Riverside JS Workshop
JavaScript Skills Ladder, Lesson 1.15

Working with DOM Elements, pt. 2

Lesson Objectives

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

- *add a click handler to multiple links.*
- *add an image to your page.*
- *remove elements from your page.*
- *retireve a child's parent.*
- *use text nodes.*

When you finished the previous lesson, you had a web application (The sample app can be downloaded here: Photo Viewer App.) to show and hide a menu and display images. You can click on the images in the menu, but the link opens the image in a new page rather than in the same page where the menu is. Our goal is to open the image in the *same* page. To do that, we need to add the image to the page. We'll do that by combining some of the new things we've learned about JavaScript in the past couple of lessons: `createElement()`, `getAttribute()`, `setAttribute()`, `appendChild()`, and `querySelectorAll()`, with a method you haven't used yet, `removeChild()`, that allows you to remove elements from the DOM (the opposite of `appendChild()`).

Add Click handler to Multiple Links

First, make sure you have your HTML. You can use the same file from the previous lesson, or type it in again. Make sure you're linking to the correct CSS and JavaScript files; if you want to create a new JavaScript file, that's fine, just remember to change the name in the `<script>` link.

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>Photo Viewer</title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="photo_viewer.css">
  <script src="photo_viewer.js"></script>
</head>
<body>
```

```
<nav>
<ul>
    <li><span id="alcatraz">Alcatraz</span>
        <ul id="alcatrazList">
            <li><a href="./img/alcatraz_1.jpg">Alcatraz 1</a></li>
            <li><a href="./img/alcatraz_2.jpg">Alcatraz 2</a></li>
            <li><a href="./img/alcatraz_3.jpg">Alcatraz 3</a></li>
        </ul>
    </li>
    <li><span id="milan">Milan</span>
        <ul id="milanList">
            <li><a href="./img/milan_1.jpg">Milan 1</a></li>
            <li><a href="./img/milan_2.jpg">Milan 2</a></li>
            <li><a href="./img/milan_3.jpg">Milan 3</a></li>
        </ul>
    </li>
    <li><span id="apostles">12 Apostles</span>
        <ul id="apostlesList">
            <li><a href="./img/12_apostles_1.jpg">12 Apostles 1</a></li>
            <li><a href="./img/12_apostles_2.jpg">12 Apostles 2</a></li>
            <li><a href="./img/12_apostles_3.jpg">12 Apostles 3</a></li>
        </ul>
    </li>
</ul>
</nav>
<div id="image">
</div>
</body>
```

The first step to getting the links to work is to add click handlers to all the links. We'll do this in a slightly more sophisticated way than how we added the click handlers to the main menu items, and it will save us typing because we can use the *same* click handler function for each link! Edit your *photo_viewer.js* file as shown:

CODE TO TYPE:

```
window.onload = init;
function init() {

    var alcatrazSpan = document.getElementById("alcatraz");
    var milanSpan = document.getElementById("milan");
    var apostlesSpan = document.getElementById("apostles");

    alcatrazSpan.onclick = selectAlcatraz;
    milanSpan.onclick = selectMilan;
    apostlesSpan.onclick = selectApostles;

    var links = document.querySelectorAll("a");
    for (var i = 0; i < links.length; i++) {
        links[i].onclick = addImage;
    }
}
```

```
function selectAlcatraz() {
    var ul = document.getElementById("alcatrazList");
    showHide(ul);
}

function selectMilan() {
    var ul = document.getElementById("milanList");
    showHide(ul);
}

function selectApostles() {
    var ul = document.getElementById("apostlesList");
    showHide(ul);
}

function showHide(el) {

    // deselect everything
    var selectedItems = document.querySelectorAll(".show");
    for (var i = 0; i < selectedItems.length; i++) {
        if (selectedItems[i] != el) {
            selectedItems[i].setAttribute("class", "");
        }
    }

