4

Accessing the DOM

In this chapter, we start putting into practice some of the concepts you learned about in previous chapters and begin coding some basic JavaScript. We get into the different ways to manipulate elements within a page, access HTML nodes, and learn how to move up and down the document structure. This is the basis for all JavaScript interaction; as you move forward in your JavaScript life cycle, these are skills you will be using on most days (the good ones, at least).

What Is the DOM?

When you hear people talking about JavaScript, they often mention the "DOM," or "Document Object Model." This is one of the most common and important aspects of JavaScript that you'll encounter. You will use it every time you modify a page and every time you insert data or modify an interface. In a nutshell, the DOM is a mapping layout model for your HTML and a way for JavaScript to get in there and do its thing. The DOM can be a JavaScript developer's best friend after it is mastered.

The DOM is not HTML and not JavaScript, but they are all very interconnected. It contains some very basic concepts, and as you lift up the hood to see all the possibilities, you'll notice that it's much more than a mass of HTML and text. It is the combinations of the elements you use (object), the order you choose to display them (model), and an endless offering of access points for you to add, delete, or modify any part of the overall structure (document) while navigating around with the various methods JavaScript provides.

Step one, and something that many beginners get stuck on conceptually, is that the HTML file is a document when we talk about the Document Object Model. Your document is made up of a bunch of objects (HTML elements, text, and attributes). Those objects come together in a certain order, which is considered the "model."

JavaScript is all about creating behaviors and data interactions by accessing the DOM. To create and control truly interactive experiences through interface design, you will certainly have to master and grow accustom to working in this robust environment.

The DOM Tree

Every document has some kind of structure behind it, whether it's a chapter of this book, a flier for a yard sale, a newspaper, or an HTML document. In this case, we'll obviously be talking about an HTML document, but the principle is true for most things we create.

The DOM starts with the Web browser. When you visit a website, the browser renders the HTML into a document, reads through the elements, and displays them to the user in the correct format. This process creates the initial model for the document, and it gets stored so we don't have to do all that work again to access each individual element (we just reference what the browser already has stored for the most part).

Consider the HTML structure shown in Listing 4.1.

Listing 4.1 **Basic HTML Structure to Illustrate the DOM**

```
<!DOCTYPE html>
<html>
<head>
       <meta charset= "utf-8 ">
       <title>Basic DOM example</title>
</head>
<body>
       <h1>Hello World!</h1>
       <p>While this is a <strong>very basic HTML document</strong>, it actually
serves as a detailed example of the document object model.</p>
</body>
</html>
```

At first glance this may look like an extremely basic example of an HTML document, and it is, but it's also a pretty good representation of all the items we have to deal with in the DOM.

The DOM is made up of items called **nodes**, which can have parents, children, and siblings (just like a family), and this determines its position in the tree. Children are underneath parents and siblings are at the same level as other siblings (brothers and sisters, if you will). Let's take a look at the DOM translation of this HTML document in Figure 4.1.

In Figure 4.1, you can see how we create a parent-child relationship in the DOM by moving from the top of the document all the way to the bottom as the nodes begin to create a pseudo-hierarchy in the diagram. I say "pseudo" because as we approach the bottom of the document the branches start to break off into different kinds of nodes that are represented by the circles that may technically be at the same level as others but are not considered to be relevant in the parent-child relationship to each other.

As you may have guessed, there are indeed different types of things we call nodes in the DOM. Nodes that represent HTML elements are called **element nodes**, ones that represent text are called **text nodes** and last are **attribute nodes**, which represent, you guessed it, attributes. All nodes are also children of the overall parent **document node**, represented at the top of Figure

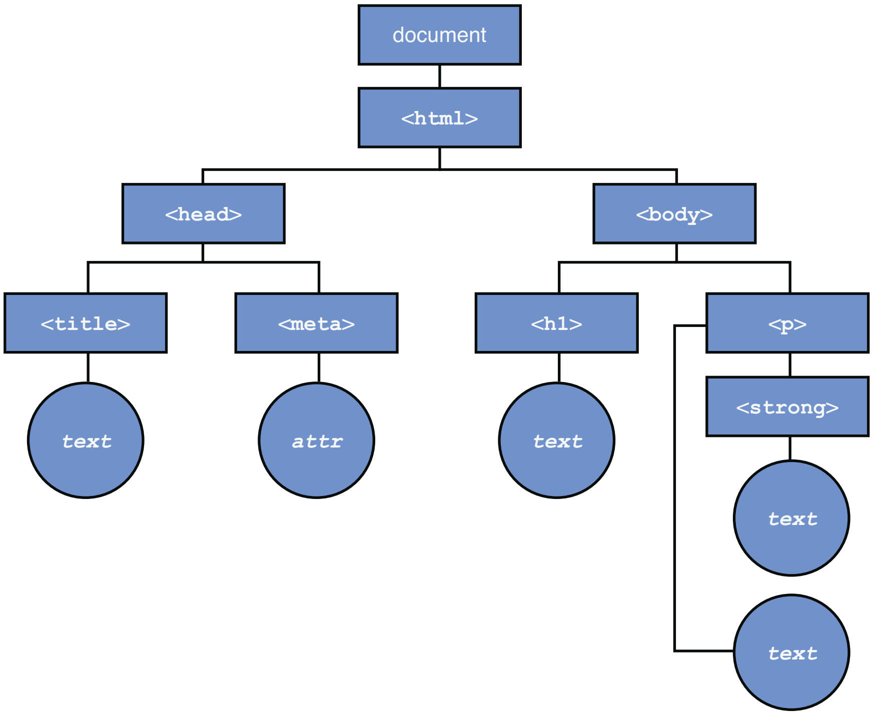

Figure 4.1 A graphical representation of the Document Object Model (DOM)

4.1. It's important to know the difference between these types of nodes because we access them differently when traveling throughout the DOM with JavaScript.

