Riverside JS Workshop
JavaScript Skills Ladder, Lesson 1.6

The Document Object Model

Lesson Objectives

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

- *use an internal representation of the page called the Document Object Model.*
- *use document to access and update parts of your page.*
- *access and update elements.*
- *use the wiindow object.*

So far in this course, we've been writing JavaScript and seeing the results using (mostly) alert and console.log, or typing JavaScript directly into the console.

However, the real goal of learning JavaScript is to write code that interacts with a web page. To do that, you need to know a bit more about how the browser represents a web page, and understand some of the built-in JavaScript objects, functions, and properties that you'll use to interact with the page.

Behind the Scenes of a Web Page

When you load a web page into your browser, either by clicking a link or by typing in a URL and pressing the Enter key, your browser retrieves the page, and begins to *parse* the page. Parsing just means the browser takes the HTML text and turns it into an *internal representation* of the page. That internal representation is called the *Document Object Model*.

The Document Object Model, often just called the *DOM*, is a hierarchical collection of objects that represent the page and the objects in it. Let's take a look at a simple example:

CODE TO TYPE:

```
<!doctype html>
  <html lang="en">
    <head>
      <title> Simple DOM </title>
      <meta charset="utf-8">
    </head>
    <body>
      <p id="answer">
        The answer to life, the universe and everything is 42!
```

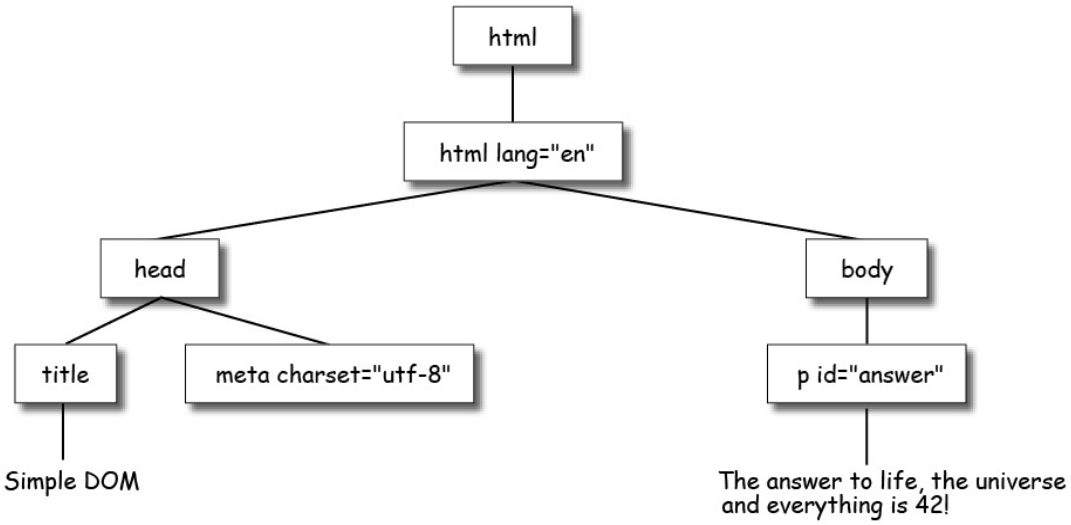
```
      </p>
  </body>
</html>
```

Save it in your work folder as *dom.html*, and open it in a browser. Now, we'll use the browser's developer tools to look at the page "behind the scenes."

Now try it yourself with the example above. If you need to review how to access the developer tools in your browser of choice, you can find answers on Google. Then, add some elements to this simple example, reload the page, and again view the elements with the developer tool.

The Document Object Model and document.getElementById()

The browser represents the DOM elements in your page as hierarchically nested objects, and we usually visualize it as an upside-down tree:



If you turn this image upside down, you'll see the "tree": *html* is the *root* of the tree, and everything in the page "grows" from the root. Notice that the *nesting* of the elements in your page corresponds to the grouping of elements in the hierarchy of the tree. So, for example, the `<title>` and `<meta>` elements are nested inside the `<head>` element in your HTML, so the title and meta objects are grouped beneath the head object in the DOM tree.

