The Document Fragment Object

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>
```

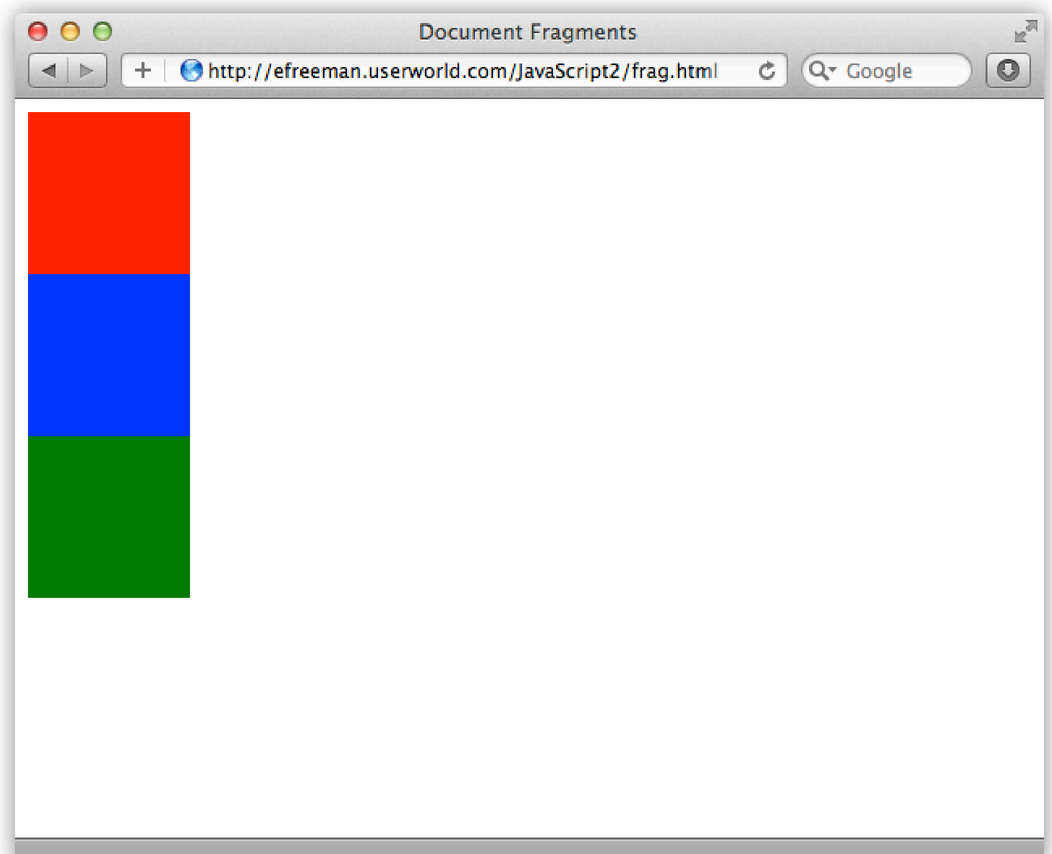
Save this file as *frag.html* in your work folder. This page doesn't contain much of anything, so don't load it into the browser until after you've added the JavaScript to create the new elements and add them to the page.