Note

The document node is extremely important and is often overlooked when building out the DOM's graphical representation. As you will learn later, whenever we want to access a node in the document, we first have to pass through the document node before we can step down the tree and start any sort of interaction.

Element Nodes

Element nodes will occupy the majority of your document and are the basis for how you will move around (we'll get to that in a bit). These nodes will define your structure and hold most of the data and content you will want to interact with and modify. The element nodes occupy most of the document and create the tree-like structure you see in Figure 4.1. Let's take a look at our element nodes highlighted in Figure 4.2.

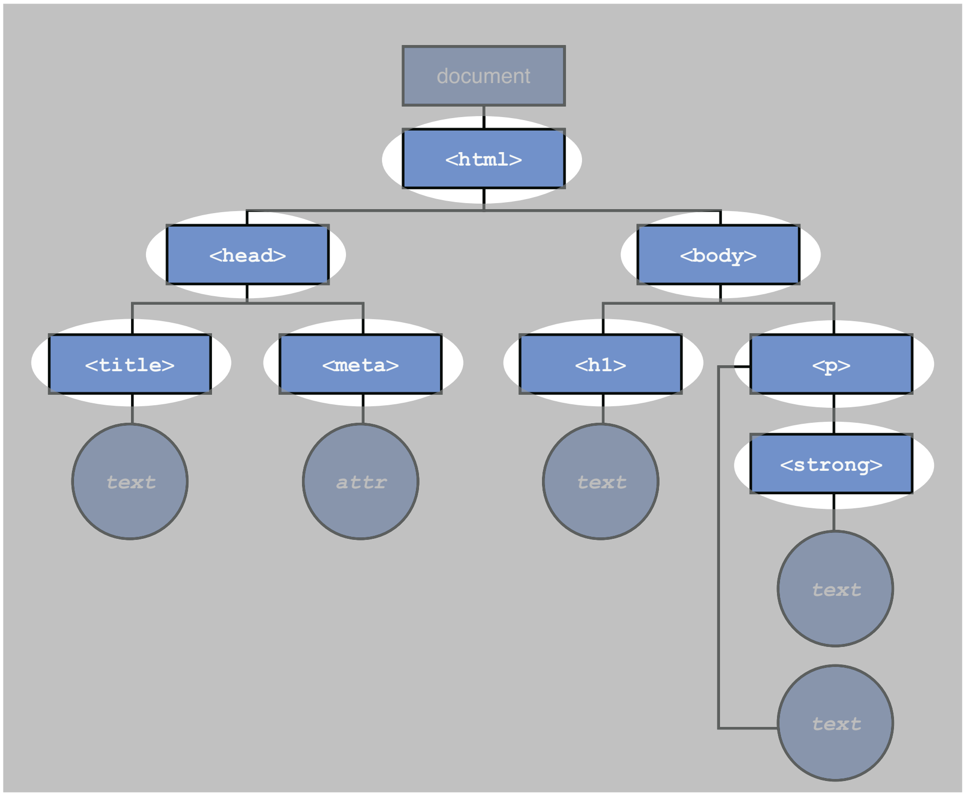

Figure 4.2 Highlighting the element nodes from our DOM. As you can see, this is the majority of our document.

Text Nodes

Text nodes are similar to the element nodes because of how they sit in the DOM and the fact that we use the same JavaScript methods to access them. Of course, some differences exist between these two types of nodes, or we wouldn't need to learn more terminology.

The most notable difference is the way they look; element nodes are contained in angled brackets (greater than and less than symbols). Text nodes are not; they are between element nodes. In Figure 4.3, you can see this depicted graphically.

One other important distinction between the two node types is that text nodes cannot have children (they've been trying for years, no luck). You can see in Figure 4.1 that the `<p>` element node has a text node and another element node `<strong>` has children. However, although the element node `<strong>` is technically inside the paragraph's content in the HTML structure, it is not a child of the text node. This is where we start getting into the pseudo-hierarchy mentioned earlier. Instead of nesting the tree further, we break off into a whole new branch and dead end at the text node. With that in mind, let's take a look at our text nodes highlighted in Figure 4.4.

$$\underbrace{\texttt{<p>}\overbrace{\texttt{Hello World!}}^{\texttt{text node}}\texttt{</p>}}_{\texttt{element node}}$$