    var ulClass = el.getAttribute("class");
    if (ulClass == "show") {
        // item is selected, so deselect it
        el.setAttribute("class", "");
    }
    else {
        // item is not selected, so select it
        el.setAttribute("class", "show");
    }
}

function addImage(e) {
    // we'll fill this in in the next step.
    return false;
}
```

Save it, and load *photo_viewer.html* into a browser. You'll notice that now when you click on the image links, nothing happens. In fact, the `addImage()` function is being called (because we've added this function as the click handler for *all* the links), and all we're doing in `addImage()` so far is returning the value `false`. What this does is keep the browser from following the link, which would take you to a new page containing the image. We'll update `addImage()` in the next step.

For now, focus on understanding how we added the `addImage()` function as the click handler for all the links.

OBSERVE:

```
    var links = document.querySelectorAll("a");
    for (var i = 0; i < links.length; i++) {
        links[i].onclick = addImage;
    }
```

We use the `document.querySelectorAll()` method to select all the `<a>` elements in the page. As you know from the previous lesson, this returns a collection of elements. In this case, it returns a collection with nine elements (because we have nine `<a>` elements in the page). So we loop through these elements and add `addImage()` as the click handler for each one.

Add an Image to the Page

The next step is to update our `addImage()` function to add the image to the page. We'll use methods you're already familiar with: `createElement()` and `appendChild()`. We'll also use a new object, the *event object*, to get information about which image link was clicked.

CODE TO TYPE:

```
window.onload = init;

function init() {

    var alcatrazSpan = document.getElementById("alcatraz");
    var milanSpan = document.getElementById("milan");
    var apostlesSpan = document.getElementById("apostles");

    alcatrazSpan.onclick = selectAlcatraz;
    milanSpan.onclick = selectMilan;
    apostlesSpan.onclick = selectApostles;

    var links = document.querySelectorAll("a");
    for (var i = 0; i < links.length; i++) {
        links[i].onclick = addImage;
    }
}

function selectAlcatraz() {
    var ul = document.getElementById("alcatrazList");
    showHide(ul);
}

function selectMilan() {
     var ul = document.getElementById("milanList");
     showHide(ul);
}

function selectApostles() {
```

```
    var ul = document.getElementById("apostlesList");
    showHide(ul);
}

function showHide(el) {
    // deselect everything
    var selectedItems = document.querySelectorAll(".show");
    for (var i = 0; i < selectedItems.length; i++) {
        if (selectedItems[i] != el) {
            selectedItems[i].setAttribute("class", "");
        }
    }

    var ulClass = el.getAttribute("class");
    if (ulClass == "show") {
        // item is selected, so deselect it
        el.setAttribute("class", "");
    }
    else {
        // item is not selected, so select it
        el.setAttribute("class", "show");
    }
}

function addImage(e) {
    // we'll fill this in in the next step.
    var a = e.target;
    var imagePath = a.getAttribute("href");

    var image = document.createElement("img");
    image.setAttribute("src", imagePath);

    var div = document.getElementById("image");

    // add image to the div
    div.appendChild(image);
    return false;
}
```

Save it, and open *photo_viewer.html* in a browser. Try clicking on some of the images. Now they should appear in the page. Each time you click, a new image is added to the page! That's not exactly what we want, but it's a good start, and we'll fix it in the next step.

OBSERVE:

```
function addImage(e) {
    var a = e.target;
    var imagePath = a.getAttribute("href");
```

Let's step through how this works. First, notice that our `addImage()` click handler has a parameter, `e`. This is an *event object* that contains information about the click event, including which element was clicked on. This is important for us in this version of the app. Why? Because each image link has the *same* click handler! How can we determine which link was

clicked, so we know which photo to display? That's where the event object comes in. The target property of the event object contains the element that you clicked on to invoke the click handler function. So, we can get that element (which we know is an <a> element) and store it in the a variable.

All event handlers, like click handlers, have a parameter that gets set to the event object. What that event object contains depends on the event you're handling. If you don't need the information, you can just leave off the parameter like we did previously. But if you *do* need the event information, like we do in this example, just include a parameter in your handler definition, and the event object will be passed into it.

Once we have the correct <a> element, we can find out which image to show by getting the value of that element's `href` property. You know how to do this: use the `getAttribute()` method.

OBSERVE:

```
    var image = document.createElement("img");
    image.setAttribute("src", imagePath);
```

Next, we create the image element using `createElement()` as usual. But for an <img> element, we aren't going to see an image unless we set the `src` attribute of the image, right? So we can set the `src` attribute using `setAttribute()` to the image path that we got from the `href` attribute.

OBSERVE:

```
    var div = document.getElementById("image");

    // add image to the div
    div.appendChild(image);
    return false;
}
```

Now that we have a new <img> element that points to the right image, we need to add that <img> element to the DOM. We add it as a child of the "image" , so we get that using `document.getElementById()`, and use `appendChild()` to add the image.

We still need the `return false` at the end; remember, that keeps the browser from following the link to the image. We're in charge of displaying the image now, so we don't want the browser to follow the link.

Removing Elements from the Page

One last thing to add to this application: instead of appending each image you click on to the previous one, we want to see only one image at a time. That means we need to remove the previous image before adding the new image. To remove elements from the DOM, use `removeChild()`. Edit the JavaScript as shown:

CODE TO TYPE:

```
window.onload = init;

function init() {
    var alcatrazSpan = document.getElementById("alcatraz");
    var milanSpan = document.getElementById("milan");
    var apostlesSpan = document.getElementById("apostles");

    alcatrazSpan.onclick = selectAlcatraz;
    milanSpan.onclick = selectMilan;
    apostlesSpan.onclick = selectApostles;

    var links = document.querySelectorAll("a");
    for (var i = 0; i < links.length; i++) {
        links[i].onclick = addImage;
    }
}

function selectAlcatraz() {
    var ul = document.getElementById("alcatrazList");
    showHide(ul);
}

function selectMilan() {
    var ul = document.getElementById("milanList");
    showHide(ul);
}

function selectApostles() {
    var ul = document.getElementById("apostlesList");
    showHide(ul);
}

function showHide(el) {
    // deselect everything
    var selectedItems = document.querySelectorAll(".show");
    for (var i = 0; i < selectedItems.length; i++) {
        if (selectedItems[i] != el) {
            selectedItems[i].setAttribute("class", "");
        }
    }

    var ulClass = el.getAttribute("class");
    if (ulClass == "show") {
        // item is selected, so deselect it
        el.setAttribute("class", "");
```

```
    }
    else {
        // item is not selected, so select it
        el.setAttribute("class", "show");
    }
}

function addImage(e) {
    var a = e.target;
    var imagePath = a.getAttribute("href");

    var image = document.createElement("img");
    image.setAttribute("src", imagePath);

    var div = document.getElementById("image");
    // remove existing children
    while(div.firstChild) {
        div.removeChild(div.firstChild);
    }

    // add image to the div
    div.appendChild(image);
    return false;
}
```

Save it, reload *photo_viewer.html* in your browser. Try clicking on some of the image links. Now you should see just one image at a time.

One of the challenges in removing elements from the DOM is finding the precise element you want to remove, and the parent element you need to remove it from. In this example, the `<img>` element we want to remove doesn't have an `id`. However, we know that the `<img>` element is the first (and only!) child of the "image" `<div>`, so we can find it using `div.firstChild`, which is the first child of the `<div>` element. We know that the `<img>` element is the only child of the `<div>`, so that's good enough for our purposes.

OBSERVE:

```
    // remove existing children
    if (div.firstChild) {
        div.removeChild(div.firstChild);
    }
```

First we check to make sure that the `firstChild` exists (if it's the first time you're clicking on an image link, there will be no `firstChild` of the "image" `<div>` and so that will cause an error if we don't check first!). If there is a `firstChild`, then we can safely remove it from the `<div>` using `removeChild()`.

Once the old image is gone, we add the new one, and voila! The image changes in the web page from the old one to the new one. Isn't playing with the DOM fun.

Text Nodes

You've seen how to use properties like `firstChild` and `parentElement` to access elements in the DOM tree. But you have to be a bit careful when using these properties (and other navigation properties and methods); as you may discover, the DOM tree sometimes contains unexpected children!

We kind of glossed over how text values are stored in the DOM. We showed the text values sitting in the DOM, and we've talked about setting the text content of elements with `innerHTML`, but we haven't talked about how text is actually stored in the DOM.

You might discover by accident that the DOM contains *text nodes* (objects in the DOM are often referred to as "nodes"). For instance, suppose you tried to write a program that would go through all the children of a `<div>` element and set the `backgroundColor` of each *child node* to a grey color (`#acacac`). Let's try that now.

Create a new file and add the HTML and JavaScript below:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Playing with Text Nodes </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
       var div = document.getElementById("article");
       for (var i = 0; i < div.childNodes.length; i++) {
          console.log(div.childNodes[i]);
          div.childNodes[i].style.backgroundColor = "#acacac";
       }
    }
  </script>
</head>
<body>
  <div id="article">
    <p id="p1">
      This is the first paragraph in my brilliant article.
    </p>
    <p id="p2">
      This is the second paragraph. Is my article getting too long?
    </p>
  </div>
</body>
</html>
```

Save it as *textnode.html* in a work folder other than the one used for your photo-viewer app. Now load the page into a browser. You see the two paragraphs in the `<div>` element, but not with a grey background! To see why, open up the console and reload the page in the browser. You see an error message like:

```
"TypeError: 'undefined' is not an object (evaluating
'div.childNodes[i].style.backgroundColor = "#acacac"')"
```

So you're getting an error message and that's why you're not seeing the `backgroundColor`, but what on earth does that error message mean? Let's step back and look at the code:

OBSERVE:

```
<script>
  window.onload = init;

  function init() {
     var div = document.getElementById("article");
     for (var i = 0; i < div.childNodes.length; i++) {
        console.log(div.childNodes[i]);
        div.childNodes[i].style.backgroundColor = "#acacac";
     }
  }
</script>
```

