

## 8.2 DOM Element Objects

In our previous examples we have seen DOM objects for elements on a page. But our interaction with these objects has been very limited: we've set the `innerHTML` for `span` elements and we've interacted with the `value` of text boxes. These DOM objects are actually rich with other properties and methods we can use in our JavaScript code. In this section we'll explore some of the basic features of DOM objects and interacting with them. We'll explore DOM interactions in much more detail in the next chapter.

### 8.2.1 Interacting with Text

Much of the manipulation we want to perform on DOM objects involves getting, manipulating, and setting text. The text inside an element is accessible as a string, and we can manipulate these strings using the string methods shown in the previous chapter. But the means of accessing the element text differs for various elements, so it merits some discussion here. There are also some browser incompatibilities related to text content.

The ECMAScript-standard way of changing the text inside an element is to set its `textContent` property. This property is supported by all standards-compliant browsers, but unfortunately not by Internet Explorer, which uses a non-standard property named `innerText` instead. Because of this ugly incompatibility, so far in this textbook we've used the `innerHTML` property instead to set text. Even though `innerHTML` is not currently part of the ECMA JavaScript standard, it will be in the next version of the ECMAScript standard and is already supported by every major browser. These properties are summarized in Table 8.9.

Property Name	Description	Supported By
<code>innerHTML</code>	text and/or HTML tags inside a node	all browsers (non-standard)
<code>innerText</code>	text (without HTML tags) inside a node	IE only
<code>textContent</code>	text (without HTML tags) inside a node	all browsers except IE
<code>value</code>	text value inside a form control	all browsers

**Table 8.9 Various DOM properties for getting text/HTML content**

For example, to set the paragraph with an `id` of `error` to show a particular error message, you could write the following code, which would work on standards-compliant browsers:

```
var errorArea = document.getElementById("error");
errorArea.textContent = "Error: Missing last name";
```

If you want the code to work on all browsers including Internet Explorer, you could write:

```
// more cross-browser compatible
var errorArea = document.getElementById("error");
errorArea.innerText = errorArea.textContent = "Error: Missing last name";
```

We find the dual assignment tedious and therefore use the `innerHTML` property instead to set the text. Technically setting `textContent` would be more stylistically correct, but the browser compatibility problems make it too cumbersome for our taste.

```
// also cross-browser compatible
var errorArea = document.getElementById("error");
errorArea.innerHTML = "Error: Missing last name";
```

An interesting feature of the `innerHTML` property is that it can be used not only to get/set text content, but also to add new HTML elements and tags to a page. This ability is very powerful, but it can be abused and often leads to ugly code. Example 8.17 demonstrates the use of `innerHTML` to add tags to a page. We strongly discourage this style; we'll show a better style later in this chapter for adding new elements to a page using a method called `document.createElement`.

```
// bad code (don't insert HTML tags using innerHTML!)
var div = document.getElementById("mainarea");
div.innerHTML = "<a href=\"foo/bar.html\">Check it out!</a>";
```

### Example 8.17 Abusing innerHTML

#### Example Program: Shuffler

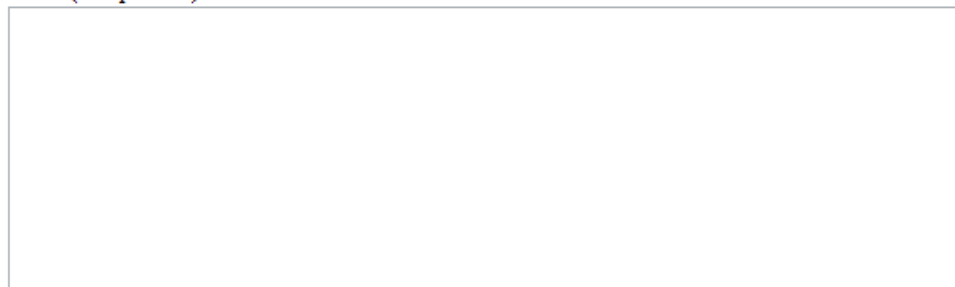
We've already seen that to get and set the text of most normal elements, you use the previous properties such as `innerHTML`. If the element in question is a form control, such as an `input` text box or `textarea`, we access the text using the `value` property instead. You can also set new text to appear in the element by assigning `value` a new string.

Suppose we want to write a page that lets the user type in lines of input into a `textarea`, with a Shuffle button below that randomly rearranges the order of the lines when clicked. Example 8.18 shows the relevant HTML code and its appearance in the browser.

```
<h1>Ye Old Shuffle Tool</h1>
<div>
  Items (one per line): <br />
  <textarea id="items" rows="10" cols="80"></textarea>
  <button id="shuffle">Shuffle It!</button>
</div>
```

## Ye Old Shuffle Tool

Items (one per line):



Shuffle It!

### Example 8.18 Shuffler HTML code

Probably the trickiest part of this program is the actual algorithm for shuffling the array. Since it is difficult and also might be of general use in other programs, let's write a function called `shuffle` that accepts an array as a parameter and shuffles that array's elements. Keeping the shuffling algorithm from the DOM and event code will help us keep the code cleaner and avoid bugs.