Figure 4.3 Showing the difference between an element node and a text node

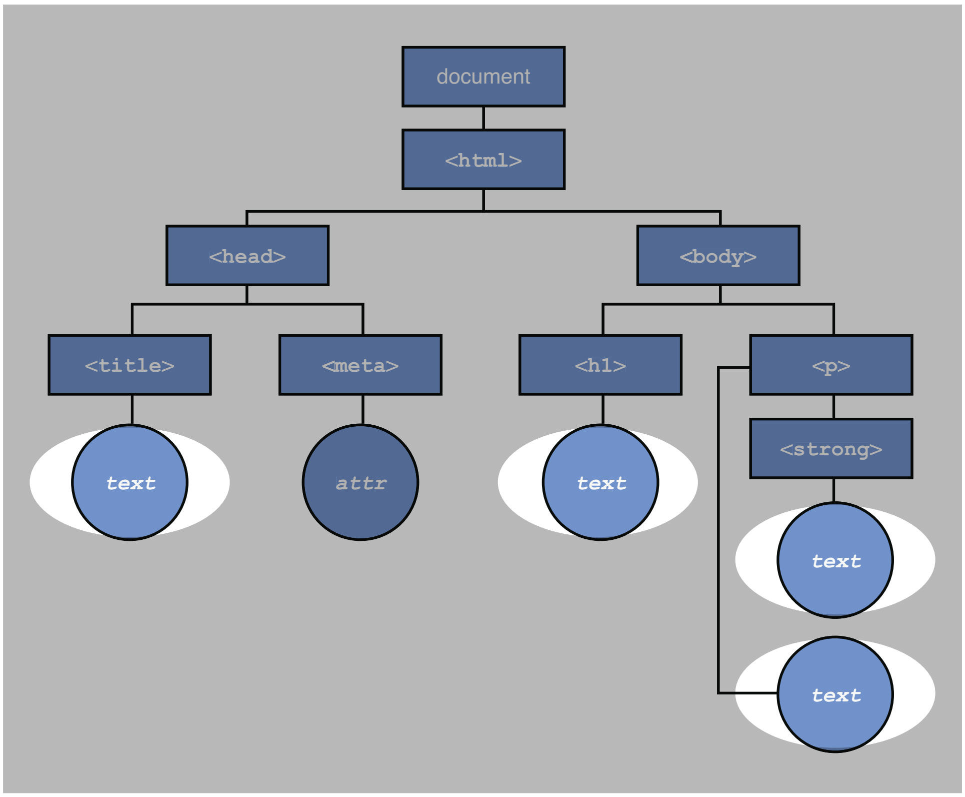

Figure 4.4 Highlighting the text nodes in our DOM. As you can see, the branches do not continue past our text nodes.

Attribute Nodes

Attribute nodes, like the document node, are easy to overlook when building your DOM because they appear to be part of the element, but they're an entirely different type of node, and are extremely important. They're also treated differently than text and element nodes are. In Figure 4.5, you can see the attribute node called out separately from the element node.

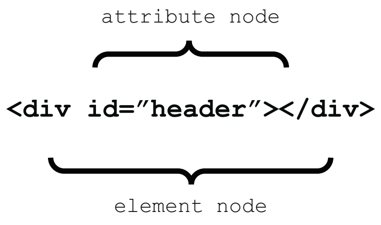

Figure 4.5 Illustrating the position of an attribute node in relation to an element node

Like text nodes, attribute nodes cannot have children (it's a sad story), but they're also not children themselves. That's right, attribute nodes are not considered to be children of their containing element; it's a little odd, but stick with me here. They sit in the DOM structure under element nodes because they're always attached to element nodes, but they are not children—they are treated and accessed differently. (Think Paul Newman in *Cool Hand Luke*.) They both do their own thing against the norm and refuse to conform. This is the reason attribute nodes have their own methods in JavaScript. We use basic JavaScript methods to get near an attribute, but after we get there we have to use special methods to get inside. This may seem a little confusing, but when you get used to it, it's actually pretty nice to have them separate.

When writing efficient JavaScript (don't worry, you will be by the time we're done here), you'll be constantly adding and removing classes from element nodes, and this is where we'll be heavily using attribute nodes. Let's take a look at our last type of node, the attribute node, highlighted in the DOM model in Figure 4.6.

Working with the Element Node

Now that we have a solid understanding of what exactly the DOM is and how parent-child relationships are built, we can start accessing it. Every node we talked about can be reached in some way. Some are easier than others, but I assure you, if something is listed in the DOM, we can grab it and store or modify it with JavaScript.

As mentioned in previous chapters, because of browser rendering weirdness, performance in JavaScript is a constant issue, especially when moving through the DOM. You always want to minimize the amount of nodes that need to be evaluated in order to get where you need to be in the document. Knowing the different ways to jump into the DOM and finding the most efficient access points may seem like a trivial thing, but every millisecond counts when we're talking about creating a good user experience, so we want to do this as fast as possible.