Take note of the CSS in the document. We'll use this CSS to style the new `<div>` 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 this file as *frags.js* in your work folder. Now open your *frag.html* file in your browser, which links to this *frag.js* file at the top (with the `<script>` element), and click. You should see a page like this:

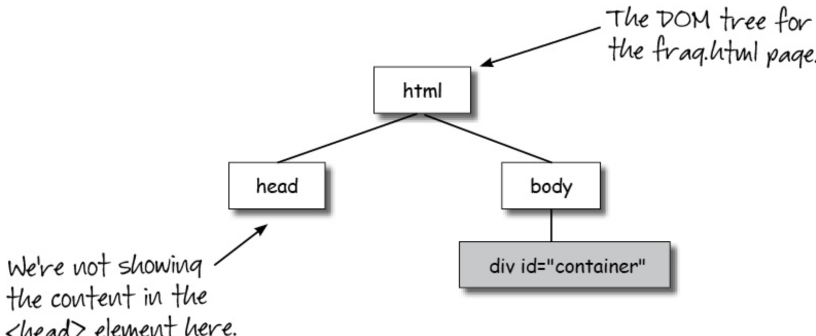


The JavaScript here is fairly straightforward. We use a loop to create three new `<div>` elements, each with the class "box", and add them one at a time to the DOM by calling

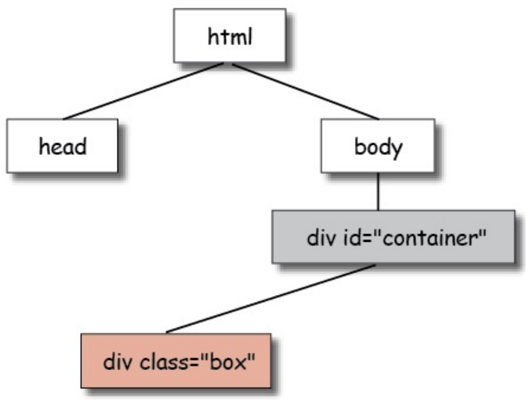
OBSERVE:

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

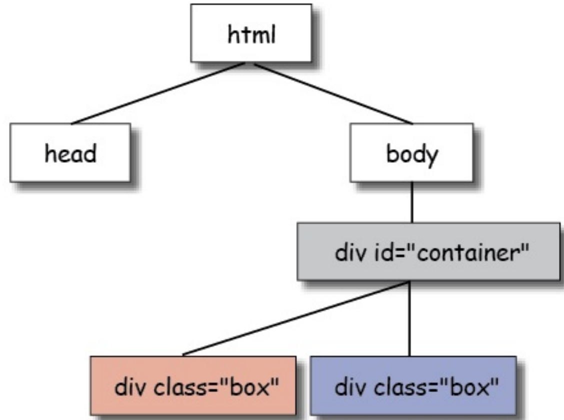
Because container is an element in the DOM, each time you call `appendChild(box)`, you add a new `<div>` element directly to the DOM:



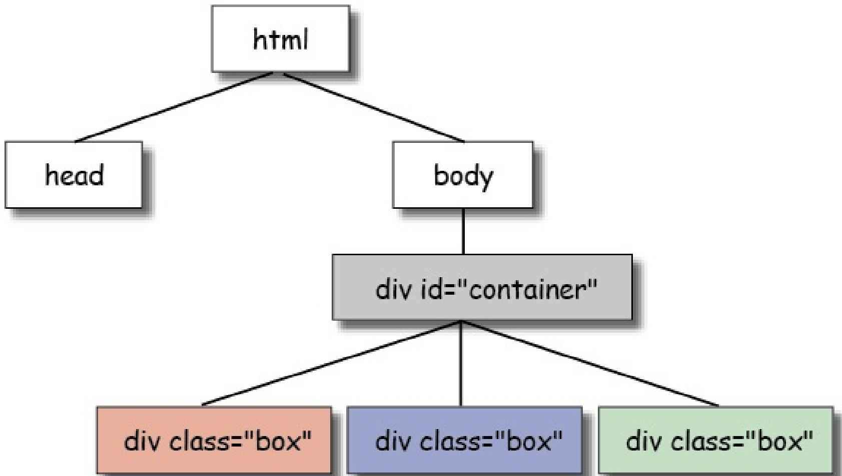
The first time through the loop, we add the red "box" `<div>`:



Then the blue:



And finally the green:



For our small loop of three `<div>` elements this is fine, but imagine you're adding 100 or 1000 `<div>` 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 the changes to *frags.js*. Then open or refresh your *frag.html* file in the browser. 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:

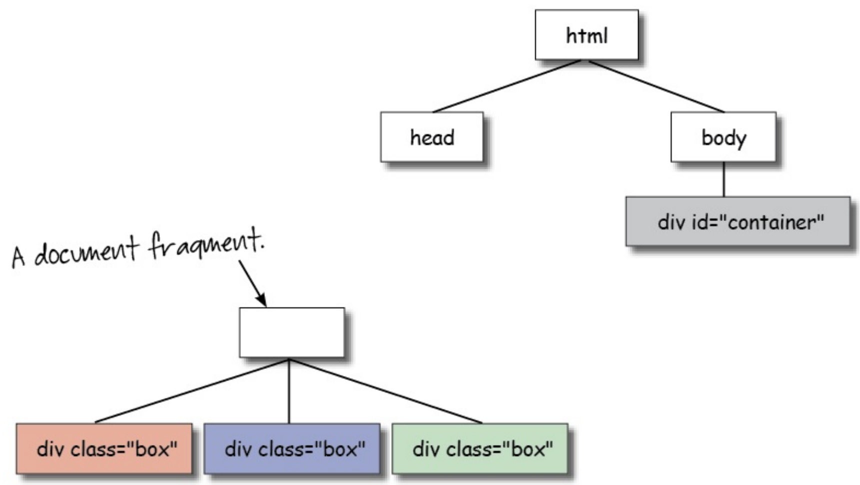
OBSERVE:

```
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" `<div>` 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.