Notice that I've included the attributes of the elements in the tree, like the *charset* attribute of the `<meta>` element, and the *id* attribute of the `<p>` element. I've included those because they get included in the internal representation of the page, and you can access those pieces through code too.

Finally, notice that I also included the *text content* of the elements, like the `<title>` and the `<p>`. You can access the content of any element, and update it too.

You can get access to this tree using the *document* object. The document object is a built-in JavaScript object that gives you access to the elements in your page through JavaScript code. You'll learn more about objects later in the course. For now, think of an object as a collection of properties and functions. You can use the document object to access properties of your page, and you can use the document object's functions—called *methods* because they are functions associated with an object, rather than independent functions you wrote yourself—to perform actions on the page.

Let's take a look at some of the ways you can use document to access and update parts of your page.

CODE TO TYPE:

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

    function init() {
      var p = document.getElementById("answer");
      var answer = prompt("Enter your answer to life, the universe and
everything:");
      p.innerHTML = answer;
    }
  </script>
</head>
<body>
  <p id="question">
    What is the answer to life, the universe and everything?
  </p>
  <p id="answer">
        The answer to life, the universe and everything is 42!
  </p>
</body>
</html>
```

Save it, and open it in a browser. Enter an answer when prompted. When you click OK, the answer is added to the web page. Now let's take a closer look.

OBSERVE:

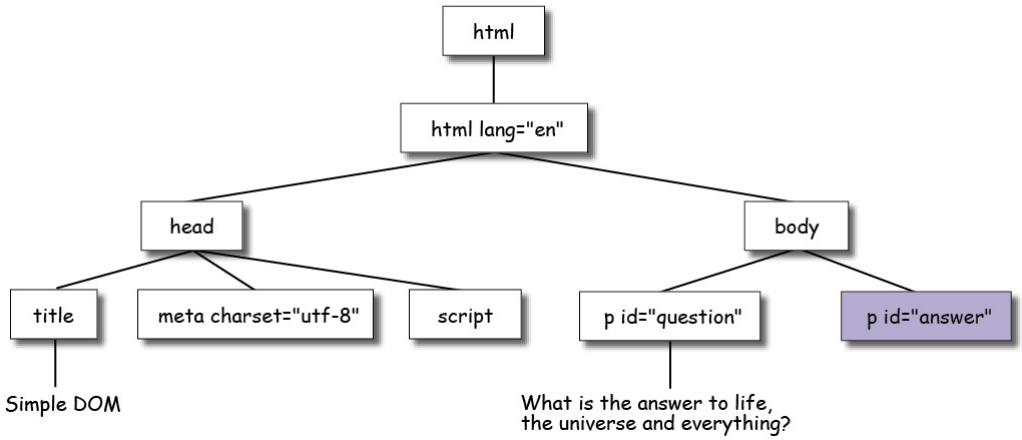
```
  <script>
    window.onload = init;