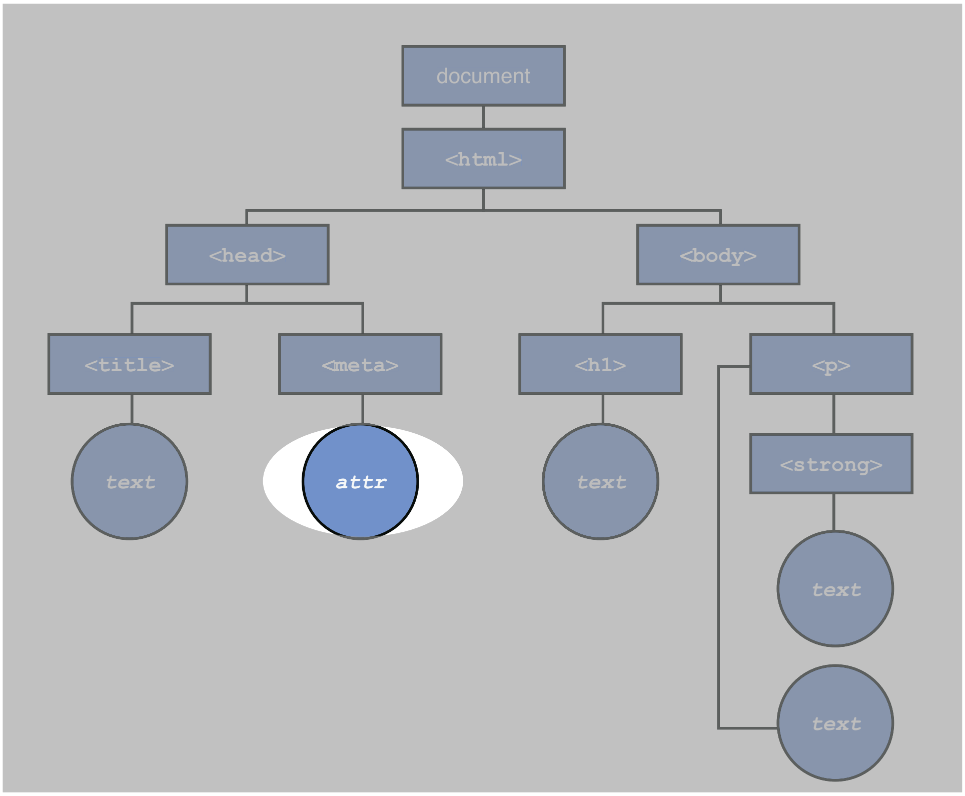

Figure 4.6 Highlighting the only attribute node in our DOM example

Targeting by ID

HTML standards state that an element's ID value must be unique to the document. In other words, there can be only one ID per page. Because of that, using an element's ID as the document access point is by far the most efficient and most desirable. We use this method whenever we can, but let's take a look at just how we would do that. Consider the following HTML snippet in Listing 4.2.

Listing 4.2 **HTML Element Labeled with an ID for Quick Access**

```
<div id="header">
       <h1><a href="/" id="homelink">Title of your site</a></h1>
</div>
```

This is a simple HTML snippet consisting of two ID values that we can use to access each element. Most times you will have natural IDs on your HTML elements (like `"header"` in the example) that you can use for targeting, but sometimes you do need to add in extra (or unnatural) IDs (like `"homelink"` in the example) to gain quick access to an element node. There are other ways, but performance-wise, this is the fastest. In Listing 4.2.1 let's take a look at the JavaScript you would use to access these nodes.

Listing 4.2.1 **JavaScript Used to Access an Element by Its ID**

```
document.getElementById("header");
document.getElementById("homelink");
```

As I mentioned earlier, you have to access the document object before you can get to the node you want. After you have access to the node, you can store, modify, or retrieve extra information about the element with the plethora of methods built into JavaScript that we will be exploring throughout this entire book. It's important to remember that, although we are using the ID attribute to access the node, we are still accessing the *element node* and not the *attribute node* at this point. If we were to access the content of the ID attribute (the text "`header`" and "`homelink`"), we would be accessing the attribute node, but since we're not doing that, we're still on the element node.

Support

Support for the `getElementById()` method is widespread throughout all browsers, because it is one of the older and most well-used methods we have. In fact, you might even call it "retro." We all know that retro is hot right now, so use it well!

Note

Using IDs throughout your HTML is not only great for JavaScript performance but also CSS performance and specificity when applying styles. IDs can also be used for accessibly enhancements when creating jump or "skip to content" links for screen readers or visually impaired users.

Targeting by Tag Name

Because IDs are unique to each document, `getElementById()` can return only a single element. Sometimes that just doesn't fly, and we need more than one, or even a group of elements. When this is the case, we use `getElementsByTagName()` in a similar way.

Note

A pretty common (and frustrating) mistake in JavaScript syntax is not pluralizing the word "elements" in `getElementsByTagName()`. Pay close attention to this and you'll save yourself some debugging time.

Targeting an element (node) by tag name is another popular method of creating an access point in the DOM. You can both target a group of elements and a single element inside of a group if you want. Suppose we have a pretty stripped-down Web page like the one shown in Listing 4.3.

Listing 4.3 **HTML Structure with Mixed and Repeating Element Nodes**

```
<!doctype html>
<html>
<head>
       <meta charset="utf-8">
       <title>getElementsByTagname Example</title>
</head>
<body>
       <h1>Using getElementsByTagname</h1>

       <p>Content paragraph</p>
       <p>Content paragraph</p>
       <p>Content paragraph</p>

</body>
</html>
```

We can grab each paragraph in Listing 4.3 with the single line of JavaScript in Listing 4.3.1.

Listing 4.3.1 **JavaScript Example of `getElementsByTagName`**

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

This will return a DOM object called a **NodeList**. A NodeList is a collection of nodes that maintain their source order. In other words, it's a list of elements, and those elements stay in the same order they appear on the page. The order is pretty important because if the items were randomly shuffled, they would be very difficult to access when we need them.

After we have the NodeList, we will also want to check to see how long it is. Checking its length will allow us to make sure values are available before we execute a block of script. We can do this with the `length` method, depicted in Listing 4.3.2.

Listing 4.3.2 **Returning the Total Number of Elements in Our NodeList**

```
document.getElementsByTagName("p").length;
// will return 3

// checking the length before executing script
if(document.getElementsByTagName("p").length > 0) {
       // it returns "3", and 3 is greater than "0", so we're good to go.
}
```

So far we have targeted a group of elements and checked their length; after that, we may want to grab an individual node to do some work on. It's pretty easy to pull an item out of the NodeList. There are two ways we can do this:

- Using the item method
- Using an array syntax

Both methods will return the same value, so it's more an issue of personal preference when you choose the way you want to access elements in a NodeList. Let's take a look at them both in Listing 4.3.3.

