence, try it:

OBSERVE:
`cream\b`

Now you see *four* results. Why? Because the string "cream" appears four times at the end of a word: "scream" three times, and "ice cream" one time, and in each case, "cream" is at the end of the word. The **\b** characters mean "match a word boundary" and because we put **\b** at the end of "cream", we're saying to match the end of the word. Try this instead:

OBSERVE:
`rea\b`

You see no results, because, while "rea" appears four times in the text, it doesn't appear at the end of a word.

You can use **\b** to match the beginning of a word too:

OBSERVE:
`\bscr`

You get three matches because you're matching "scr" three times in the three instances of "scream".

Matching Words that End in "es"

Now, you should be able to decipher the term "[a-z]+es\b" that we used earlier to match all words ending in "es" from the Sherlock Holmes text.

[a-z] says match any letter from a to z; [a-z]+ says match those letters one or more times so we can find words of any length. Since we're not matching spaces (or other delimiters, like "," or ";"), this locates only single words. But we are finding only words that end in "es" because of the "es\b". So [a-z]+es\b says: "Find all words with one or more characters matching a-z, ending in es (that is, the characters "es" are on a word boundary at the end of the word).

You can do much more with Regular Expression matching; we've hit some of the highlights here, but if you like using Regular Expressions (and you will as you write more code!), you should check out a good Regular Expressions reference book or online page for the various pattern options you can use. There are many of them out there.

Regular Expressions can be hard to read and understand. Keep that in mind as you learn more about Regular Expressions. Because they are so hard to read, using them in your programs can then make your programs more difficult to maintain. Malformed Regular Expressions can cause errors, so use them judiciously, and keep them simple!

Summary of String Properties and Methods

In this lesson you learned lots about JavaScript Strings. Now you know that strings are String objects, with properties and methods, like other JavaScript objects.

Property or Method	What It Does
length	Property: Returns the number of characters in the string.
charAt()	Method: Returns the character at a specific position in the string (remember that strings are 0-based like arrays).
trim()	Method: Removes leading and trailing spaces (but not spaces in the middle).
indexOf()	Method: Returns the position of a substring in the original string, or -1.
toUpperCase()	Method: Returns a new string that is the upper case version of the original string.
toLowerCase()	Method: Returns a new string that is the lower case version of the original string.

split()	Method: Splits a string into substrings based on a split character, returns an array.
match()	Method: Matches a Regular Expression to a string, returns an array.

These string manipulation techniques come in handy when you're processing forms or other kinds of text entered by the user in a web page.

Dates and Date Formatting

Lesson Objectives

When you complete this lesson, you will be able to:

- use various methods to get the date and time.
- set the date and time.
- compare and set dates relative to the present.
- convert strings to dates.

Earlier, you learned how to create a unique key for your to-do items in Local Storage using the **Date** object and the **getTime()** method, which returns the current date and time represented as the number of milliseconds since 1970.

It's time we return to the **Date** object, and its methods, and explore the power of this object further. The **Date** object has many of methods for getting and setting the date and/or the time; we'll focus on a few of the most useful methods.

Dates and times are a bit tricky to work with on the computer. If all you care about is the date right now on your own computer, it's fairly straightforward, but if you're creating a web page on the internet, then you'll have users from all around the world in different time zones visiting your site. Working with dates and times in code can be tricky when you're considering all time zones or, say, converting the way we write dates in the US to the way people write dates in other countries. Unfortunately, there's no one best way to tackle this stuff, so we'll look at the methods we have available, and you can experiment with your own code to see what works best for you.

What's the Date and Time Right Now?

Let's start with a web page that displays the current date and time when you load the page. Create a new HTML file as shown:

CODE TO TYPE:

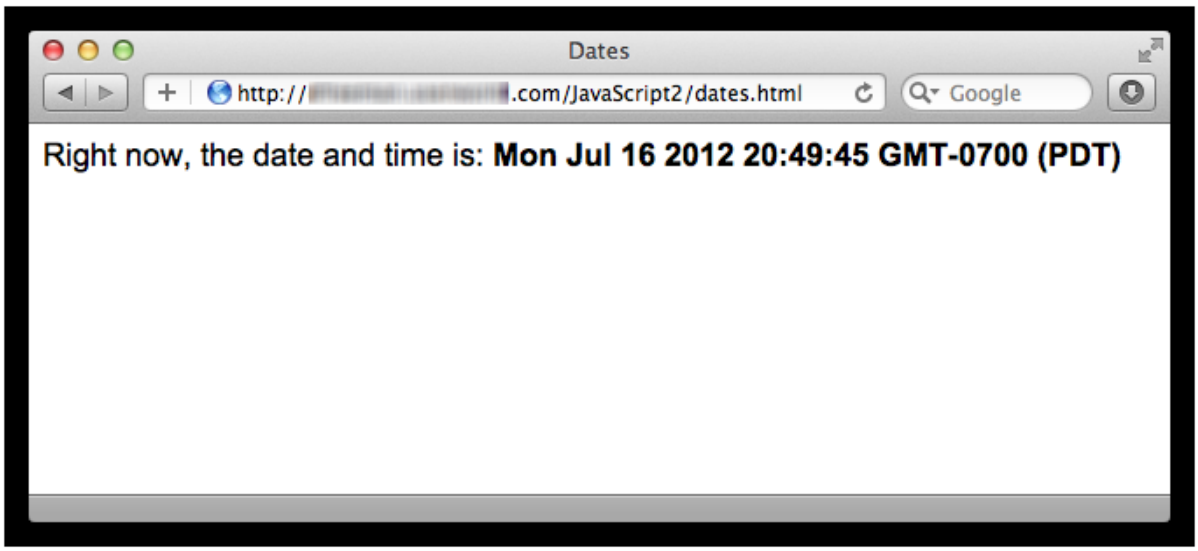
```
<!doctype html>
<html>
<head>
<title>Dates</title>
  <meta charset="utf-8">
  <script src="dates.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    span {
      font-weight: bold;
    }
  </style>
</head>
<body>
    <div>
      Right now, the date and time is:
      <span id="datetime"></span>
    </div>
</body>
</html>
```

Save it in your **/javascript2** folder as **dates.html**. Next, we'll create a new JavaScript file:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var now = new Date();
    datetime.innerHTML = now;
}
```

Save it in your **/javascript2** folder as **dates.js**. Open **dates.html** again and click Preview. You see a page with the current date and time. Because the JavaScript is running in the browser that's on your computer, you'll see the date and time for where you are, even though the file is stored and served from the web server at O'Reilly School of Technology.



In this code, we create a new **Date** object, and then set the content of the "datetime" element in the HTML to that date. By default, the value that you get is a string that shows you both the date and time.

OBSERVE:

```
datetime.innerHTML = now;
```

Here, we set the value of a property, **innerHTML**, that expects a String, not a Date. So JavaScript automatically calls the Date method **toString()** on the **now Date object** to convert it to a String. Try changing the code to:

OBSERVE:

```
datetime.innerHTML = now.toString();
```

You get exactly the same result.

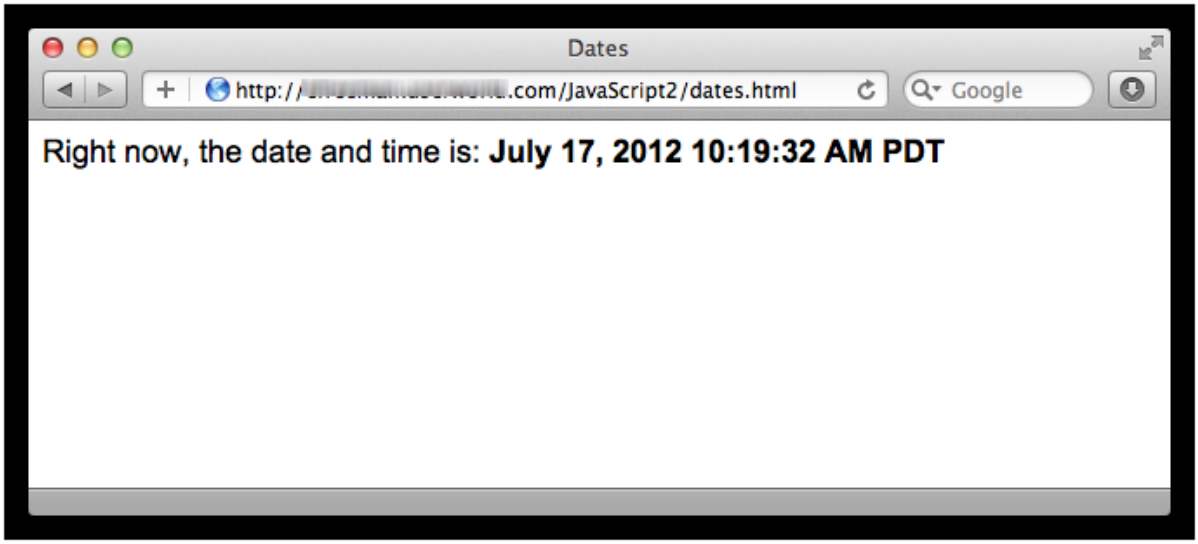
Other Methods for Getting the Date and Time

Along with **toString()**, there are several other methods for getting a string that represents the Date in a readable format, including: **toDateString()**, **toLocaleDateString()**, **toTimeString()**, **toLocaleTimeString()** and **toLocaleString()**. Try each of these by changing the code in the JavaScript in **dates.js**.

- **toString()**: Displays the date and time as a string using the local time zone.
- **toDateString()**: Displays the date as a string using the local time zone.
- **toLocaleDateString()**: Displays the date as a string using the local time zone, formatted using local conventions.
- **toTimeString()**: Displays the time as a string using the local time zone.

- **toLocaleTimeString()**: Displays the time as a string using the local time zone, formatted using local conventions.
- **toLocaleString()**: Displays the date and time as a string using the local time zone, formatted using local conventions.

Notice the differences in these various methods of creating a string from the **Date** object. For instance, compare the way the date string is displayed using **toLocaleString()** with how it was displayed using **toString()** (above):



Dates and Time Zones

In the previous example using **toString()**, in the date and time information displayed in the web page, you can see the time zone information: **GMT-0700 (PDT)**? I'm in the Pacific time zone in the United States, so my time is currently 7 hours behind the GMT (Greenwich Mean Time) measured from Greenwich, in London, England. GMT is similar to the Coordinated Universal Time (UTC), which is now the worldwide standard for measuring time. For most purposes (and certainly our purposes here), GMT and UTC are equivalent.

You can use the **Date** methods, **getTimezoneOffset()** and **toUTCString()** to help figure out differences in the local time where you are (or where someone using your web page is), and the UTC time. Let's update **dates.js** to use these methods to show how many hours we are from UTC time:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var now = new Date();
    datetime.innerHTML = now.toUTCString();
    var hoursDiff = (now.getTimezoneOffset()) / 60;
    datetime.innerHTML += ", " + hoursDiff + " hours difference from me.";
}
```

Save it, open **dates.html** again, and click Preview. Now you see the current time expressed in universal time (or GMT), and the number of hours difference from your local time to GMT.

The **getTimezoneOffset()** method returns the difference in minutes, not hours, so here, we divide by 60 to get the hours. Although you may want the time in minutes, because in some places the time difference from GMT does not occur on the hour. For instance, some places will be several hours plus a half hour different from GMT. I'm exactly 7 hours behind GMT, so my result is:

Setting a Date and Time

So far, all we've done with **Date** is get the current date, and convert it to a String for display in a web page. But what if you want to create a specific date?