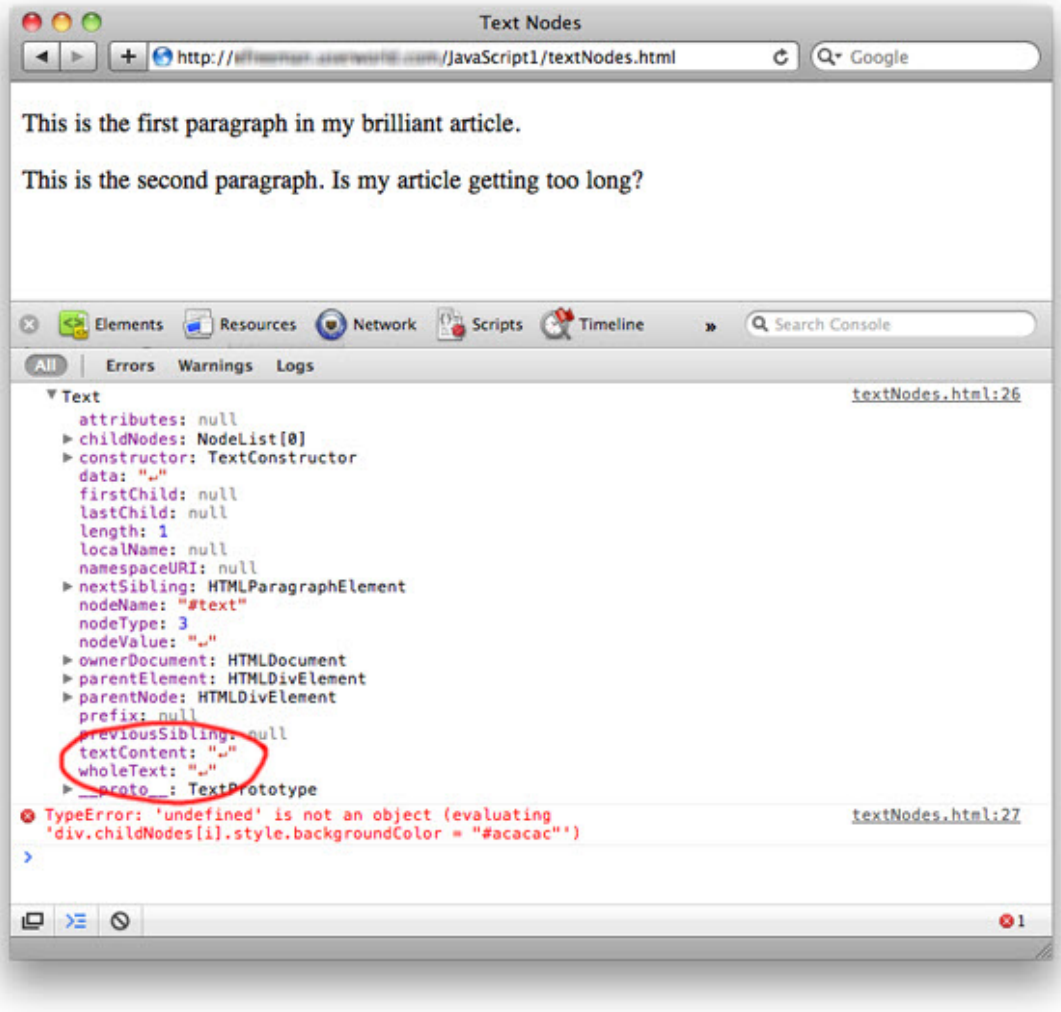
What we're attempting to do is: first, get the "article" `<div>` element, and then loop through all the `<div>` element's children, adding a grey background to each one. Notice that we use the `childNodes` property to do this loop. The `childNodes` property is a collection of all the children of the `<div>`. Now, if you look at the "article" `<div>` in the HTML, you'll note that it contains two `<p>` elements. So inside the for loop, we get each child node of the `<div>` again using the `childNodes` collection, and setting its `style.backgroundColor property` to grey. Okay, no problem; this should work fine, right? Except it doesn't...

You might assume that the *for* loop will find two children of the `<div>`; the two `<p>` elements, right? Look at the console again. Notice that in the code, we are using `console.log()` to display the value of each child of the `<div>`, using `childNode[i]`. Instead of seeing the `<p>` element in the console, however, you will probably see *Text*.

Note *Browsers sometimes differ in how they handle text nodes, so if you aren't seeing a Text node in the console, don't worry. Just follow along to see how Text nodes work. Or try another browser to see if you get different behavior.*

What you're seeing is what's called a *text node*. This is an object in the DOM that contains text. You might be wondering where this came from. After all, we have no text in the element itself; all the text is contained in the two <p> elements that the "article" element contains, right? So where is this Text node coming from?!