Listing 4.3.3 **Using `getElementsByTagName` to Target a Single Node**

```
// the first paragraph using the item method
document.getElementsByTagName("p").item(0);

// the first paragraph with array syntax
document.getElementsByTagName("p")[0];

// the second paragraph with the item method
document.getElementsByTagName("p").item(1);

// the second paragraph with array syntax
document.getElementsByTagName("p")[1];
```

Note

NodeLists and arrays start at 0, not 1. So the list looks like 0,1,2,3,4, etc. This is why we are using 0 in the preceding example to access the first element. If only a single element returns in the NodeList, you still have to use the [0] value to grab it.

When we put all this code together, you can see a functional JavaScript snippet starting to get created. So far we've targeted a NodeList, checked its length, and accessed the individual element inside the NodeList. This is a very common thing to do in JavaScript, so let's see what it looks like all put together in Listing 4.3.4.

Listing 4.3.4 **Putting It All Together**

```
// checking the length before executing script
if(document.getElementsByTagName("p").length > 0) {
       // it returns "3", and 3 is greater than "0", so we're good to go.

       // let's access some elements with the array syntax
       document.getElementsByTagName("p")[0];
```

```
        document.getElementsByTagName("p")[2];

}
```

Support

Support for the `getElementsByTagname()` method is widespread enough that we can feel free to use this as much as we want without worrying.

Targeting by Class

So far the node targeting methods we've talked about have been similar to CSS selectors. In CSS you can target by both `ID` and `TagName`, and they're very commonly used. You can also target by class. JavaScript is no different; however, it has lagged behind the curve a bit.

For years now, since targeting an element by its class was active in CSS, the community wanted the feature added to JavaScript. It took a while, but we finally have it! Recently added to JavaScript was the `getElementsByClassName()` method. This method functions much the way `getElementsByTagName()` does when it comes to producing and accessing the NodeList, but we can use some interesting combinations of selectors with this method, so let's get started with the HTML in Listing 4.4.

Listing 4.4 **HTML Snippet Illustrating Element Node with Classes**

```
<!doctype html>
<html>
<head>
       <meta charset="utf-8 ">
       <title>getElementsByTagname Example</title>
</head>
<body>
       <h1>Using getElementsByTagname</h1>

       <p class="dropcap huge">Content paragraph</p>
       <p id="e" class="dropcap">Content <span class="huge">paragraph</span></p>
       <p class="dropcap">Content paragraph</p>

</body>
</html>
```

Again, we will be working with the paragraphs in the HTML. Note the classes we're using. The first element has two classes, the second has one class on the paragraph and one on the child span node, and the last element has one class. With the JavaScript in Listing 4.4.1, we can target those nodes individually.

Listing 4.4.1 **Using `getElementsByClassName`**

```
// returns all elements with a class of dropcap
document.getElementsByClassName("dropcap");

// returns all elements with both "dropcap" and "huge" classes (1 in our example)
document.getElementsByClassName("dropcap huge");

// returns all elements classed with "huge" that are inside elements ID'd with "e" (1
element in our example – the span)
document.getElementsById("e").getElementsByClassName("huge");
```

We have a little more flexibility with this than we do with some of the others, because it's much newer to JavaScript. But as you can see, this can be a very helpful method.

Support

As I mentioned, this is a pretty new addition to the JavaScript family. This is where things start to get a little dicey; I will say that support is very good right now, but not at 100%. At the time of this writing, all current browser versions support this method. When we start to step back in time a little, we notice where support begins to weaken. The browsers you'll be concerned with are most likely IE 8 and earlier, but let's take a look at the support table, Table 4.1.

Table 4.1 **Browser Support for `getElementsByClassName`**

Internet Explorer	9.0 & up
Firefox	3.0 & up
Chrome	4.0 & up
Opera	9.5 & up
Safari	3.1 & up

Because this method was so highly sought after for so long, other developers in the community have created custom functions to handle the `getElementsByClassName` method in the case of no support. Every JavaScript library/framework available right now has also addressed this issue, but in different ways. Later on in the book we will get into extending JavaScript through libraries. In cases like this where support is lacking in older browsers, a library or custom function fallback is a good way to ensure that all users will get a similar experience, regardless of how they view your site/application.

Using CSS Selectors in JavaScript to Target Nodes

The desire to use easy-to-remember CSS selectors in JavaScript came to fruition in 2009 when JavaScript guru John Resig developed the Sizzle Selector Library. Sizzle allowed us to maintain a consistent syntax in selectors between CSS and JavaScript. This caused an explosion in the JavaScript community, and all of a sudden this previously difficult-to-use language for designers became a delight to work with.

With the release of the HTML5 specification came a new flood of JavaScript APIs as well. One of the main themes with HTML5 was listening to what the community wanted to be formalized into the language. It was clear that they wanted to use CSS selectors in JavaScript, so out came two new methods: `querySelector()` and `querySelectorAll`().

Both methods can take selectors in CSS format. The difference between the two is that `querySelectorAll()` will return a NodeList (the same way `getElementsByClassName()` and `getElementsByTagname()` do) and `querySelector()` will return only the first element that it comes across. Let's look at the following HTML in Listing 4.5.

Listing 4.5 **Basic HTML Structure with Elements, IDs, and Classes**

```
<body>
    <div id="header  >
        <h1>Using querySelector</h1>
    </div>

    <p class="dropcap huge">Content paragraph</p>
    <p class="dropcap">Content <span class="huge">paragraph</span></p>
    <p class="dropcap">Content paragraph</p>

</body>
```

We can target those elements with the following JavaScript, as shown in Listing 4.5.1.