The **Date()** constructor function creates a date that represents *now*, if you don't pass in any arguments. But you can also create specific dates, by passing in the year, month, day, hours, minutes, seconds, and even milliseconds of a specific date and time you want. Let's create the Date that represents New Year's Day, 2050. Modify **dates.html** as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>Dates</title>
  <meta charset="utf-8">
  <script src="dates.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    span {
      font-weight: bold;
    }
  </style>
</head>
<body>
    <div>
      ~~Right now, t~~The date and time is:
      <span id="datetime"></span>
    </div>
</body>
</html>
```

Save it, and modify **dates.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var now = new Date();
    datetime.innerHTML = now.toUTCString();
    var hoursDiff = (now.getTimezoneOffset()) / 60;
    datetime.innerHTML += ", " + hoursDiff + " hours difference from me.";
    var nyd = new Date(2050, 0, 1);
    datetime.innerHTML = nyd.toString();
}
```

Save it, open **dates.html** again, and click Preview.

OBSERVE:

```
var nyd = new Date(2050, 0, 1);
```

In this example, we created a **Date** object for New Year's Day, 2050, by passing in the **year 2050**, the **month 0** (for January), and the **day 1** (for the 1st of January). Notice that the **month** is 0, not 1. Why? Because in JavaScript, the months start at 0 and go to 11. That's a little weird, but that's the way it works. Days, however, do start at 1, and go up to 31 depending on the month.

Because we didn't supply any arguments for the time, JavaScript assumed we wanted 12:00 AM (midnight) on January 1, 2050 (which is perfect for celebrating the New Year!). We could have supplied time arguments, like this:

OBSERVE:

```
var nyd = new Date(2050, 0, 1, 0, 1, 0);
```

...where **0** is the hour (midnight), **1** is the minute (1 minute after midnight), and **0** is the seconds. Try other dates and times. Notice that if you want to specify a time, you *must* also specify a date. That is, while all the arguments to **Date()** are optional, you must supply them in order, and you can't skip any. So, if you want to supply minutes, you must also supply a year, month, day, and hour, but you can skip the seconds and milliseconds.

Try changing the code so you display the date and time using **toLocaleString()** instead. Modify **dates.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var nyd = new Date(2050, 0, 1, 0, 1, 0);
    datetime.innerHTML = nyd.toLocaleString();
}
```

Save it, open **dates.html** again, and click Preview. Compare your result with the previous result using **toString()**. Which do you like better?

Setting Date and Time Elements Separately with Methods

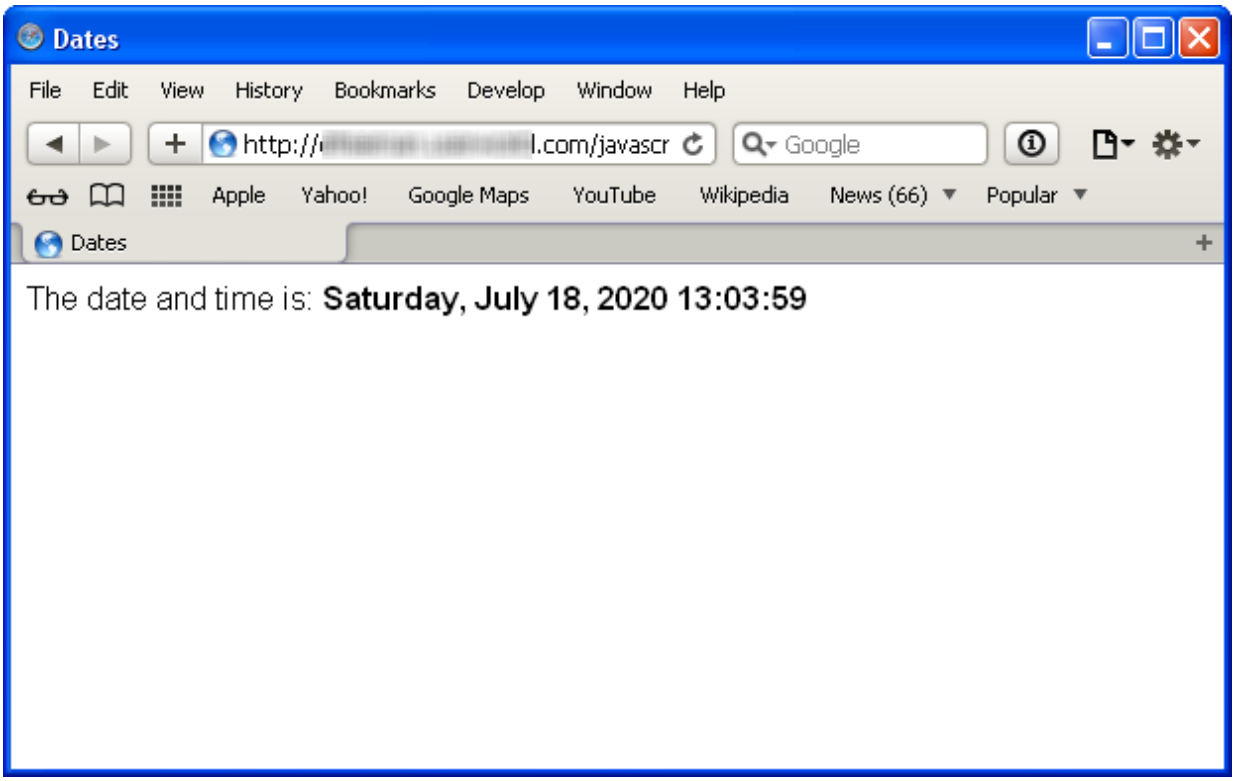
Using the **Date** constructor, you can specify the year, month, day, hour, minutes, seconds, and milliseconds all together to create a date and time. But what if you need to set the various values separately?

In that case, you can use the **Date** object's **set** methods. Modify **dates.js** as shown

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var nyd = new Date(2050, 0, 1, 0, 1, 0);
    datetime.innerHTML = nyd.toLocaleString();
    var aDate = new Date();
    aDate.setFullYear(2020);
    aDate.setHours(13);
    aDate.setMinutes(3);
    aDate.setSeconds(59);
    datetime.innerHTML = aDate.toLocaleString();
}
```

Save it, open **dates.html** again, and click Preview. You see your current month and day (that is, the day you are doing this lesson), but in the year 2020. And the time should be set to 1:03pm in your time zone.



Here, we created a new **Date** object that, by default, represents the present time, and then changed the year to 2020, the hour to 1pm, and the minute to 3 minutes after 1pm. Everything else stays the same (that is, the values in place when you created the Date representing the present).

There are methods for setting every aspect of a **Date**, including:

- **setFullYear()**: sets the year.
- **setMonth()**: sets the month.
- **setDate()**: sets the day of the month.
- **setHours()**: sets the hour of the day.
- **setMinutes()**: sets the minutes after the hour.
- **setSeconds()**: sets the seconds after the minute.
- **setMilliseconds()**: sets the milliseconds after the seconds.
- **setTime()**: takes a date represented as milliseconds after 1970 and sets the full date.

Experiment with these other methods and try using them in your code to set specific dates and times.

Comparing and Setting Dates Relative to the Present

Two tasks you might need to do fairly often when working with dates are *comparing dates* and *setting dates relative to the present*. For both of these tasks, working with dates expressed in milliseconds since 1970 is the way to go.

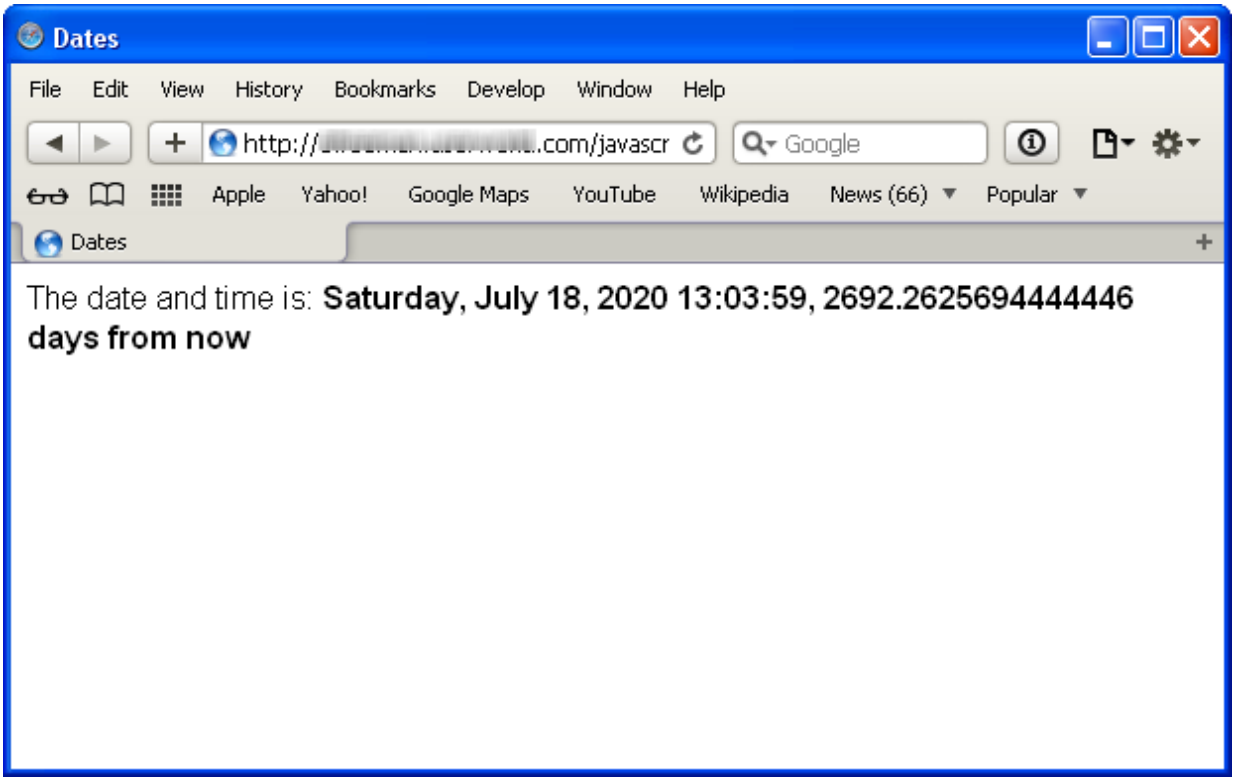
First, let's try comparing dates. Modify **dates.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var aDate = new Date();
    aDate.setFullYear(2020);
    aDate.setHours(13);
    aDate.setMinutes(3);
    aDate.setSeconds(59);
    datetime.innerHTML = aDate.toLocaleString();
    var now = new Date();
    var diff = aDate.getTime() - now.getTime();
    var days = diff / 1000 / 60 / 60 / 24;
    datetime.innerHTML = aDate.toLocaleString() + ", " + days + " days from now"
;
}
```

 Save it, open **dates.html** again, and click 

. You'll see the number of days between now and the same date in 2020 (and remember, your date will be different from mine because the dates are based on *now*, that is, the date and time you are doing this lesson).



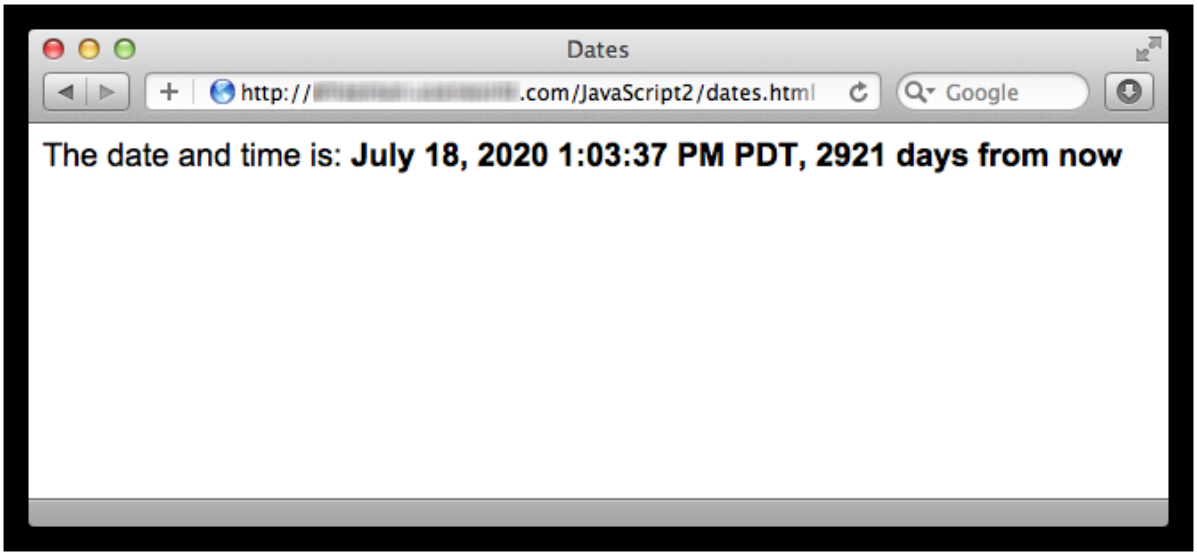
To compute the difference between two dates, we create two **Date** objects. We have one **Date** object for the date in 2020, and one **Date** object for now. Then, we can use the **getTime()** method to get the date and time in milliseconds since 1970. The number of milliseconds to the date in 2020 will be longer than the number of milliseconds to now (assuming it's still before 2020 of course!). So we subtract the milliseconds to now from the milliseconds to the date in 2020 to get the difference in milliseconds. Then, we convert from milliseconds to days by dividing by 1000 (the number of milliseconds in a second), then 60 (the number of seconds in a minute), then 60 again (the number of minutes in an hour), and then 24 (the number of hours in a day). The result is the number of days between now and the date in 2020.

That number is kind of ugly when we display it because of the precision of the number after the decimal point. Let's cut off everything after the decimal point to make it easier to read. We can use the **Math.floor()** method to do this. Modify **dates.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var aDate = new Date();
    aDate.setFullYear(2020);
    aDate.setHours(13);
    aDate.setMinutes(3);
    var now = new Date();
    var diff = aDate.getTime() - now.getTime();
    var days = Math.floor(diff / 1000 / 60 / 60 / 24);
    datetime.innerHTML = aDate.toLocaleString() + ", " + days + " days from now"
;
}
```