    function init() {
      var p = document.getElementById("answer");
```

```
      var answer = prompt("Enter your answer to life, the universe and
everything:");
      p.innerHTML = answer;
    }
  </script>
```

There are two steps in this code where your JavaScript interacts directly with the web page: first, where you get the <p> element from the page, and second, where you update the content of the <p> element.

Getting Access to Elements

So what's really going on with `document.getElementById("answer")`? Let's break it down into three parts. We use the document object's method getElementById to get the element with the id "answer". Remember, a method is just a function associated with an object. What does it mean to "get an element"? It means we're getting a handle to the *element object* in the DOM tree that represents that element in the HTML. In this case, we get the object that represents the <p> element with the id "answer." Once we have that element object, we can access its properties and methods.

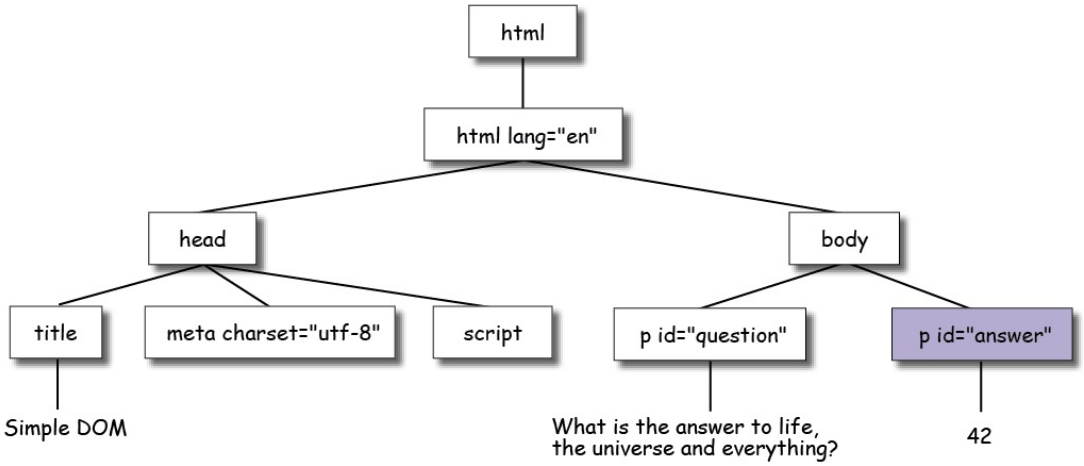


We have a new element object, *script*, grouped under the *head* and we have two *p* element objects in the *body*, each with its own id. By using the id "answer" in `document.getElementById("answer")`, we target one specific *p* object.

Updating Elements

One of the properties of an element object is the `innerHTML` property. This represents all the content of that element. This could include additional HTML elements, or just text, as in this example. We're using this property to *set* the content of the <p> element with the id "answer" by setting `p.innerHTML` to the content of the variable `answer`, which contains the value you

typed in at the prompt. If you typed "42," the DOM tree looks like this after this JavaScript code runs:



This is really cool, because you are updating your web page dynamically using JavaScript! The web page will magically change to include the new content when this code runs.

So, `document.getElementById()` is a common way you'll get access to elements in your page using JavaScript, and the `innerHTML` property is a common way you'll update the content of elements. These are the most useful tools in your toolbox for using JavaScript to interact with your page, but there are many more.

Getting Elements with document.getElementsByTagName()

With `document.getElementById()`, you can retrieve only one element at a time from the DOM. Why? Because, remember that in HTML, ids must be unique, so only one element can have a given id. This is a good thing because we want to be able to target elements uniquely, using ids with both CSS and JavaScript.

However, there are many times when you want to access multiple elements at once. In this case, it's more efficient to access elements another way, for example, using the elements' tag name instead. Let's take a look at an example. Create a new file as shown:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Accessing multiple elements </title>
  <meta charset="utf-8">
```

```
  <script>

  </script>
</head>
<body>
  <form>
    <p>Enter numbers to add:</p>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> <br>
    <p>= <span id="sum"></span></p>
    <input type="button" id="submit" value="Add">
  </form>
</body>
</html>
```

Save it in your work folder as *dommulti.html*, and open it in a browser. There's no script yet to process the input, so the *Add* button doesn't do anything. Let's step through the HTML for the input form:

OBSERVE:

```
  <form>
    <p>Enter numbers to add:</p>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> <br>
    <p>= <span id="sum"></span></p>
    <input type="button" id="submit" value="Add">
  </form>
```

We have a form with three "number" `<input>` elements, and a submit button `<input>` element. When the user clicks the Add button, we want to add up all the numbers entered in the number inputs, and put the resulting sum in the `<span>` element with the id "sum".

Open the JavaScript console in your browser in the window where you've previewed the HTML above, and enter the command as shown:

TYPE IN THE CONSOLE:

```
document.getElementsByTagName("input");
```

You'll see the four `<input>` elements in the page: the three number inputs, and the button input. It looks something like this:

So, `document.getElementsByTagName()` returns a collection of HTML elements—all the elements that match the id you specify in the call to `document.getElementsByTagName()`. What if you want to use `document.getElementsByTagName()` in a program? And once you've got the collection of elements, what can you do with them? Let's find out. We'll write a JavaScript program to take the values the user types into the form inputs, add them up, and put the result in the page in the "sum" `<span>` element.