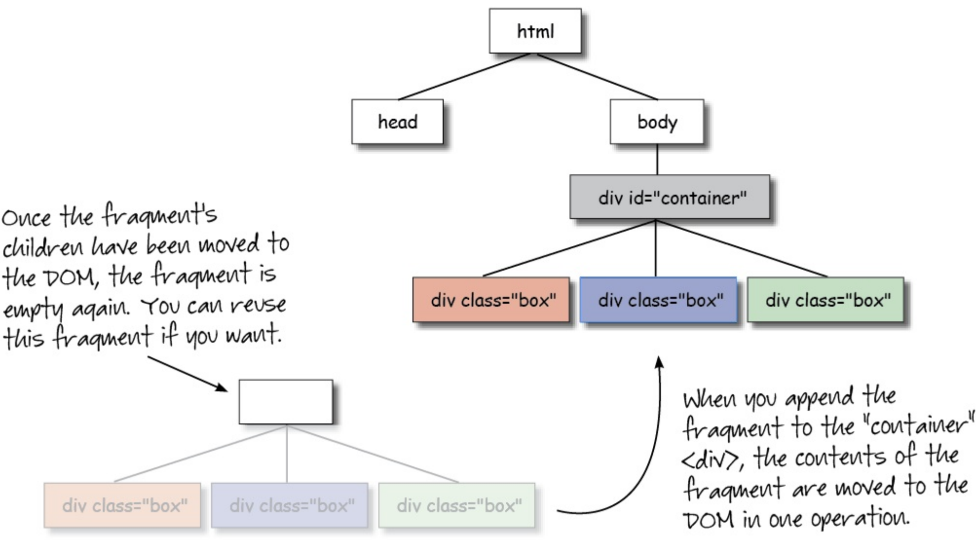
Next we loop as before. Only this time each time through the loop, we add a new "box" `<div>` to the fragment rather than the DOM. We still use the `appendChild()` method as usual, because a fragment object is just like an element object and as such. It has all the same methods.

Once we've completed the loop, here's what we've got: we have a DOM tree with an empty "container" `<div>`, and a document fragment sitting off separately from the DOM with three "box" `<div>`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" `<div>` 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" `<div>` contains the three "box" `<div>`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:

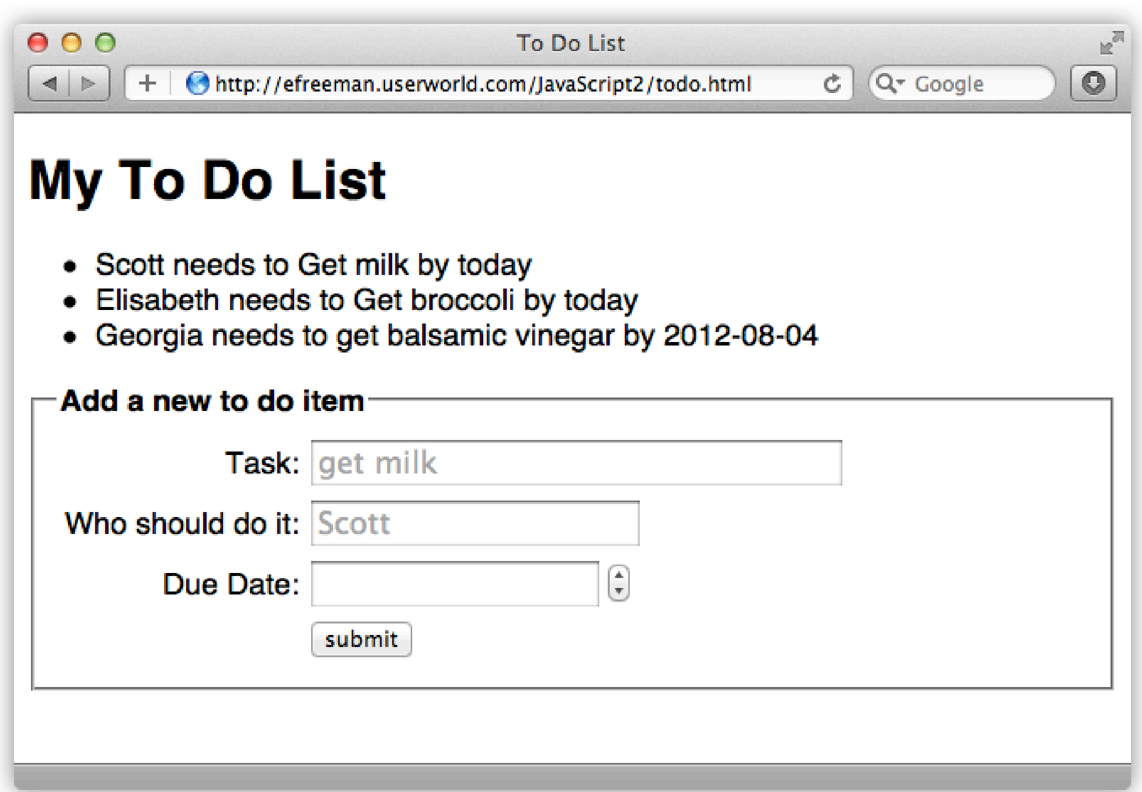
CODE TO TYPE:

```
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.dueDate;
            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 the changes to your *todo.js* file. Then open your *todo.html* file in your browser. 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 these changes, and open or refresh *todo.html* in the browser. 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. Then we 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 `<span>` 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 `<span>` 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 `<span>` 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 `<span>` 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` `<span>` element. If it *is* done, then we add the class "`done`". Also notice that we're setting the `innerHTML` of the "`notDone`" `<span>` to five spaces—` ` is the HTML entity for a non-breaking space—and setting the `innerHTML` of the "`done`" `<span>` 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.

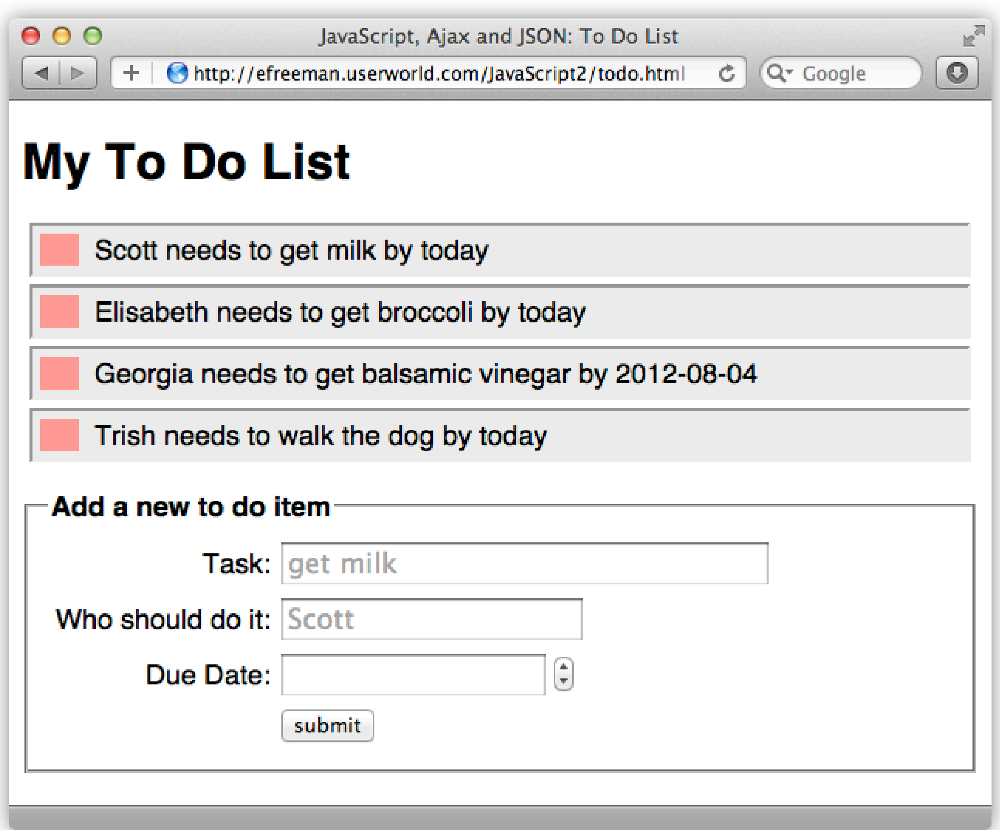
After creating the `<li>` element and the two `<span>` elements, we add the two `<span>` elements to the `<li>`. The first is the `spanDone`, so we can show if an item is done or not on the left part of the to-do item, and then the `spanTodo`, which contains the text of the to-do item (the same text 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 `<span>` 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 the changes to your *todo.css* file, and open or refresh your *todo.html* file in the browser. Now you should see your to-do items looking a lot snazzier:

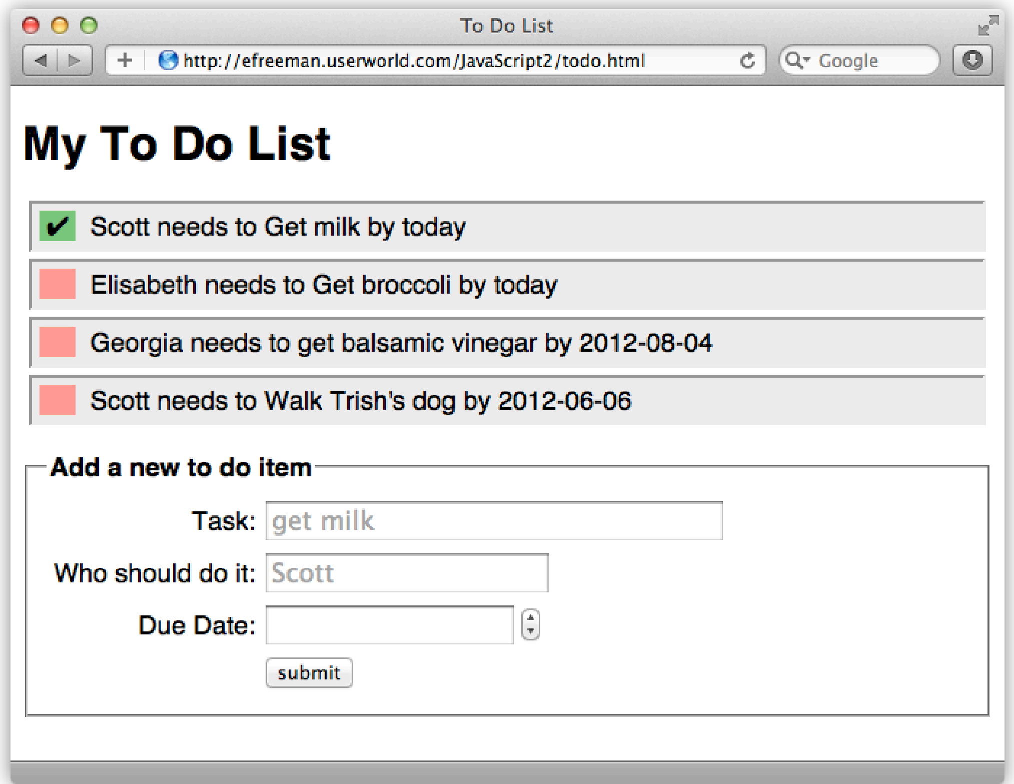


To test the "`done`" vs. "`notDone`" code, make sure you have one to-do item in your *todo.json* file that uses "true" for the `done` property. You can do this by editing your *todo.json* file and updating the code. For instance, I updated the first item in my file, so my JSON looks like this:

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!), refresh the page in your browser. 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).

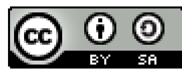
Now, you've probably noticed that the only way to mark an item as "done" is to edit your *todo.json* file, which doesn't seem like a very user-friendly process. Don't worry about that for now, we'll come back to this issue in a later lesson.

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 frequently as you build it, to see if it needs refactoring.

More about the material in this lesson

The Riverside JS Workshop would like to acknowledge the generosity of O'Reilly Media, Inc. for making this material available to us through the Creative Commons License. We would also like to acknowledge the great work of Elisabeth Robson who authored this content. She is one of our favorites. Elisabeth has written a number of *Head First* programming books for web developers. We recommend them highly. You can browse her work here.



Copyright © 1998-2015 O'Reilly Media, Inc.
This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See http://creativecommons.org/licenses/by-sa/3.0/legalcode for more information.

The original source document has been altered. It has been edited to accommodate this format and enhance readability.