Save it, open **dates.html**, and click Preview. Now the number of days is be easier to read:



Remember that **Math** is a built-in JavaScript object with lots of handy methods you can use for doing math computations. **Math.floor()** drops all of the numbers after the decimal point in a floating point number, so you get a number that is less than (or equal to, if the number is even) the original. Compare that to **Math.ceil()** which rounds up and then drops the numbers.

Now let's try creating a date that's three days from now. Modify **dates.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var aDate = new Date();
    aDate.setFullYear(2020);
    aDate.setHours(13);
    aDate.setMinutes(3);
    var now = new Date();
    var diff = aDate.getTime() - now.getTime();
    var days = Math.floor(diff / 1000 / 60 / 60 / 24);
    datetime.innerHTML = aDate.toLocaleString() + ", " + days + " days from now"
;
    var now = new Date();
    var threeDays = (24 * 60 * 60 * 1000) * 3;
    var threeDaysFromNow = new Date(now.getTime() + threeDays);
    datetime.innerHTML = now.toLocaleString() + "; 3 days from now: " + threeDay
sFromNow.toLocaleString();
}
```

Save it, open **dates.html**, and click Preview. You see two dates: now, and three days from now.

We also use the date expressed as milliseconds since 1970 to create a date three days from now. Here, we create a **Date** representing now. Then we figure out how many milliseconds are in three days, and create another **Date** that is now (expressed in milliseconds) plus the number of milliseconds in three days. That gives us a **Date** three days from now.

Experiment by creating other dates. Can you create a date that is three days in the past from now?

Converting Strings to Dates

You might have an application, like the To-Do List application we've been building in this course, that asks the user to submit a date. In the To-Do List application, we ask the user to submit a date for when the to-do item is due.

When you submit a date using a form, whether you're using <input type="date"> or <input type="text">, the value you get when you process the input data with JavaScript is a String. But sometimes you might need that value as a **Date** object; that's when you'll use the techniques we're learning in this lesson.

The **Date** object has a method named **parse()** that you can use to parse a string representing a date. Let's see how we can use it in an example. First, we'll update our HTML to add a form entry for a date, and then update our JavaScript to process the string value we get from the form. We'll convert the string into a Date, and then display the Date in the page, in the "datetime" . Modify **dates.html** as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>Dates</title>
  <meta charset="utf-8">
  <script src="dates.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    span {
      font-weight: bold;
    }
    form {
      margin-bottom: 20px;
    }
  </style>
</head>
<body>
    <form>
      <label>Enter a date:</label>
      <input type="date" id="aDate">
      <input type="button" id="submit" value="Submit">
    </form>
    <div>
      The date and time is:
      <span id="datetime"></span>
    </div>
</body>
</html>
```

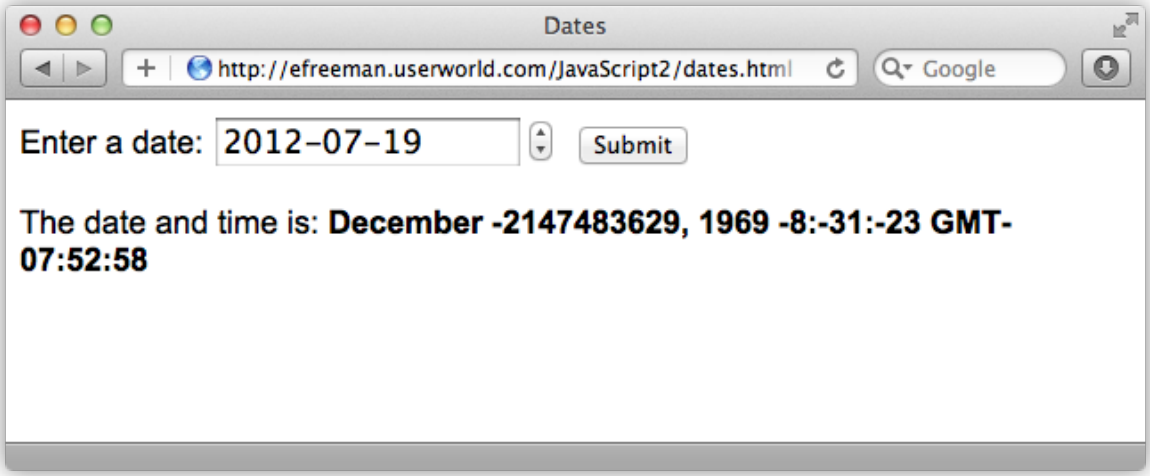
Save it, and modify **dates.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var datetime = document.getElementById("datetime");
    var now = new Date();
    var threeDays = (24 * 60 * 60 * 1000) * 3;
    var threeDaysFromNow = new Date(now.getTime() + threeDays);
    datetime.innerHTML = now.toLocaleString() + ", 3 days from now: " + threeDaysFromNo
w.toLocaleString();
    var submit = document.getElementById("submit");
    submit.onclick = getDate;
}
function getDate() {
    var aDateString = document.getElementById("aDate").value;
    if (aDateString == null || aDateString == "") {
        alert("Please enter a date");
        return;
    }
    var aDateMillis = Date.parse(aDateString);
    alert(aDateMillis);
    var aDate = new Date(aDateMillis);

    var datetime = document.getElementById("datetime");
    datetime.innerHTML = aDate.toLocaleString();
}
```

Save it, open **dates.html** again, and click Preview. Enter a date in the date input field. Click **Submit**. You first see an alert, and then you see a date displayed in the page below the form input. Try writing the date in different formats.

You probably see the string "NaN" in the alert, and you probably get a date that makes no sense in the page. If you're using a browser like Safari or Chrome that displays arrows next to the "date" input control, and you use these to enter a date, like "07-19-2012," you'll still see a nonsensical date displayed in the page.

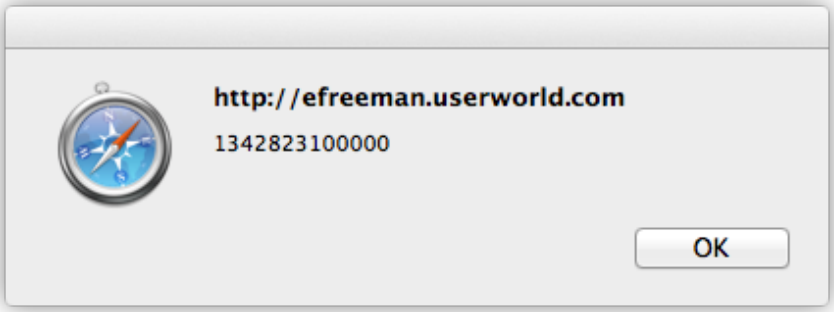


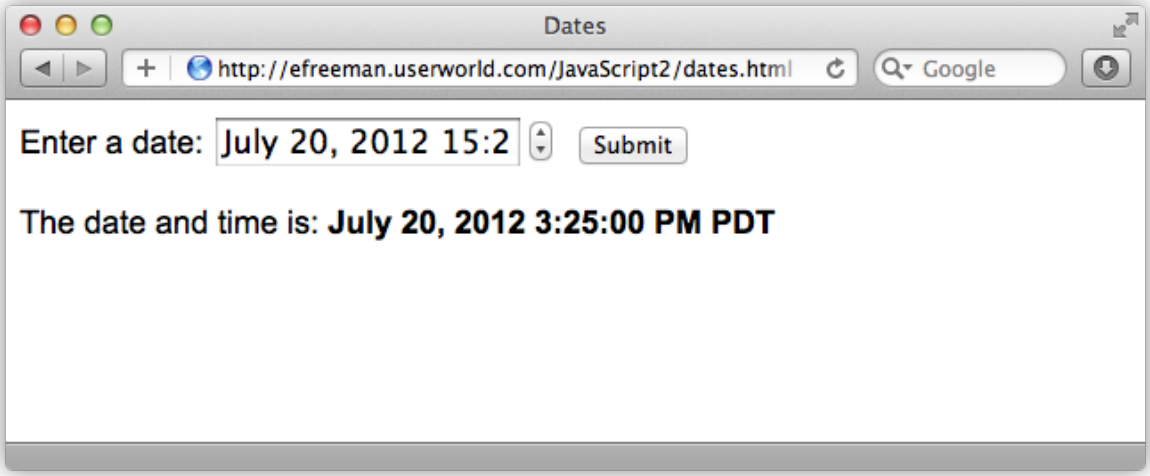
Something's definitely gone wrong, because that date doesn't make sense.

You'll likely find that this code works properly only when you enter dates in particular formats. One of the formats you can use is the same format you see when you use **toString()** or **toLocaleString()**, as we've done in these examples. Try entering a date using this format:

July 20, 2012 15:25 PDT

Now it should work. You see an alert with the number of milliseconds representing that date, and the date appears properly under the form.





Here are some other formats you can try:

- July 20, 2012
- 2012/7/20
- 2012.7.20
- 2012-7-20
- 7/20/2012
- 7-20-2012
- 7.20.2012

Take note of the different formats you try, which ones work, and which ones don't.

Let's go through the code and see what's happening:

OBSERVE:

```
function getDate() {
    var aDateString = document.getElementById("aDate").value;
    if (aDateString == null || aDateString == "") {
        alert("Please enter a date");
        return;
    }
    var aDateMillis = Date.parse(aDateString);
    alert(aDateMillis);
    var aDate = new Date(aDateMillis);

    var datetime = document.getElementById("datetime");
    datetime.innerHTML = aDate.toLocaleString();
}
```

First, we **get the value of the "aDate" input** as a string. We **check to make sure the string isn't empty**; if it is, we **alert the user** and ask them to enter a date, and then return from the function.

If we get a string, we **try to parse it** using **Date.parse()**. **Date.parse()** will return the date in milliseconds from the string you pass in, but only if the method can parse the string. As you've discovered, there are only certain string formats that **Date.parse()** will parse correctly. If **Date.parse()** fails, then instead of returning the milliseconds (a large number), it returns "NaN," meaning "Not a Number." That's JavaScript's way of letting you know it couldn't parse this string into a number. So, if you enter a date using the wrong type of string format, you see "NaN" in the **alert**.

After getting the date (or trying to get the date) in milliseconds using **Date.parse()**, we create a new **Date** object from that value. If we're successful, **aDate** will be a valid **Date** object. If not, **aDate** will be a nonsensical date, because the **Date()** constructor can't make sense of the value, "NaN".

Finally, we display the **Date** in the **"datetime"** using the **toLocaleString()** method.

Note

We've been using **Date** objects by using the **Date()** constructor to create a date object and then calling methods on that object. So what is **Date.parse()**? **Date.parse()** is an example of a **static method**. In JavaScript, functions are objects; so the **Date()** constructor function that you use to create new date objects is, itself, an object. And it so happens that that object has a method named **parse()**. You call static methods using the name of the constructor function, **Date**, but without the parentheses **()** that invoke the function. Don't worry if you don't fully understand this; this topic is more advanced JavaScript and isn't within the scope of this course. Still, you're getting a taste of the kinds of things you can do with JavaScript when you delve into *object-oriented programming*. But, that's a topic for another course...

Dates and HTML Input Types

When you enter a date into an <input> element and you parse it using the **Date.parse()** method, you must enter a date string that the **parse()** method can understand. For browsers that support the "date" <input> type, and offer a date picker for entering a date, this becomes less of an issue because the date picker usually creates the date as a string with the correct date format, a format that can be parsed using **Date.parse()**. However, if you enter a date using an invalid format, that can create an error in your code, and possibly cause your web page to malfunction.

We already checked to make sure the user is entering something into the field (by checking to see if the input is null or the empty string); we should also be checking to make sure that the user is entering a valid date. We'll tackle this using **Exception Handling** in the next lesson.

In this lesson, we've explored the JavaScript **Date** object. You've learned how to create Dates, display Dates, compare Dates, and convert strings to Dates. You've also learned a few ins and outs of time zones. As you can see, working with Dates can sometimes be tricky! But fun too, right? Take a short break and then you'll be ready to tackle more Dates in the project before you move on to the next lesson.

Handling Exceptions with Try/Catch

Lesson Objectives

When you complete this lesson, you will be able to:

- throw and catch exceptions.
- use the finally clause to execute code regardless of what happens in the try/catch.

As you have realized by now, there are plenty of things that can go wrong when writing JavaScript code! There are unexpected events, bugs, and mistyped data submitted by users. So far we've handled these *exceptional conditions* and *errors* either by ignoring them (not a good long-term plan!) or by testing with if/then/else statements to check for certain conditions.

JavaScript includes another way of handling *exceptions*: the **try/catch** statement. **Try/catch** lets you *try* some code and if something goes wrong, you can *catch* the error and do something about it.

In this lesson you'll learn how to use **try/catch** (and the optional **finally** part of this statement) to handle exceptions.

What Causes an Exception?

An exception is often caused by an error in your JavaScript that causes your code to stop executing; that is, it's an error from which the JavaScript intepreter can't easily recover. In a previous lesson, you wrote some code to see whether the result of calling **Date.parse()** on an invalid string was equal to NaN. Clearly, if you give **Date.parse()** an invalid string, that's an error (in this case, a user error for using the wrong date format), but it's not a fatal error; JavaScript is happy to set the result to NaN and proceed. Of course, later on in your code, the fact that NaN is not a real number might cause an exception, but that's another issue.

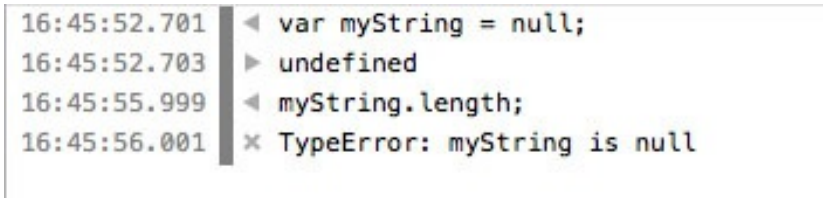
So what kind of code causes an exception that causes your code to stop running? Let's take a look at an example:

INTERACTIVE SESSION:

```
var myString = null;
myString.length;
```

Open a browser window, and open the JavaScript console (you may have to load a web page to be able to access the console; any web page will do, including a previous file you've created in the course). Type in those two lines; you see a JavaScript Error like this (in Safari):

...or like this (in Firefox):



Note Error messages in various browsers will differ slightly, but all browsers should give you an exception for this code, and a similar error message. Try different browsers to see what you get!

An error like this in your code will cause the code to stop executing. Let's try making an error in code loaded with an HTML page (rather than just at the console), and this time, we'll **try** it and **catch** the error with the **try/catch** statements. First, we'll create a super-simple HTML page, and then the JavaScript to create the error.

 Open a new HTML file and type the code as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>Exceptions</title>
  <meta charset="utf-8">
  <script src="ex.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
  </style>
</head>
<body>
</body>
</html>
```

 Save it in your **/javascript2** folder as **ex.html**. Next, create a new JavaScript file as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var myString = null;
    try {
        var len = myString.length;
        console.log("Len: " + len);
    }
    catch (ex) {
        console.log("Error: " + ex.message);
    }
}
```

 Save it in your **/javascript2** folder as **ex.js**. Open **ex.html** again, and click  . Open the JavaScript console and you see an error message like this (in Safari):

OBSERVE:

```
TypeError: 'null' is not an object (evaluating 'myString.length')
```

...or like this (in Firefox):

OBSERVE:

```
TypeError: myString is null
```

The error messages generated from your JavaScript code are the same as those you saw using the console earlier.

Let's take a closer look at the **try/catch** statement:

OBSERVE:

```
function init() {
    var myString = null;
    try {
        var len = myString.length;
        console.log("Len: " + len);
    }
    catch (ex) {
        console.log("Error: " + ex.message);
    }
}
```

After setting **myString** to null, which we know will cause an error when we try to access the **length** property (because null doesn't have a length property), we start the **try/catch** block. We use { and } to delimit each part of the statement; these are required.

JavaScript will try to execute all the code in the **try** part of the statement. If it works and no exception is created, then the **try** ends normally, and the **catch** is *not* executed. So the flow of execution would continue below the **catch**; in this case, that means the function simply returns.

But if something goes wrong, and an exception is generated, as we know it will in this code, then as soon as the exception is generated, the flow of execution jumps from the **try** to the **catch**. In this case, that means we never see the **console.log()** message that displays the value of **len**.

JavaScript automatically generates a **value for the exception** and passes it into the **catch** clause (much like passing an argument to a function parameter). For errors generated internally by the JavaScript interpreter, like this one is, that value is typically an **Error** object. Here, we assume it is such an error, and we name that object **ex**, and access its **message** property to display in the console, with information about what the error was.

So in this code, we cause, or *raise* (as it's often called), an exception by attempting to access a property that doesn't exist, and we *catch* that exception so that it doesn't cause our program to stop running entirely. This is usually better for the application; if you handle the errors that are caused in your program gracefully, then the end user can continue using your application, whereas if your JavaScript stops running, that might cause your application to stop working altogether!

Throwing Exceptions and the Finally Clause

Throwing Exceptions

You might want to use the **try/catch** statement in situations where JavaScript might not raise an exception internally, but where you are testing for errors or exceptional conditions in your code. In that case, you can raise your own exceptions by using the **throw** statement. Let's expand our example just a bit and throw our own exception (and catch it, of course). Modify **ex.html** as shown:

CODE TO TYPE:

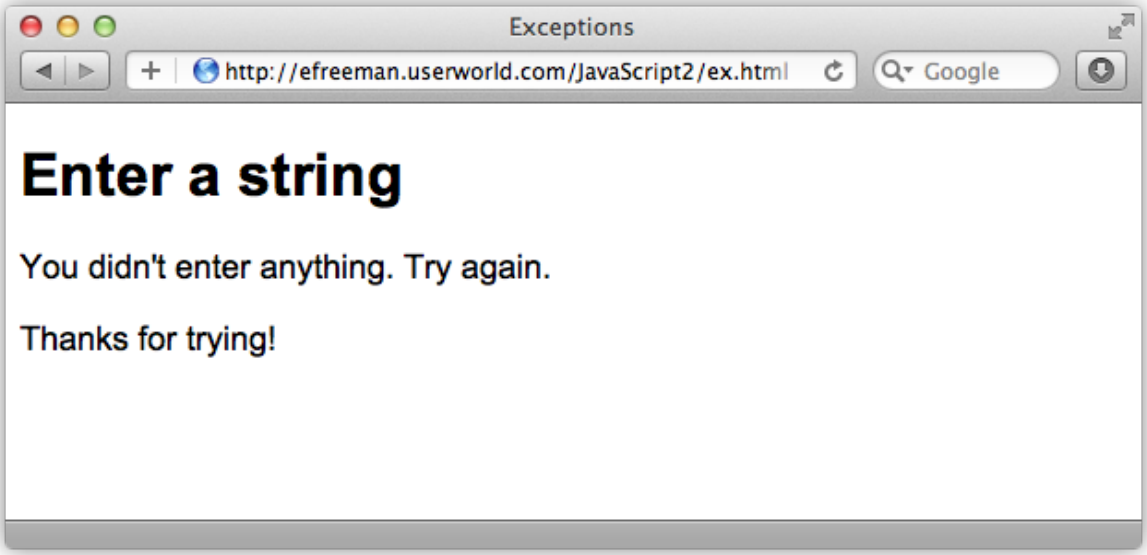
```
<!doctype html>
<html>
<head>
<title>Exceptions</title>
  <meta charset="utf-8">
  <script src="ex.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
  </style>
</head>
<body>
    <h1>Enter a string</h1>
    <p id="stringInfo"></p>
    <p id="error"></p>
    <p id="msg"></p>
</body>
</html>
```

Save it. Now, update **ex.js**:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var myString = null;
    try {
        var len = myString.length;
        console.log("Len: " + len);
    }
    catch (ex) {
        console.log("Error: " + ex.message);
    }
    var myString = prompt("Enter a string:");
    try {
        var len = myString.length;
        if (len == 0) {
            throw new Error("You didn't enter anything. Try again.");
        }
        else {
            displayLength(myString, len);
        }
    }
    catch (ex) {
        displayError(ex.message);
    }
    finally {
        displayMessage("Thanks for trying!");
    }
}

function displayError(e) {
    var error = document.getElementById("error");
    error.innerHTML = e;
}

function displayLength(myString, len) {
    var stringInfo = document.getElementById("stringInfo");
    stringInfo.innerHTML = "The string '" + myString + "' has length: " + len;
}

function displayMessage(m) {
    var msg = document.getElementById("msg");
    msg.innerHTML = m;
}
```

Save it, open **ex.html** again, and click Preview. You'll be prompted to enter a string. Try entering a real string; see what happens. Now try clicking **OK** without entering anything, an see what happens.



Let's go through the code to see what's going on:

OBSERVE:

```
function init() {
    var myString = prompt("Enter a string:");
    try {
        var len = myString.length;
        if (len == 0) {
            throw new Error("You didn't enter anything. Try again.");
        }
        else {
            displayLength(myString, len);
        }
    }
    catch (ex) {
        displayError(ex.message);
    }
    finally {
        displayMessage("Thanks for trying!");
    }
}
```

Now, instead of setting **myString** to null, we're prompting the user to enter a string. Once we've done that, we **try** to get the length of the string. When you use **prompt()**, even if you don't enter anything, the value returned is still a string (in that case, it would be an empty string, ""), and getting the length of an empty string will result in 0, rather than an exception. We can **check to see if the length is 0**, and if it is, we can **throw our own exception**. You can throw any value you want; in this case, we are throwing an **Error** object (just like JavaScript did in the previous example). As soon as we **throw the Error**, the code skips any other code in the **try** clause, and jumps to the **catch** clause. There, we pass the value of the **message** property from the **Error** object to the function **displayError()**, which displays that value in the web page. The value of **message** is the value we passed to the **Error()** constructor when we **threw the Error**.