Listing 4.5.1 **Using `querySelectors`**

```
// get the header ID element
document.querySelector("#header");

// get the first element with a dropcap class
document.querySelector(".dropcap");

// get all the paragraphs with a "dropcap" class – produces a nodeList
document.querySelectorAll(".dropcap");

// get all elements with a class of "dropcap" OR "huge"
document.querySelectorAll(".dropcap, .huge");

// get all paragraphs that have a class
document.querySelectorAll("p[class]");
```

`querySelector()` is an extremely flexible and powerful method in the JavaScript language. It can take multiple selectors, IDs, classes, attributes, and even advanced CSS pseudo classes to target elements. The only complaint at this time about these new methods is that the error messages are not very helpful, but hopefully this will be worked out in the browser.

Support

This is one of those things that works where you expect it to work. What do I mean by that? Many of the HTML5 features, especially the JavaScript APIs, don't have great support in older browsers, so earlier than IE 8 (IE 6 and 7) and earlier than Firefox 3.6 is generally a good support bottleneck; this is where a lot of the newer features start to fail. This is something to keep in mind as you're developing. Let's take a look at Table 4.2 for the support of this feature.

Table 4.2 **Browser Support for `querySelector` and `querySelectorAll`**

Internet Explorer	8.0 & up
Firefox	3.5 & up
Chrome	1 & up
Opera	10 & up
Safari	3.2 & up

Because selectors are such an important part of your JavaScript code, it's always very important that they function the way you want them to. For things like `querySelector()` you need to check out your own browser statistics and decide if it's something you can safely use. It's that same old story—we need to start moving people off old browsers, which can, unfortunately take some time.

Just like any of these selectors that require a small support boost, libraries are available to help you detect for feature support and use an appropriate method based on those results. These support solutions fall under the same category as the previously mentioned `getElementsByClassName()` method. Often when using a JavaScript library, you won't know for sure which method is being used because the library will figure that out for you; but it is very important to know that they exist as options.

Working with the Attribute Node

Now that we have a strong grasp of how to work with and target element nodes, we can start working within those nodes and play around with the *attribute node*. As illustrated in Figure 4.5, an attribute is contained inside an element node, but is not considered to be a child of that node, so we need some special methods to get, modify, or remove that information. Those methods are

- `getAttribute()`
- `setAttribute()`
- `removeAttribute()`
- `hasAttribute()`

In all these methods we target the attribute and return the value, so in an example of `class="visible"`, we would be targeting "class" and returning the value "visible".

Getting and changing attributes is paramount to writing a Web application that performs well. Because many of the most common interactions we create, like hide/show, can be created more efficiently through CSS, those can be as simple as adding or removing a class. In this example we will be working with the following HTML shown in Listing 4.6.

Listing 4.6 **HTML Example for Attribute Nodes**

```
<img src="images/placeholder.png" class="show" alt="placeholder image" id="plc">
```

Getting an Attribute

Before we can get an attribute, we need to jump into the DOM at an element node (using one of the methods we just learned about), and then we can gain access to that element's attribute. This is one of the ways we gain more information about an element. The `getAttribute()` method takes one argument, which is the name of the attribute you want to target.

Before we target an attribute, much like we did with checking the NodeList length, we want to make sure an element contains the attribute we're looking for. This can be done with the `hasAttribute()` method. Suppose you want to target the image referenced in Listing 4.6 and get the class value. The JavaScript in Listing 4.6.1 will illustrate how you might accomplish that task.

Listing 4.6.1 **JavaScript Used to Get the Class Value of Our Image**

```
// first we target the element and check if it has a class on it
if(document.getElementById("plc").hasAttribute("class")) {

    // after we know a class exists, we can then get the value
    document.getElementById("plc").getAttribute("class");

    // this would return the value of "show", which is the class on our image
}
```

You may be well aware that a class exists in the area you're targeting, but using `hasAttribute()` is more of a best practice, just in case it doesn't exist (it happens). We don't have to waste valuable processing time executing the JavaScript for no reason.

Setting an Attribute

Setting an attribute is valuable in many instances, whether you're temporarily saving data, setting a class, or replacing an attribute with something new. This method has a built-in if statement, so we don't need to run our conditional on it unless we want to do something other than just retrieving an attribute value.

This method takes two arguments—the first is the name of the attribute you want to set, and the second is the value you want to set it to. See Listing 4.6.2.

Listing 4.6.2 **JavaScript Used to Set the Class Attribute of Our Image**

```
// replace the current class with a value of "hidden"
document.getElementById("plc").setAttribute("class", "hidden");

// add a new attribute to the image
document.getElementById("plc").setAttribute("title", "we moved this element off
➥screen");
```

In this example, we changed the class from "`visible`" to "`hidden`" and, for better accessibility and screen readers, we added a title attribute to let them know we're not displaying this image anymore. Because the class already existed, our new class overwrote it. The title attribute, on the other hand, was not already on the element, so it was created for us (you just dynamically created your first node in the DOM—congrats!). Changing display properties like this allows us to maintain the separation between presentation and behavior that we spoke about in Chapter 1 with progressive enhancement. With this example, you might consider the following CSS in Listing 4.6.3.

Listing 4.6.3 **CSS to Apply to Our Image**

```
/* this is the default state of the image */
.visible {
    position: static;
}

/* this is applied when our class is changed and moves the image out of view */
.hidden {
    position: absolute;
    top: -9999px;
    left: -9999px;
}
```

Note

It's important to not use `display: none` when working like this in CSS because screen readers cannot view content that is set to `display: none`. That is why we choose to hide content like this by simply moving it off the screen.