Setting Up the window.onload and button.onlick Events

You might remember from an earlier lesson when we said that you can't access values in the DOM—either to read or update—before the page has completely loaded. That means we can't call `document.getElementsByTagName()` until the page has loaded. So we're going to need to set up a function to call when the page has loaded, by setting the `window.onload` property, like we did in the previous example above.

But we don't want to run the JavaScript code to add up the numbers when the *page* loads. We want to add them up only when you click the Add button. Just like we can delay executing code until after the page has loaded with the `window.onload` property, we can also delay executing code until the user clicks a button with the `button.onclick` property.

So, how do you tell JavaScript to run code when a button is clicked? Let's take a look:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Accessing multiple elements </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var button = document.getElementById("submit");
      button.onclick = addUp;
    }

    function addUp() {
```

```
      alert("testing sum");
    }
  </script>
</head>
<body>
  <form>
    <p>Enter numbers to add:</p>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> <br>
    <p>= <span id="sum"></span></p>
    <input type="button" id="submit" value="Add">
</form>
</body>
</html>
```

Save it, and open it in a browser. Try clicking the *Add* button. Do you see the alert?

Now, again, don't worry too much if you don't understand all this code yet. You'll learn more about functions and events shortly. Get the basic idea for how this works, and then focus on how we used JavaScript to get and update elements (the next step).

OBSERVE:

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

    function init() {
      var button = document.getElementById("submit");
      button.onclick = addUp;
    }

    function addUp() {
      alert("testing sum");
    }
  </script>
```

First we set up an `init()` function that will execute when the page finishes loading. By setting the `window.onload` property to `init`, we're telling JavaScript, "Run the function init when you finish loading the page."

The init function gets the button element from the DOM using `document.getElementById()` and its id "submit" (look in the form HTML for the button, and you'll see it has that id).

Now, just like we can set the `window.onload` property to a function to tell it to run that function when the page is loaded, we can assign a function to a button element's *onclick property* to tell it to run that function when the button is clicked. So by setting button.onclick,

we're telling JavaScript to call the `addUp()` function when the Add button is clicked. This is called "event handling" and we'll get into a lot more detail in the next lesson!

In the `addUp()` function, so far, we merely added an alert so we can see the button is working. Now we need to actually add up the numbers!

Get the Form Input Values and Update the Page

The next step is to write the code for the `addUp()` function. This function will get all the values typed into the form inputs, add them up, and put the result in the "sum" `<span>`.

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> Accessing multiple elements </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      var button = document.getElementById("submit");
      button.onclick = addUp;
    }

    function addUp() {
      alert("testing sum");
      var sum = 0;
      var inputs = document.getElementsByTagName("input");
      for (var i = 0; i < inputs.length - 1; i++) {
        var addendString = inputs[i].value;
        var addend = parseInt(addendString);
        if (!isNaN(addend)) {
          sum += addend;
        }
      }
      var span = document.getElementById("sum");
      span.innerHTML = sum;
    }
</script>
</head>
<body>
  <form>
    <p>Enter numbers to add:</p>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> + <br>
    <input type="number" size="4"> <br>
    <p>= <span id="sum"></span></p>
    <input type="button" id="submit" value="Add">
  </form>
</body>
```

```
</html>
```

Save it, and open it in a browser. Enter some numbers and click **Add**. What happens if you leave one of the form inputs blank? What if you enter a string instead?

Note Some browsers won't let you type a string into a "number" input; others will.

OBSERVE:

```
function addUp() {
  var sum = 0;
  var inputs = document.getElementsByTagName("input");
  for (var i = 0; i < inputs.length - 1; i++) {
    var addendString = inputs[i].value;
    var addend = parseInt(addendString);
    if (!isNaN(addend)) {
      sum += addend;
    }
  }
  var span = document.getElementById("sum");
  span.innerHTML = sum;
```

Let's step through this code. First, we initialize the variable `sum` to 0; this is where we'll keep a running total of the numbers in the form.

Next, we use `document.getElementsByTagName("input")` to get all the elements with the tag name "input." This returns four input element objects in an *HTML collection*, as you saw earlier when you used the console. We store that result in the variable inputs.

An HTMLCollection is similar to an array, and you can use a *for loop* to access each element in the collection just like you use a for loop to access each item in an array.

Note A collection is *not equivalent* to an array, so you can't do everything with the collection that you can with an array. But the syntax for looping through the items in the collection is exactly the same as for looping through an array.

So, we loop through the items in the inputs collection so we can get each number from the "number" inputs. But remember, there's one `<input>` element in the collection that's *not* a number: the "button" `<input>` element! So instead of looping four times, we only want to loop three times, so our looping test uses `inputs.length - 1` instead of the usual `inputs.length`. Of course, this only works because know that the "button" input will be the

last item in the array. If that wasn't the case, we'd have to check each item in the collection to make sure it was a "number" input and not a "button" input.

Each item in the collection—that is, each inputs[i]—is an input element object. We can get the value of what the user typed into a given input element by using its value property. So, for instance, inputs[i].value will contain the string "3" if you typed in 3.

Notice that the value we get from the input is a string, not a number! We can convert the string to an integer number with the function parseInt(). But what if you type in "x" or "cheese" instead of a number? Then that number isn't going to get converted to a number correctly, right? If the string you pass to the `parseInt` function doesn't represent an actual number, then the result of the function is `NaN`, which means "Not a Number." So, the variable addend will contain `NaN` and if we add that to sum, it will mess everything up (that is, it will cause the entire sum to be NaN!).

So, we need to check to see if the value in the addend variable is `NaN`. To do this, we use the `isNaN()` function. If addend is *not* equal to NaN, then we can add it to sum. Otherwise, we just ignore it.

Finally, we get the "sum" `<span>` element using `document.getElementById()`, and set its `innerHTML` property to the value in sum. This causes the page to update, and the number appears in the page!

That was a lot of new concepts to get through, so spend some time going over this code again, and make sure you understand each part. Can you draw the DOM tree for the HTML in this example? Try drawing the DOM tree for both before and after you click the Add button.

Experiment a little. What happens if you enter a negative number, like -3, in the form? Do you get the correct sum? What about if you enter a floating point number, like 3.2, into the form? Do you get the correct sum? If not, why not? What if you change `parseInt` to `parseFloat`?

The Window Object

So now you know that the *document* object is an important object because it's how you get JavaScript to talk to your web page, access elements and content in the page, and update your page.

You've also been using another important object: the *window* object. You used it when you set the `window.onload` property to delay executing a JavaScript function until after the page loads.

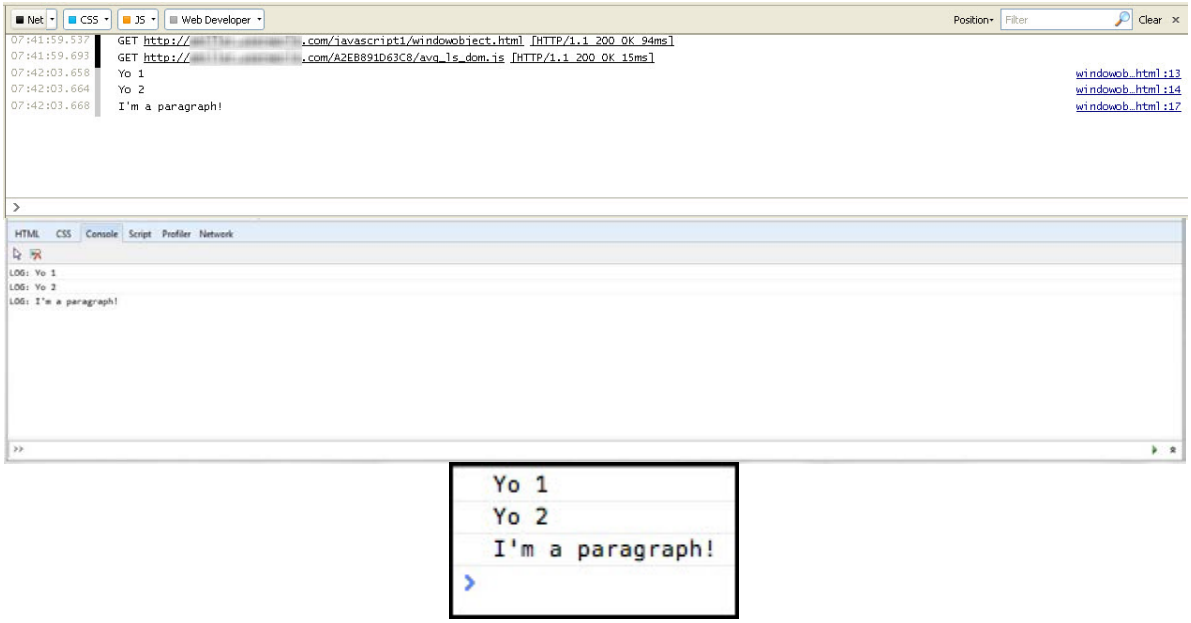
That window is the default object in any JavaScript program. What does that mean? It means that anything you do at the "top level"—like using document (for example, with `document.getElementById`) or a function like `alert()`—you're actually doing in the window object!

So instead of `alert("hey");` you could write `window.alert("hey");`. Or instead of `console.log("hey");`, you could write `window.console.log("hey");`. The same is true with document: instead of `document.getElementsByTagName("input");`, you could write `window.document.getElementsByTagName("input");`. You can try it yourself in the JavaScript console window, or by creating an HTML file like this:

CODE TO TYPE:

```
<!doctype html>
<html lang="en">
<head>
  <title> The Window Object </title>
  <meta charset="utf-8">
  <script>
    window.onload = init;

    function init() {
      alert("hey 1");
      window.alert("hey 2");

      console.log("Yo 1");
      window.console.log("Yo 2");

      var p = window.document.getElementById("theP");
      window.console.log(p.innerHTML);
    }
</script>
</head>
<body>
  <p id="theP">I'm a paragraph!</p>
</body>
</html>
```

Save it in your work folder as *windowobject.html*, and open it in a browser. Do you see two alerts? Check the console. Do you see three messages in the console, like this:




```
Yo 1
Yo 2
I'm a paragraph!
>
```

You get the same results whether you prefix these functions with *window* or not.

So, why don't you have to type `window.alert("hey");` instead of `alert("hey");` every time? Because window is the *default* object, JavaScript *assumes* window if you don't type it. That then begs the question, why do we write `window.onload = init;`, when we could just write `onload = init;`? The reason is that while there's usually only one `alert` function, there could be many `onload` properties being set in your code. So it's best to be absolutely clear, and specify `window.onload` so there's no ambiguity about what you mean. This will create fewer bugs and make your code easier to read.

We'll come back to the window object again later. For now, it's just important that you know it's another important object in JavaScript, like document, and that it is the "default object" that has properties and methods you'll use often when writing JavaScript programs.

Another action-packed lesson! In this lesson you've learned:

- The browser represents pages internally using the Document Object Model, which we often visualize as an upside-down tree.
- The Document Object Model (DOM) contains all the elements and content of your page as objects.
- You can access these objects using `document.getElementById()` and `document.getElementsByTagName()`.
- `document.getElementById()` returns one element, the element with the id you specify.

- `document.getElementsByTagName()` returns a collection of elements that match the tag name you specify. Even if only one (or no) elements match, you'll get an array back.
- You can loop through the collection of elements returned by `document.getElementsByTagName()` just like you would loop through an array of items.
- It's important not to access the objects in the DOM until the page has completely loaded.
- The window object is another important object you'll use in your JavaScript programs.
- The window object is the "default object" for your JavaScript programs.

Spend some time practicing with `document.getElementById()` and `document.getElementsByTagName()` before you do the project and go on the next lesson.

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.