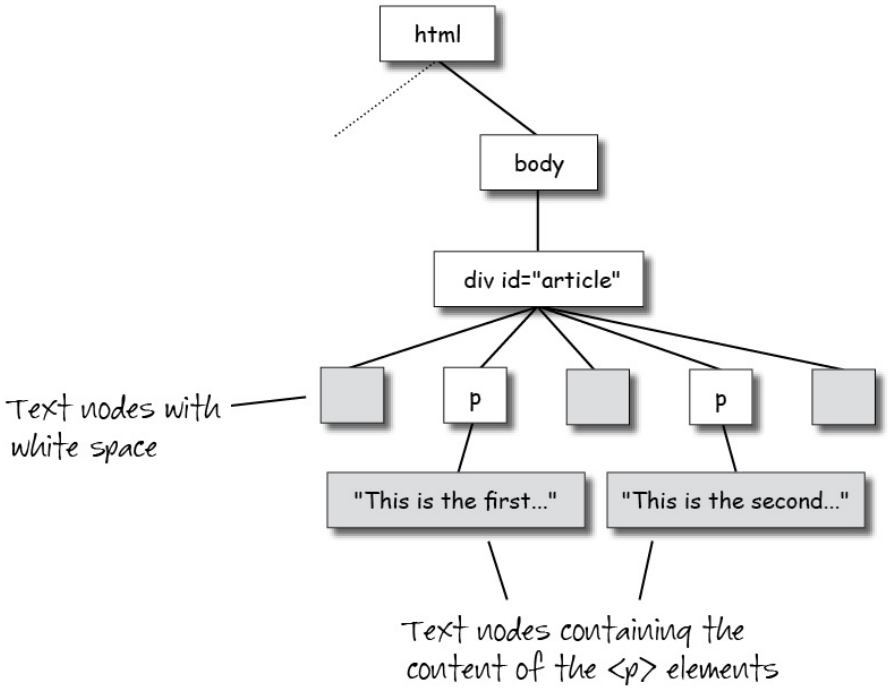
Again, using the console, click on the arrow next to the word "Text" to open up the Text object. Look for a property named "textContent." Do you see a little arrow symbol that looks like a return key? Here's what it looks like in Safari:



It turns out that the DOM stores white space as well as actual text content as text nodes. These text nodes are objects in the DOM just like element objects, and just like text nodes that contain actual content! That's no problem, unless you're expecting to get an element node and you get a text node by mistake and encounter an error like this one. And you're getting the error because you can't set the background color of a text node (or any other style property for that matter); *you can only set style properties on element objects*! In fact, text nodes are quite

different from element objects (or "element nodes"), so you have to be sure you test for them—avoid them altogether (which you'll see how to do in a moment).

Here are the text nodes in the DOM tree for this example:



You can access the content of the <p> elements using text nodes; to do that, just change your JavaScript to access the `childNodes` of one of the <p> elements instead of the :

CODE TO TYPE:

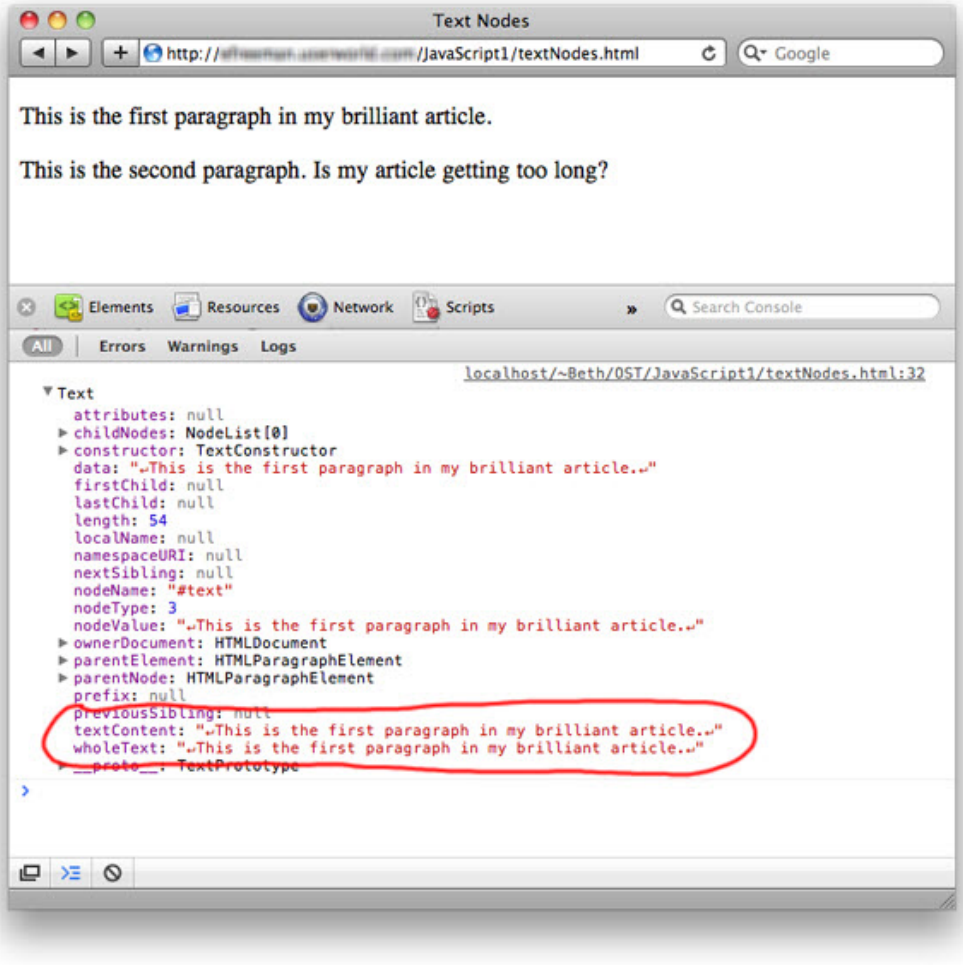
```
<!doctype html>
<html lang="en">
<head>
  <title> Playing with Text Nodes </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var div = document.getElementById("article");
      for (var i = 0; i < div.childNodes.length; i++) {
        console.log(div.childNodes[i]);
        div.childNodes[i].style.backgroundColor = "#acacac";
      }

      var p = document.getElementById("p1");
      for (var i = 0; i < p.childNodes.length; i++) {
```

```
        console.log(p.childNodes[i]);
      }
    }
  </script>
</head>
<body>
  <div id="article">
    <p id="p1">
      This is the first paragraph in my brilliant article.
    </p>
    <p id="p2">
      This is the second paragraph. Is my article getting too long?
    </p>
  </div>
</body>
</html>
```