Removing an Attribute

Removing an attribute is as simple as getting one. We just target the element node and use the method `removeAttribute()` to get it out of there. There are no JavaScript exceptions thrown if you try to remove an attribute that doesn't exist, but it's still a best practice to use the same `hasAttribute()` method we mentioned earlier, shown in Listing 4.6.4.

Listing 4.6.4 **JavaScript Used to Remove the Class Value of Our Image**

```
// first we target the element and check if it has a class on it
if(document.getElementById("plc").hasAttribute("class")) {

    // after we know a class exists, we can then remove it
    document.getElementById("plc").removeAttribute("class");

}
```

In this example, we simply removed the class from our image.

Support

These methods have full support in all browsers and are okay for you to freely use in all browsers.

Working with the Text Node and Changing Content

Now that we have stepped through both element and attribute nodes, we can move onto our last type of node, the text node. When you're working with text nodes, most of the work you will be doing is modification of the content. There is also no traveling about inside a text node, because it can't have any children. In some aspects the way you will be interacting with these nodes is similar to the way you interact with the attribute node. You will be modifying, getting, and removing them, but using different methods to do so.

Changing content is extremely common with JavaScript. Whether it's for a button text toggle value or the Ajax loading of content, you will find yourself inserting text or HTML into the DOM from time to time. If you want to change or modify all the content in an area rather than parsing through the DOM, you can do it in one fell swoop with the JavaScript method `innerHTML`. There are a lot of benefits to this method, notably performance. It is a lot faster to wipe out a large chunk of information than it is to do the same thing on each individual DOM node you want to replace. In Listing 4.7 you will see the HTML snippet this example will be working with.

Listing 4.7 **HTML Example to Illustrate Changing Content**

```
<body>

    <div id="target-area">
        <p>This is our text.</p>
    </div>

</body>
```

In this example, we want to target the text node and change it to "hello world". This content could come from anywhere, such as an Ajax call, or it can be hard-coded in, like in the example from Listing 4.7.1.

Listing 4.7.1 **Using `innerHTML`**

```
// targeting the text node
document.getElementById("target-area").innerHTML;

// changing the content in the text node — you can jump straight to this.
document.getElementById("target-area").innerHTML = "<p>hello world</p>";
```

Moving Around the DOM

When moving around inside the DOM, we start to really use the tree structure of the DOM and benefit from knowing the layout of our HTML document and how the elements relate to each other. To this point in the chapter, we have talked about targeting all the different types of DOM nodes, but they have all been direct access via jumping into the middle of the document by ID, class, tag name, or by specific position in the NodeList. Unfortunately, in the world of writing clean, semantic HTML, we tend to not riddle our HTML with tons of classes and IDs (well, some do but let's not get off on *that* tangent right now), and sometimes you'll only be able to reasonably jump so far into the tree. From there, you will have to navigate up and down with some of the most useful methods native to JavaScript. In this section we'll be traveling around the DOM with five new methods:

- `parentNode`
- `previousSibling`
- `nextSibling`
- `firstChild`
- `lastChild`

Just as you can get the children of an element node, you can also get its parent (or parents), and they all have parents. Let's take a look at the HTML in Listing 4.8 and assume that we want to get the parent element for the target node and add a class of "active" to it.

Listing 4.8 **Basic HTML Example of Navigation**

```
<ul id="nav">
    <li><a href="/" id="home">Home</a></li>
    <li><a href="/about" id="about">About Us</a></li>
    <li><a href="/contact" id="contact">Contact Us</a></li>
</ul>
```

Using the JavaScript in Listing 4.8.1, you can execute the desired behavior.

Listing 4.8.1 **Targeting a `parentNode`**

```
// target the "about" link and apply a class to its parent list item
document.getElementById("about").parentNode.setAttribute("class", "active");
```

Besides moving in a parent-child relationship (up and down), you can also move side to side in the DOM and target nodes that sit next to a target node. These are called **siblings**, and two special methods allow us to target them: `previousSibling` and `nextSibling`. In our example we still want to target the About Us link, but instead of adding a class to the parent element, we want to add a class to Home (`previousSibling`) and Contact Us. We could use the JavaScript in Listing 4.8.2.

Listing 4.8.2 **Adding a Class to Previous and Next Sibling Nodes**

```
// get "about" parent, then its previous sibling and apply a class
document.getElementById("about").parentNode.previousSibling.setAttribute "class",
➥"previous");

// get "about" parent, then its next sibling and apply a class
document.getElementById("about").parentNode.nextSibling.setAttribute("class", "next");
```

The JavaScript from Listings 4.8.1 and 4.8.2 should generate the HTML in Listing 4.8.3.

Listing 4.8.3 **Generate HTML After Adding Classes**

```
<ul id="nav">
    <li class="previous"><a href="/" id="home">Home</a></li>
    <li class="active"><a href="/about" id="about">About Us</a></li>
    <li class="next"><a href="/contact" id="contact">Contact Us</a></li>
</ul>
```

Using these methods will help you travel around your HTML document without having to riddle it with extra markup, IDs, and classes, while keeping performance at a manageable level. However, always evaluate the load time of a page and balance it with best practices you'd like to follow. In other words, if moving up and down the DOM tree is too resource intensive for any particular case, you may want to consider adding a class or ID to jump to. (I know, sorry to be like that, but performance is always a gray area—use your best judgment.)