If **the length is greater than 0**, we display the string and the length of the string by passing the two values to **displayLength()**.

The Finally Clause

We added on a **finally** clause in this example. This clause is executed whether the exception is thrown or not. In other words, if the length is 0, and we execute the **catch** clause, once the **catch** is complete, we execute the **finally** clause. If the length is greater than 0, we execute the code in the **try** clause, skip the **catch** clause (because there's no exception), and execute the **finally** clause.

Finally allows you execute some code regardless of what happens in the **try/catch**, so it's handy for clean-up code, for example. In this case, all we do is display the same message whether the prompt is successful

or not.

Using Exceptions and Try/Catch

In the previous lesson, we created a program to parse a string and convert it to a JavaScript **Date**, but we weren't checking to make sure the **Date.parse()** actually worked!

This is a good example of where we can use **try/catch** and throw an **exception** instead. Doing this will make the code more robust; it's a good way to handle this type of error. Let's give it a try. Edit **dates.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var submit = document.getElementById("submit");
    submit.onclick = getDate;
}

function getDate() {
    var aDate;
    var aDateString = document.getElementById("aDate").value;
    if (aDateString == null || aDateString == "") {
        alert("Please enter a date");
        return;
    }
    var aDateMillis = Date.parse(aDateString);
    alert(aDateMillis);
    var aDate = new Date(aDateMillis);
    try {
        if (isNaN(aDateMillis)) {
            throw new Error("Date format error. Please enter the date in the format MM/
DD/YYYY, YYYY/MM/DD, or January 1, 2012");
        }
        else {
            aDate = new Date(aDateMillis);
        }
        var datetime = document.getElementById("datetime");
        datetime.innerHTML = aDate.toLocaleString();
    }
    catch (ex) {
        alert(ex.message);
    }
}
```

Save it, open **dates.html**, and click Preview. Try entering a string using a format the program will recognize, and a string that it will not recognize. You see the same behavior you saw at the end of the previous lesson, but the way we handle the error (that is, the format that the program can't parse) is different. Now we use **try/catch** and throw an exception if we can't match the date format the user has entered.

Notice that as soon as we throw the **Error** object when we can't match the format, the rest of the **try** clause is skipped, so we don't have to check to see if an error was generated. As soon as that **Error** is thrown, the code jumps directly to the **catch** clause. This is a convenient way to tell your program to, "stop everything we're doing and go here!" This technique can be really useful. It can also make the code a bit easier to read.

Remember that the **try** clause must always be matched with either a **catch** or a **finally**. Typically, you'll see **try/catch**, but you'll find **finally** will also come in handy. If you want to raise your own exceptions, you can use **throw** and throw a value that is caught by the **catch** clause parameter. You can throw any value you want; JavaScript typically throws the **Error** object, and you can do this too by creating a **new Error** object, and passing in the value for the **message** property. Always check the JavaScript documentation in a good reference to find out exactly which type of exception to expect for circumstances where JavaScript might not throw the **Error** object, so you know which kind of value to expect in your **catch** clause.

Geolocation and Google Maps

Lesson Objectives

When you complete this lesson, you will be able to:

- use the geolocation object to get your position coordinates (latitude and longitude).
- put your location into a webpage using geolocation.
- handle geolocation errors.
- add a Google map.
- add a marker to your map.

One fairly new feature in JavaScript and web browsers is Geolocation. It's been around in various forms for a while, but it was standardized recently through the W3C in the Geolocation specification. All modern browsers (including IE9+) now support this version of Geolocation, which makes it easier to get location data into your web pages.

Geolocation is especially fun when you're using a web application on a mobile browser that supports Geolocation, because you're likely to be moving around, and more likely to be using an app where seeing your location comes in handy. Fortunately, Geolocation is supported by both the iOS browser (on iPhone and iPad) and the Android browser (on a variety of smart phones).

In this lesson we'll build a simple "Random Thoughts" application that allows you to add random thoughts to a web page. The app will capture your location when you begin adding your thoughts and add it to a map. Sound like fun? Let's get going!

How Geolocation Works

Geolocation in browsers is supported with a built-in JavaScript object named, you guessed it, **geolocation**. The **geolocation** object is a property of the **navigator** object that all browsers also have built-in. Let's see how to use the **geolocation** object to get your position coordinates (latitude and longitude). We'll start with an empty web page and fill it out in more detail as we build our Random Thoughts application. Start a new HTML file as shown:

CODE TO TYPE:

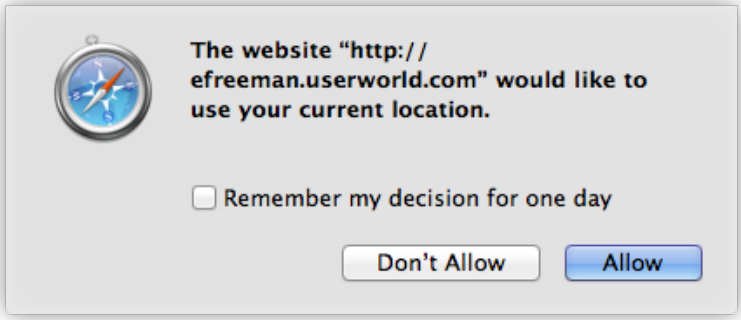
```
<!doctype html>
<html>
<head>
<title>My Random Thoughts</title>
  <meta charset="utf-8">
  <script src="random.js"></script>
  <style>
  </style>
</head>
<body>
</body>
</html>
```

Save it in your **/javascript2** folder as **random.html**. Next, create the JavaScript in a new file:

CODE TO TYPE:

```
window.onload = init;

function init() {
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(getLocation);
    }
    else {
        console.log("Sorry, no Geolocation support!");
    }
}
function getLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    alert("My position is: " + latitude + ", " + longitude);
}
```

Save it in your **/javascript2** folder as **random.js**, open **random.html** again, and click Preview. You're prompted to confirm that you're okay with sharing your location. This is to protect your privacy. Here's what the prompt looks like in Safari:



In Chrome:

In Internet Explorer:

And in Firefox:

Once you allow the browser to use your location (assuming you're okay with that, and we'll talk more later about what happens if you don't allow it), then you'll see an alert with your location:

If you don't see an alert, we'll add some code soon that you can use to help troubleshoot whatever the problem might be. Assuming you're using a modern browser, you'll see a location, even if you're on a desktop machine. However, sometimes your location might be based on your ISP's network hub rather than your actual location, so it may not be as precise on a desktop computer as it would be, say, on a phone with GPS. We'll review the various ways browsers determine your location shortly.

For now though, let's just go through the code:

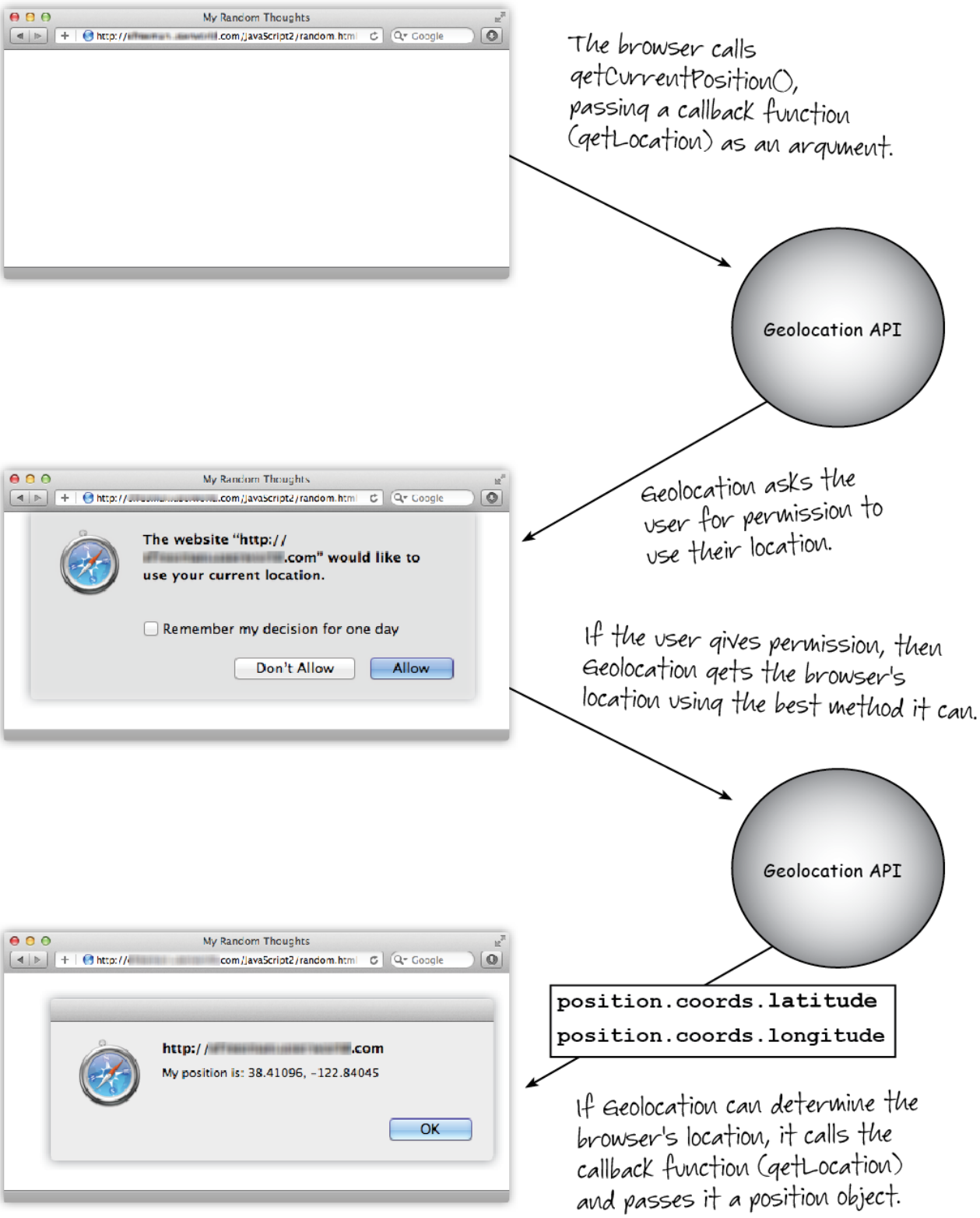
OBSERVE:

```
function init() {
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(getLocation);
    }
    else {
        console.log("Sorry, no Geolocation support!");
    }
}
function getLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    alert("My position is: " + latitude + ", " + longitude);
}
```

First, in the **init()** function that runs when the page has loaded, we **check to see if the navigator.geolocation** object exists. We know **navigator** exists (all browsers have this object), but some browsers might not have the **geolocation** object.

If the **geolocation** object exists, then we call its **getCurrentPosition()** method. The **argument we pass** to **getCurrentPosition()** is a function value, **getLocation**. If this is the first time you've seen a function passed as an argument, you might be wondering, how on earth does that work?

Well, remember that JavaScript functions are values that can be saved in properties (like we do when we say **window.onload = init**) or stored in variables (like we do when we set an object's property name to a function value), or passed as arguments to other functions. In this case, we're passing the name of a *callback function*, which we've named **getLocation**, to **getCurrentPosition**, so that Geolocation can call that function when the browser has successfully retrieved your location. Here's how it works:



Just like our **window.onload** handler function, **init()** is called when the page is loaded, the **getLocation()** function is called when the browser has retrieved your location. Remember to pass only the name of the function; do *not* write:

OBSERVE:

```
navigator.geolocation.getCurrentPosition(getLocation());
```

Why? Because that would *call* the function and try to pass the value the function returns to **getCurrentPosition()**, which is not what we want! We want to pass the function *value* to **getCurrentPosition()**, so **getCurrentPosition()** calls the callback for us.

Once the browser has retrieved your location successfully, it tells Geolocation to call your callback function, **getLocation()**, and then passes your location to this function as a **position** object.

The **position** object contains another object, **coords**, which has two properties we're interested in: **latitude** and **longitude**. These two properties are the coordinates of your location. For now, all we're doing is saving those values in variables, and using **alert()** to display them.

How the Browser Retrieves Your Location

So, what exactly is a location, and how does the browser get it? If you've ever studied a globe, or navigated on a sailboat, then you're familiar with the lines on a map or globe that represent latitude and longitude. **Latitude** is the distance north or south of the equator, and **longitude** is the distance east or west of Greenwich, England (Greenwich seems to be a popular place from which to measure), and together they identify a specific location on the Earth.

Latitude and longitude are often specified in degrees, minutes, and seconds (which you may be familiar with if you're an astronomer), or as decimal values. In the Geolocation API, we always deal with the decimal values, so the values you get from the **position** object are the decimal versions of latitude and longitude.

The browser retrieves your location using one of four methods:

- GPS: this is available on many smart phones and other GPS-enabled devices, and is the most accurate way to get your location. These devices use data from satellites to get your location. In these devices, the browser has access to the GPS information (assuming you have GPS turned on, which is a real battery drainer, so don't forget to turn it off when you don't need it!).
- Cell Phone Tower Triangulation: If you're on a cell phone without GPS (or you have it turned off), then those cell phones can still get a rough idea of your location by seeing which cell phone towers can see your phone. The more towers you're near, the more accurate your location will be.
- WiFi: Like cell phone tower triangulation, WiFi positioning uses one or more WiFi access points to compute your location. This is handy when you're indoors on your laptop.
- IP Address: If you're connected to a wired network, then this is the method you'll use (like, on your desktop computer). In this case, your IP address is mapped to a location via a location database. This has the advantage of being able to work anywhere, but it's often less accurate, because sometimes the database maps your IP address to your ISP's location, or your neighborhood, rather than your specific location.

Whatever method your device or computer uses to get your location, once that location is found, the browser can then get that back to your JavaScript code using the **position** object, and your callback function.

Getting Your Location into a Web Page with Geolocation

Okay, let's do something fun with your location. Modify **random.html** as shown:

CODE TO TYPE:

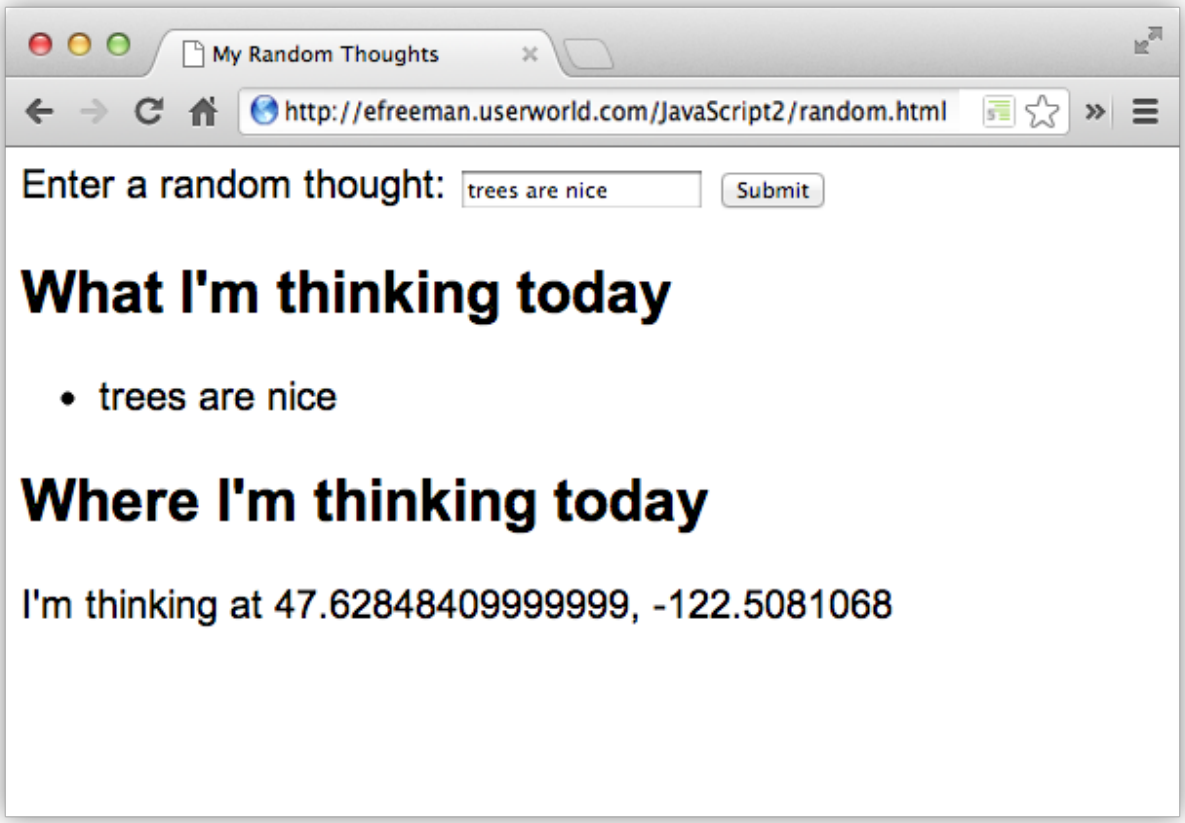
```
<!doctype html>
<html>
<head>
<title>My Random Thoughts</title>
  <meta charset="utf-8">
  <script src="random.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    form {
      margin-bottom: 20px;
    }
  </style>
</head>
<body>
    <form>
      <label>Enter a random thought:</label>
      <input type="text" id="aThought">
      <input type="button" id="submit" value="Submit">
    </form>

    <h2> What I'm thinking today </h2>
    <ul id="thoughts">
    </ul>

    <h2> Where I'm thinking today </h2>
    <div id="map">
    </div>
</body>
</html>
```

 Save it. We added a form to enter a thought, added a list, "thoughts," that we'll use to display the thoughts in the page, and a "map" , where we'll display the location of the thoughts. Don't preview yet; first, modify **random.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(getMyLocation);
    }
    else {
        console.log("Sorry, no Geolocation support!");
    }
}
function getMyLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    alert("My position is: " + latitude + ", " + longitude);
}

function Thought(id, text) {
    this.id = id;
    this.text = text;
}

function init() {
    var submit = document.getElementById("submit");
    submit.onclick = getThought;
}

function getThought() {
    var aThought = document.getElementById("aThought").value;
    if (aThought == null || aThought == "") {
        alert("Please enter a thought with at least one word");
        return;
    }
    var id = (new Date()).getTime();
    var thought = new Thought(id, aThought);

    // get the location of the thought
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(getLocation);
    }
    else {
        console.log("Sorry, no Geolocation support!");
        return;
    }

    addThoughtToPage(thought);
}

function addThoughtToPage(thought) {
    var ul = document.getElementById("thoughts");
    var li = document.createElement("li");
    li.setAttribute("id", thought.id);

    var spanText = document.createElement("span");
    spanText.setAttribute("class", "thoughtText");
    spanText.innerHTML = thought.text;

    li.appendChild(spanText);
    ul.appendChild(li);
}

function getLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var mapDiv = document.getElementById("map");
    mapDiv.innerHTML = "I'm thinking at " + latitude + ", " + longitude;
}
```

Save it, open **random.html** again, and click Preview. Enter a thought in the form input text control, and click **Submit**. You'll be prompted to confirm that you're okay with sharing your location. Approve the request to share, and see what happens. Here's what you should see if your Geolocation is working well:



Let's check out the code in a little more detail:

OBSERVE:

```
window.onload = init;

function Thought(id, text) {
    this.id = id;
    this.text = text;
}

function init() {
    var submit = document.getElementById("submit");
    submit.onclick = getThought;
}

function getThought() {
    var aThought = document.getElementById("aThought").value;
    if (aThought == null || aThought == "") {
        alert("Please enter a thought with at least one word");
        return;
    }
    var id = (new Date()).getTime();
    var thought = new Thought(id, aThought);

    // get the location of the thought
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(getLocation);
    }
    else {
        console.log("Sorry, no Geolocation support!");
        return;
    }

    addThoughtToPage(thought);
}

function addThoughtToPage(thought) {
    var ul = document.getElementById("thoughts");
    var li = document.createElement("li");
    li.setAttribute("id", thought.id);

    var spanText = document.createElement("span");
    spanText.setAttribute("class", "thoughtText");
    spanText.innerHTML = thought.text;

    li.appendChild(spanText);
    ul.appendChild(li);
}

function getLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var mapDiv = document.getElementById("map");
    mapDiv.innerHTML = "I'm thinking at " + latitude + ", " + longitude;
}
```

First, we added an object constructor function to create **Thought** objects. A **Thought** object has an id and some text that the user types into the form.

When the **user clicks the Submit button** in the form, we call the **getThought()** function and check to make sure the user really typed something in the form. If they did, we create a new **Thought** object, by first creating a unique **id** (using the time in milliseconds like we did in the lesson on Dates and Date Formatting), and then using the constructor to create a new **Thought** object, passing in the id and the text of the thought.

Next we want to get the user's location, so we can add the location position information to the page. To do that, we use the **geolocation** object again, call the **getCurrentPosition()** method, passing in the **getLocation()** function, like we did before.

You can see that **getLocation()** has one parameter, the **position** object. **getLocation()** gets your latitude and

longitude from **position**, just like before, and then updates the page with this data.

Looking back up at the **getThought()** function, after we call the **getCurrentPosition()** method of **geolocation**, we call another function, **addThoughtToPage()**, passing the thought that needs to be added to the page. The **addThoughtToPage()** function adds the thought information to the page by creating a new <li> element and adding the data from the **thought** object it's been passed: the id and the text of the thought.

Handling Errors

So, what if something goes wrong when the browser tries to get your location? Or, what happens if you don't allow your position to be shared with the browser? If you haven't been able to see any location data, the call to **getCurrentPosition()** is likely failing and no location is being retrieved. If you have been getting your location successfully, try shift-reloading the page, and deny the request from the browser to use your location. What happens?