Save it, and reload the page in your browser. Now, the Text object you see in the console contains real text content. Take a look by again opening up the Text object in the console, and looking for that `textContent` property. Now you should see something more like this:



Check that the value of the `textContent` property you're seeing in the console matches what you wrote as the content of the first `<p>` element in the HTML.

Avoiding Text Nodes Using getElementById() or querySelectorAll()

So what's a solution to our original problem? In other words, if we want to set the `backgroundColor` of all the children of the "article" `<div>` to grey, how do we do it without getting hung up by these pesky text nodes? Easy! You already know a couple of different ways of doing this using `document.getElementById()` and `document.querySelectorAll()`, right? Let's see how you might do this using `document.querySelectorAll()`.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Playing with Text Nodes </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var paras = document.querySelectorAll("div#article p");
      for (var i = 0; i < paras.length; i++) {
        paras[i].style.backgroundColor = "#acacac";
      }
    }
  </script>
</head>
<body>
  <div id="article">
    <p id="p1">
      This is the first paragraph in my brilliant article.
    </p>
    <p id="p2">
      This is the second paragraph. Is my article getting too long?
    </p>
  </div>
</body>
</html>
```

Save it, and reload the page in your browser. Now when you load the page, your paragraphs should have a grey background color, just like you would expect.

We used `document.querySelectorAll("div#article p")` to get the `<p>` elements from the DOM. This ensures that we don't get any text nodes; we only get element objects in the resulting collection, so we can safely set the background color to grey without

getting an error message. The selector "`div#article p`" selects all the `<p>` elements that are children of the `<div>` element with the `id` "article."

Query selectors are a powerful way of selecting DOM objects from your page to manipulate. You'll probably find you use this method a lot!

Note *Make sure you test your code in a variety of browsers. At this point, the only browser versions you still have to worry about when using the query selector methods are IE 6 and IE 7. IE 8+ and all the modern versions of the other major browsers support these methods. If you want to support these older browsers, you'll need to use* `document.getElementById()` or `document.getElementsByTagName()` instead.

Now you know the basics of how to navigate the DOM and manipulate your web pages with JavaScript. With these tools you can create pretty amazing web applications!

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.