Accessing First and Last Child

JavaScript offers us some alternative routes to get to node information; for example, moving straight to the first or last item in a NodeList. This is sort of like the jumping around we do with `getElementById` and `getElementsByTagname`, but like `parentNode` and `childNodes`, they're at the NodeList level. So, you would target down to the NodeList and dive in from there compared to doing it from the document level. But enough of that. Let's dive back into our HTML example from Listing 4.8 and try to add classes to the first and last list items in our navigation with the JavaScript from Listing 4.8.4.

Listing 4.8.4 **Adding a Class to the First and Last Items in Our Nav**

```
// travel to the first node and add the class
document.getElementById("nav").firstChild.setAttribute("class", "first");

// travel to the last node and add a class
document.getElementById("nav").lastChild.setAttribute("class", "last");
```

And that will generate the HTML in Listing 4.8.5.

Listing 4.8.5 **Generate HTML After Adding Classes**

```
<ul id="nav">
    <li class="first"><a href="/" id="home">Home</a></li>
    <li><a href= "/about" id="about">About Us</a></li>
    <li class="last"><a href="/contact" id="contact">Contact Us</a></li>
</ul>
```

Moving up and down the DOM is something you will do a lot, so it's worth spending some time on these methods to wrap your head around what's going on when you are using them on a real project.

Note

These methods can be a little frustrating to work with because some browsers will insert a text node to maintain whitespace in your code. In the example we were using, this can cause a return of "#text" rather than the first node you're expecting. These examples have been idealized for readability, but in some cases you will want to remove the whitespace or adjust your targeting methods, Removing whitespace is a good idea for a lot of sites in production because it will make your overall page weight smaller and improve performance.

Dynamically Adding and Removing Nodes from the DOM

Up to now, we have been accessing nodes in the DOM that already exist. Often when building an interface, you'll need to dynamically insert nodes into the DOM, create new elements, fill them with content, and drop them into the document somewhere. Luckily, we have this capability built into JavaScript. In this section, we will be introducing four new methods:

- `createElement()`
- `createTextNode()`
- `appendChild()`
- `removeChild()`

The first three methods will inevitably intertwine as you insert new elements into the DOM, so we will go over them all at once, and use the HTML from Listing 4.9.

Listing 4.9 **HTML Example to Create a New Element**

```
<body>
    <div id="target-area">

        <p id="tagline">Hello World!</p>

    </div>
</body>
```

Adding Elements to the DOM

Creating and inserting a new element is generally a three-step process:

1. Create the element.
2. Fill it with content (or leave it blank).
3. Put in the DOM.

First, we need to use the `createElement([tagname])` method to create the element. This method takes one argument, which is the tag name of the element you want to create. When I say we create the element, it doesn't actually exist anywhere until we put it in the DOM; it helps to perceive this method as thinking up an idea. The idea is floating around your head, but isn't a physical thing until you put it on paper, tell someone, or build it.

The second step is to add content to your element. You can do this right away, or you can do it later, but eventually the element will be filled with something; what good is an empty element? We use the `createTextNode()` method to do this. It takes one argument, which is a string of text that will be inserted into the element.

The last step is picking a point in the DOM to insert your new element. We do this with the `appendChild()` method. This method takes one argument, which is the completed element you want to insert. Then, like magic, you've created a new element.

We went over creating new nodes earlier in the chapter with the `setAttribute()` method because it also can dynamically create nodes in the DOM. Now let's look at an example of creating a new element; remember, we'll be using the HTML from Listing 4.9 for this exercise and applying the JavaScript from Listing 4.9.1.

Listing 4.9.1 **HTML Example to Create a New Element**

```
// store the target area to a variable to keep things neat
var targetArea = document.getElementById("target-area");

// create our <p> element
var p = document.createElement("p");

// create a text node inside the <p>, note that we're using a variable "p" here
var snippet = p.createTextNode("this was a generated paragraph");

// insert our generated paragraph into the DOM
targetArea.appendChild(snippet);
```

The generated HTML should look like Listing 4.9.2.

Listing 4.9.2 **New Element Inserted into the DOM**

```
<body>
    <div id="target-area">
        <p id="tagline">Hello World!</p>
        <p>this was a generated paragraph</p>
    </div>
</body>
```

Removing Elements from the DOM

Just as you can add elements to the document, you can also take them away. It is a similar process, but instead of needing three methods, we need only one. It takes a year to build a house, but only a few minutes to bring it down.

Using the same HTML from Listing 4.9, let's say that instead of creating a new paragraph, we want to remove the tagline. We can do this by navigating to its parent and removing the child with the `removeChild()` method, as shown in Listing 4.9.3.

Listing 4.9.3 **JavaScript for Removing an Element from the DOM**

```
// store the target area to a variable to keep things neat
var targetArea = document.getElementById("target-area");

// store the tagline in a variable as well
var tagline = document.getElementById("tagline");

// remove it from the DOM
targetArea.removeChild(tagline);
```

Now you are all set: you can add and remove elements from the DOM at will, and you're officially on your way to becoming a great JavaScript mind in the industry!

Summary

In this chapter, we learned about the Document Object Model, or the DOM, and the different types of nodes that construct the tree and all its branches. We went over how to map out an HTML document and talked about how the DOM related to JavaScript.

We also learned about how to navigate the DOM tree to target specific nodes or elements you want to store or modify. Besides navigating up and down the tree, we discovered many ways to jump directly to a node based on information attached to it, such as IDs, classes, and even the tag name itself.

Later on we created new DOM nodes, filling them with content and then inserting them into the document.

Exercises

1. What are the four types of DOM nodes?
2. Explain what the DOM is and how it relates to HTML and JavaScript.
3. What happens if you try to set a node attribute for an attribute that does not exist on the node already?