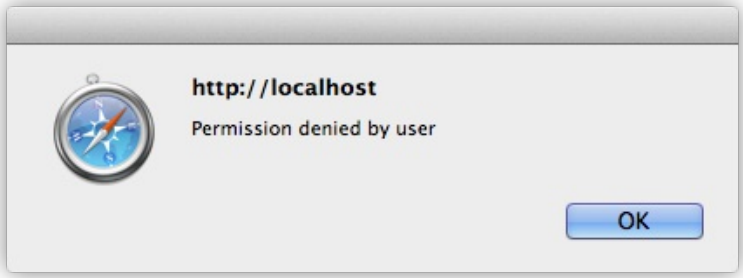
It would be nice for your application to know a little bit more about what went wrong. We can pass a second callback function, an **error callback function**, to **getCurrentPosition()** that will be called if **getCurrentPosition()** is unable to retrieve a location from the browser. Let's see how that works. Modify **random.js** as shown:

CODE TO TYPE:

```
window.onload = init;

function Thought(id, text) {
    this.id = id;
    this.text = text;
}

function init() {
    var submit = document.getElementById("submit");
    submit.onclick = getThought;
}

function getThought() {
    var aThought = document.getElementById("aThought").value;
    if (aThought == null || aThought == "") {
        alert("Please enter a thought with at least one word");
        return;
    }
    var id = (new Date()).getTime();
    var thought = new Thought(id, aThought);

    // get the location of the thought
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(getLocation, locationError);
    }
    else {
        console.log("Sorry, no Geolocation support!");
        return;
    }

    addThoughtToPage(thought);
}

function addThoughtToPage(thought) {
    var ul = document.getElementById("thoughts");
    var li = document.createElement("li");
    li.setAttribute("id", thought.id);

    var spanText = document.createElement("span");
    spanText.setAttribute("class", "thoughtText");
    spanText.innerHTML = thought.text;

    li.appendChild(spanText);
    ul.appendChild(li);
}

function getLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var mapDiv = document.getElementById("map");
    mapDiv.innerHTML = "I'm thinking at " + latitude + ", " + longitude;
}

function locationError(error) {
    var errorTypes = {
        0: "Unknown error",
        1: "Permission denied by user",
        2: "Position not available",
        3: "Request timed out"
    };
    var errorMessage = errorTypes[error.code];
    if (error.code == 0 || error.code == 2) {
        errorMessage += " " + error.message;
    }
    console.log(errorMessage);
    alert(errorMessage);
```

```
}
```

Save it, open **random.html** again, and click Preview. If you were not seeing location information before, you should see an alert now with more information about what went wrong. If you were seeing location information before, go ahead and deny the browser's request to use your location, and again, you should see an alert with the error message, like this:



Note

Some browsers may prompt you to *Always* or *Never* allow the location information, so if you deny the request, you may need to go to the appropriate setting in the browser to set it back to allow or prompt for permission. In some browsers, you can just reload the page, or close and restart **random.html**. Others may require that you change the option back in the Tools area.

To reset the permission in Firefox, select **Tools | Page Info | Permissions** and change the permission for **Share Location**.

To reset the permission in Chrome, select **Chrome | Preferences | Settings**, click on **Advanced Permissions**, then **Privacy | Content Settings | Location** and change the permission to **Ask me when a site tries to track my physical location**. Then click on **Manage Expectations** to see which sites are listed to Allow or Block automatically, and remove your oreillystudent.com site if it's listed there, so the browser will prompt you to Allow or Deny the next time you try.

Let's take a look at the changes:

OBSERVE:

```
...
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(getLocation, locationError);
    }
.
.
.
function locationError(error) {
    var errorTypes = {
        0: "Unknown error",
        1: "Permission denied by user",
        2: "Position not available",
        3: "Request timed out"
    };
    var errorMessage = errorTypes[error.code];
    if (error.code == 0 || error.code == 2) {
        errorMessage += " " + error.message;
    }
    console.log(errorMessage);
    alert(errorMessage);
}
```

Here we added a second argument to the call to **getCurrentPosition()**, passing in a second callback function, **locationError()**. The *error callback handler* is passed information from the browser when it's called: an **error** object. The **error** object has a property, **code**, that contains a number representing the type of error. In **locationError()**, we map that **code** to an error message using an **errorTypes** object literal. If the error **code** is 0 or 2, sometimes we can retrieve more information about what went wrong in the **error** object's **message** property. We then display the full

error message both to the console and in an alert().

If you cause an error by denying the browser's request to use your location, you generate an error code of type 1, and you see the message "Permission denied by user." If you see one of the other error messages, the browser is unable to access your position for some other reason. It could be that your signal isn't strong enough on your cell phone (or you're out of range of a cell tower), or your GPS is turned off, or your ISP isn't mapped to a location in the location database, for instance.

If you see an error message now, what does it say? If you're approving the request to share your location, but still getting an error, try your program on another computer or device if you have one available. Ideally, your program works successfully at this point, and you won't need the error handling code, but it's still good to have it in there.

Note that even if **locationError()** is called, we still add the thought to the page, using **addThoughtToPage()**, so the basic app still works; it just doesn't show you a location for the thought.

Adding a Google Map

Wouldn't it be a lot more fun if we could see the location of our thoughts on a map? Let's make that happen. We'll use the Google Maps API. We'll need to link to the Google Maps JavaScript library and add a for the map to your HTML. Modify **random.html** as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
<title>My Random Thoughts</title>
  <meta charset="utf-8">
  <script src="http://maps.google.com/maps/api/js?sensor=true"></script>
  <script src="random.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
    }
    form {
      margin-bottom: 20px;
    }
    div#map {
      width: 400px;
      height: 400px;
    }
  </style>
</head>
<body>
    <form>
      <label>Enter a random thought:</label>
      <input type="text" id="aThought">
      <input type="button" id="submit" value="Submit">
    </form>

<h2> What I'm thinking today </h2>
    <ul id="thoughts">
    </ul>

    <h2> Where I'm thinking today </h2>
    <div id="map">
    </div>
</body>
</html>
```

Save it. We added the link to the Google Maps API JavaScript in the line:

OBSERVE:

```
<script src="http://maps.google.com/maps/api/js?sensor=true"></script>
```

This includes all the JavaScript for the Google Maps API, so when you use the Google functions to create a map, the browser will be able to find the JavaScript. **sensor=true** on the end of the URL is *required*. It tells the maps API that we're using the browser's Geolocation capabilities to get our location.

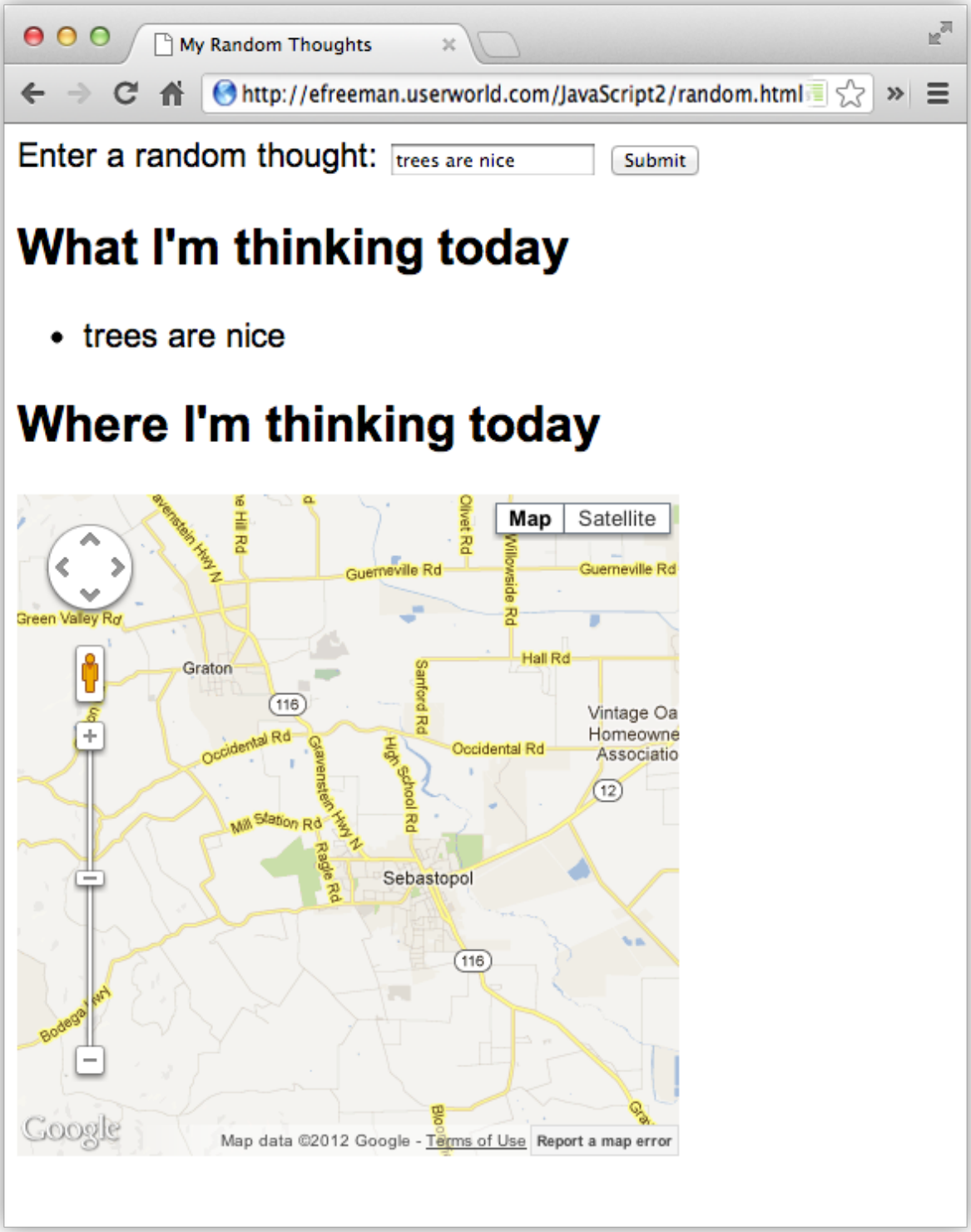
Next, we'll use the JavaScript functions in the Google Maps API to add a map to our page. Modify **random.js** as shown:

CODE TO TYPE:

```
window.onload = init;

var map = null;

function Thought(id, text) {
    this.id = id;
    this.text = text;
}

function init() {
    var submit = document.getElementById("submit");
    submit.onclick = getThought;
}

function getThought() {
    var aThought = document.getElementById("aThought").value;
    if (aThought == null || aThought == "") {
        alert("Please enter a thought with at least one word");
        return;
    }
    var id = (new Date()).getTime();
    var thought = new Thought(id, aThought);

    // get our location
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(getLocation, locationError);
    }
    else {
        console.log("Sorry, no Geolocation support!");
        return;
    }

    addThoughtToPage(thought);
}

function addThoughtToPage(thought) {
    var ul = document.getElementById("thoughts");
    var li = document.createElement("li");
    li.setAttribute("id", thought.id);

    var spanText = document.createElement("span");
    spanText.setAttribute("class", "thoughtText");
    spanText.innerHTML = thought.text;

    li.appendChild(spanText);
    ul.appendChild(li);
}

function getLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    var mapDiv = document.getElementById("map");
    mapDiv.innerHTML = "I'm thinking at " + latitude + ", " + longitude;
    if (!map) {
        showMap(latitude, longitude);
    }
}

function showMap(lat, long) {
    var googleLatLong = new google.maps.LatLng(lat, long);
    var mapOptions = {
        zoom: 12,
        center: googleLatLong,
        mapTypeId: google.maps.MapTypeId.ROADMAP
    };
    var mapDiv = document.getElementById("map");
```

```
    map = new google.maps.Map(mapDiv, mapOptions);
    map.panTo(googleLatLong);
}

function locationError(error) {
    var errorTypes = {
        0: "Unknown error",
        1: "Permission denied by user",
        2: "Position not available",
        3: "Request timed out"
    };
    var errorMessage = errorTypes[error.code];
    if (error.code == 0 || error.code == 2) {
        errorMessage += " " + error.message;
    }
    console.log(errorMessage);
    alert(errorMessage);
}
```

 Save it, open **random.html** again, and click  . Enter a thought and click **Submit**. A map appears!



Let's look at how we added the map in a bit more detail:

OBSERVE:

```
var map = null;
.
.
.
function getLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    if (!map) {
        showMap(latitude, longitude);
    }
}

function showMap(lat, long) {
    var googleLatLong = new google.maps.LatLng(lat, long);
    var mapOptions = {
        zoom: 12,
        center: googleLatLong,
        mapTypeId: google.maps.MapTypeId.ROADMAP
    };
    var mapDiv = document.getElementById("map");
    map = new google.maps.Map(mapDiv, mapOptions);
    map.panTo(googleLatLong);
}
```

First, we create a global **map** variable to hold the map. We want only one **map** object, so we're just going to create it once, store it in this global variable, and check each time we add a new thought to make sure we're using that same map.

When the browser calls **getLocation()** with your position, we check to see if the **map** object has been created yet. If it hasn't, we call **showMap()** to create the map. We pass the **latitude** and **longitude** objects we retrieved from the **position** object to **showMap()** and then use those to create the map.

The **showMap()** function uses the Google Maps API to create a **google.maps.LatLng** object from the latitude and longitude we passed in to the function. We then use that **google.maps.LatLng** object to create a set of options we'll use to create the actual map; these options tell the map things like the zoom level (how zoomed in or out you are on the map; the higher the number, the closer the zoom), where to center the map, and the type of map you want (ROADMAP, SATELLITE, TERRAIN, or HYBRID).

We then get the "map" in our HTML and use that, along with the mapOptions, to create a **google.maps.Map** object, which we store in the **map** global variable. Finally, we center the map on the latitude and longitude, by calling the **map**'s **panTo()** method.

Note that this code is only called the first time you add a thought because we want to create the map just once. The next time you add a thought, this code is skipped.

Now, if you're sitting at your desk and not moving around, all your thoughts will have the same location, so your map won't move. But if you are running this app on your smart device and moving around, then each thought will have a different location and you'll see the map pan to a different location each time you add a new thought from a different position. If you have a device and can go mobile, give it a try! Just enter the URL of your application at oreillystudent.com into the mobile browser. It will be something like this:

http://*yourusername*.oreillystudent.com/javascript2/random.html

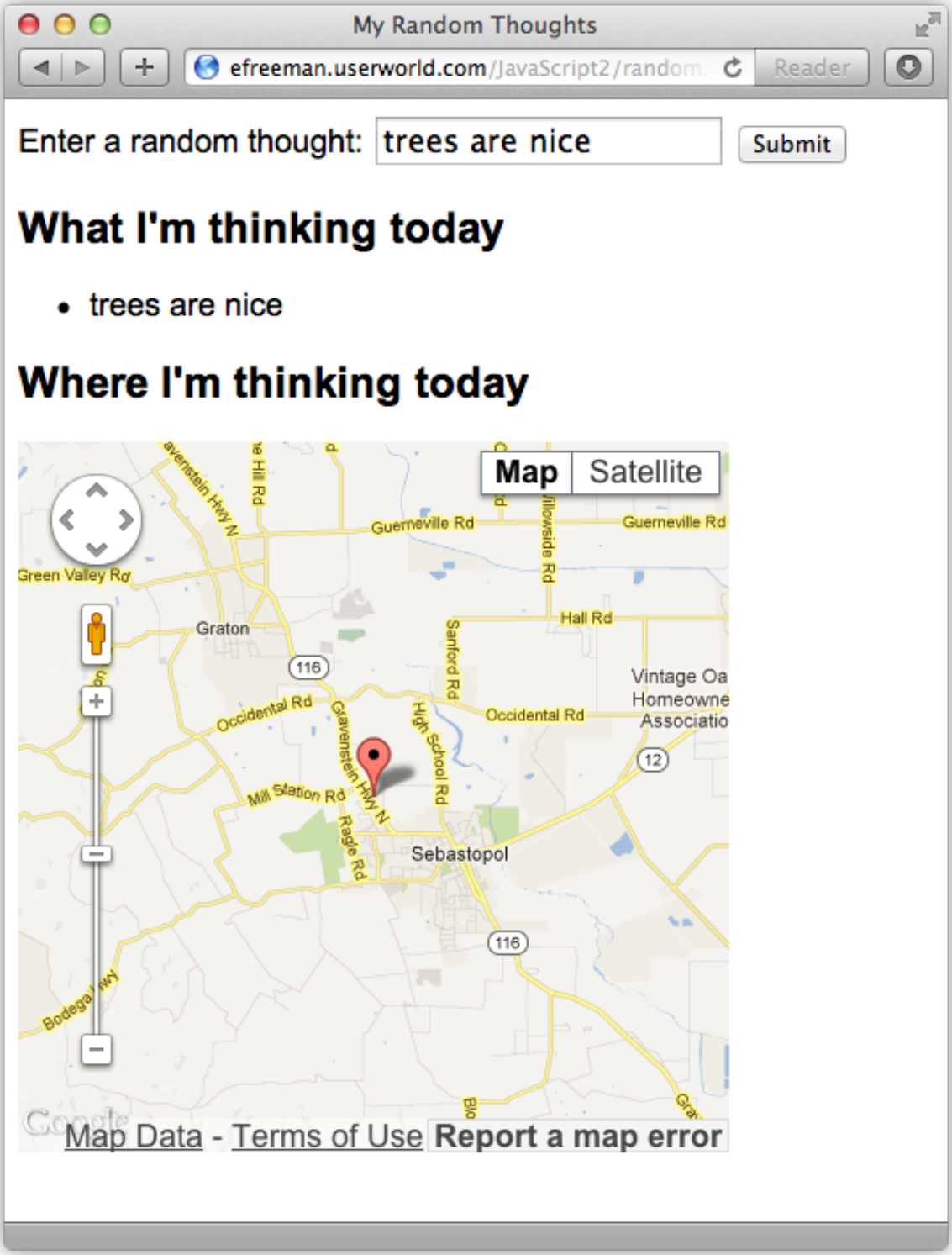
Adding a Marker to the Map

This map would be a whole lot more useful if we could see the precise location of our thoughts, right? So before we end this lesson, let's add a marker for each thought to the map. Modify **random.js** as shown:

CODE TO TYPE:

```
.
.
.
function getLocation(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;
    if (!map) {
        showMap(latitude, longitude);
    }
    addMarker(latitude, longitude);
}
.
.
.

function addMarker(lat, long) {
    var googleLatLong = new google.maps.LatLng(lat, long);
    var markerOptions = {
        position: googleLatLong,
        map: map,
        title: "Where I'm thinking today"
    }
    var marker = new google.maps.Marker(markerOptions);
}
.
.
.
```

Save it, open **random.html** again, and click Preview. Now when you add a new thought and it appears on the map, you see a nice red marker showing you exactly where you were when you added the thought. Again, if you add multiple thoughts at your desk, you'll see only one marker, because all the markers will sit right on top of each other because you aren't moving around.



To add a marker, we call a new function, **addMarker()**, from **getLocation()**, passing in the latitude and longitude:

OBSERVE:

```
function addMarker(lat, long) {
    var googleLatLong = new google.maps.LatLng(lat, long);
    var markerOptions = {
        position: googleLatLong,
        map: map,
        title: "Where I'm thinking today"
    }
    var marker = new google.maps.Marker(markerOptions);
}
```

addMarker() creates **googleLatLong** and **google.maps.Marker** objects. In the **markerOptions**, we give the marker object the position where it should be located, the map to which it should be added, and a title containing some text.

Another action-packed lesson! In this lesson, you learned the basics of the Geolocation and Google Maps APIs, and used these APIs to create a nice little application that lets you add thoughts to a web page, and a map. If you're able to test the application on a mobile device, we highly recommend it, so you can see your thoughts appear on the map in different locations.

Practice your Geolocation and mapping skills a bit before moving on to the final lesson. Hang in there; you're almost done!

Dates and Date Formatting

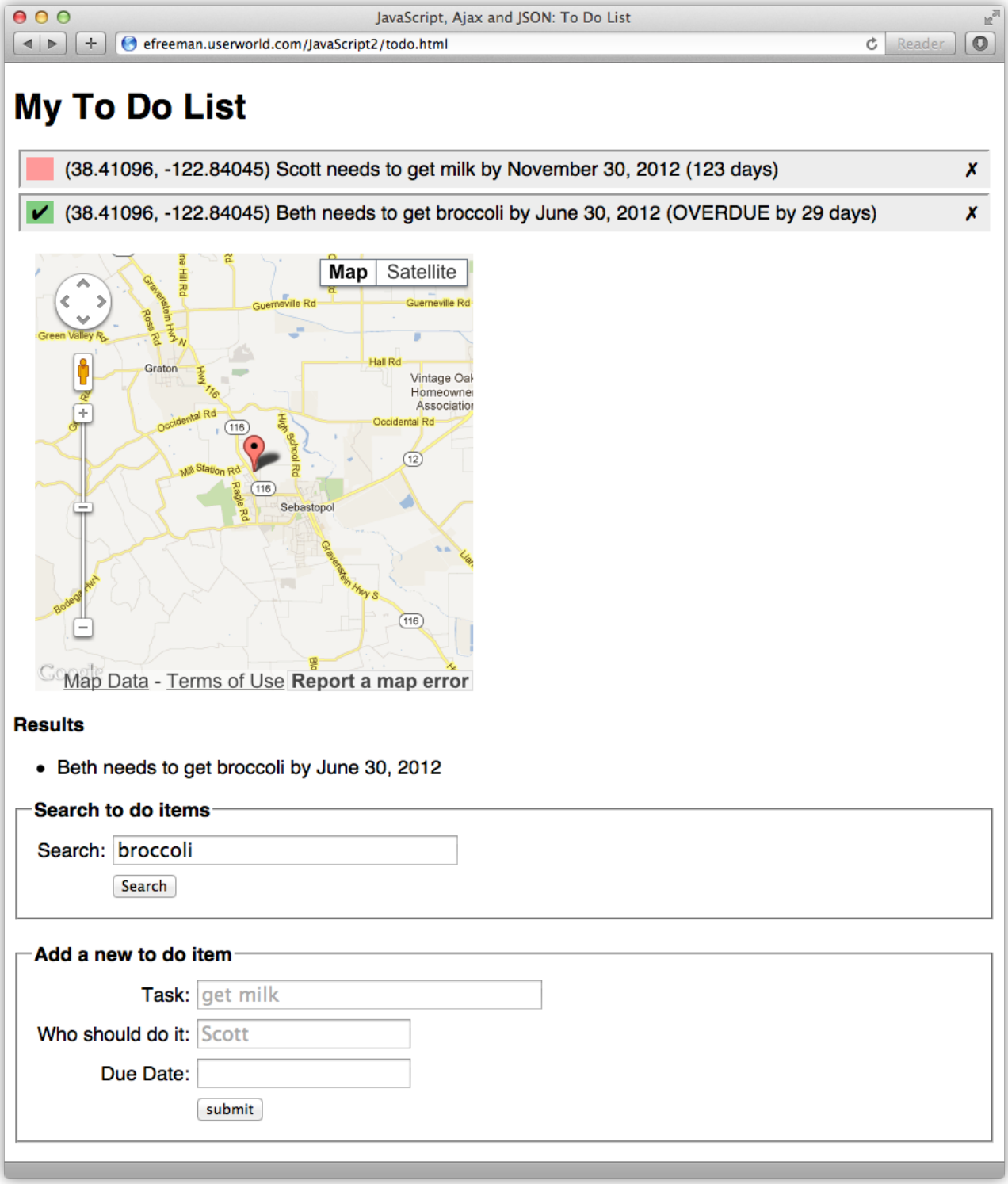
Final Project

In your final project, you're going to add more features to the To-Do List application based on what you've learned in the previous few lessons.

Start with the code from your project from the Strings lesson. Your completed To-Do List application should support the following features:

- Use Local Storage (not Ajax) for storing to-do items.
- Support deleting to-do items.
- Support marking to-do items as done.
- Support a basic text search over the "task" and "who" fields (so I can search by person or word in task).
- Support for dates, showing how many days until a task is due, or how many days overdue a task is.
- Use Exception handling for potential errors in Date processing.
- Support Geolocation, so a task has a location associated with it.
- Use Modernizr to separate Local Storage and Geolocation code from the main code. Your application should still function properly (although, obviously, with less capability) even if Local Storage and Geolocation are not supported.

Here's how your final To Do List application might look:



Below, we've included code you can begin with if you don't want to use your existing code. You'll add any new features listed above to the code below (which is from the Strings lab).

Document your code by adding comments explaining what you're doing and why. Submit all your files once you have the application working, including:

- Your HTML file.
- Your CSS file.
- Your JavaScript files (you'll have five files).

See the project instructions for initial code to help you get started. Good luck! As always, be sure to email your instructor at learn@oreillyschool.com if you need additional guidance.