A simple algorithm for shuffling is to loop over the elements of the array, choosing a new random index for each element and swapping it to that index. To make sure that the algorithm is fairly

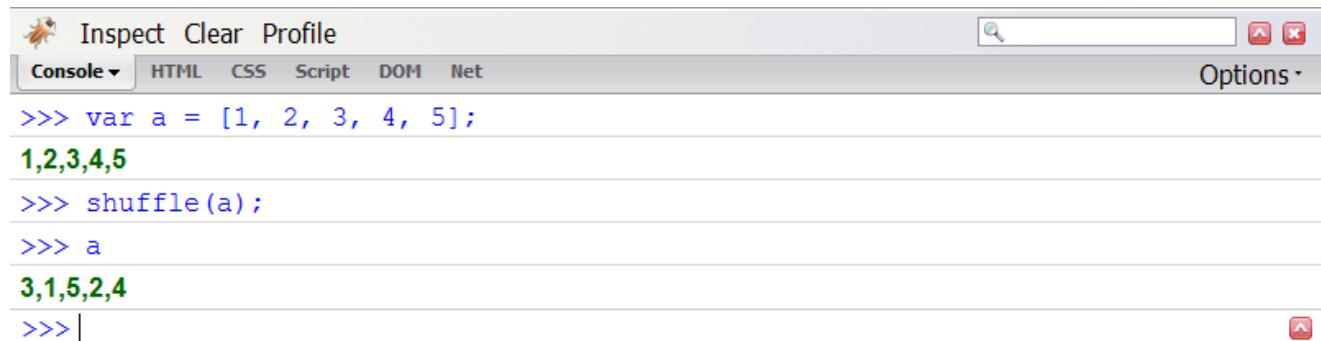
balanced (that each arrangement of the elements is equally likely), we must make sure to choose a random index greater than or equal to the element's current position. (The proof that our algorithm is balanced is outside the scope of this textbook.) Example 8.19 shows the code.

```
// Randomly rearranges the elements of the given array.
function shuffle(a) {
  for (var i = 0; i < a.length; i++) {
    // pick a random index j such that i <= j <= a.length - 1
    var j = i + parseInt(Math.random() * (a.length - i));

    // swap the element to that index
    var temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }
}
```

### Example 8.19 Array shuffle code

It can be tough to get a tricky function like this working straight away. We suggest typing in the `shuffle` function into your `.js` file, then opening the page in Firefox and popping up the Firebug console. Then declare one or two short arrays and call the `shuffle` function on them. Inspect the results to make sure that they look suitably random. Once we're convinced that our function works, we go on to the rest of the code. Figure 8.1 demonstrates this technique.



**Figure 8.1** Testing `shuffle` function in Firebug console

Now that our shuffling algorithm is implemented, the rest of the code to attach the event handlers can be written. When the Shuffle button is clicked, we'll grab the text from the `items` text area, split it into an array of lines using its `split` method, and then call the `shuffle` function on that array of lines. Example 8.20 shows the code.

```
window.onload = function() {
  document.getElementById("shuffle").onclick = shuffleClick;
};

function shuffleClick() {
  var items = document.getElementById("items");
  var lines = items.value.split("\n"); // split into lines
  shuffle(lines);
  items.value = lines.join("\n"); // put back into text area
}
```

### Example 8.20 Shuffler JavaScript code

## 8.2.2 Adjusting Styles

The DOM can be used to modify the styles and onscreen appearance of page elements. This is done through the DOM object's **style** property. This property represents the HTML element's **style** attribute and directly connects to the CSS for that element. The **style** property is not a string but instead an object that contains dozens of properties, one for each possible CSS property of that element. The syntax for accessing these properties is shown in Example 8.21.

```
element.style.property           // get value
element.style.property = value;  // set value
```

### Example 8.21 DOM style property

For most of the CSS properties, there is a corresponding DOM **style** object property with the same name. For example, suppose you have the following HTML code in your page:

```
<p id="slogan">Eat at Joe's. You can't beat our prices!</p>
```

To set the preceding paragraph's text color to red, you would use the following JavaScript code:

```
var paragraph = document.getElementById("slogan");
paragraph.style.color = "red";
```

The names of the DOM style properties are as similar as possible to their CSS counterparts, but there are a few small differences. DOM style properties can't have dashes in them because `-` is the subtraction operator in JavaScript. Wherever there would have been a dash, the DOM property's name capitalizes the next letter. For example, the CSS **background-color** property is called **backgroundColor** in the DOM.

Table 8.10 lists several examples of the mapping between CSS property names and DOM property names. This is obviously an incomplete list, since there are too many style properties to list here. But this chapter's References section points to a page on the W3Schools web site that has a complete list of all style properties. All values for DOM style properties are stored as strings, with an empty string `""` as the value if the property is not set.

CSS Property Name	DOM Property Name	Example
background-color	backgroundColor	"#ff00dd"
border	border	"1em solid red"
border-top-width	borderTopWidth	"3px"
color	color	"red"
float	cssFloat	"left"
font-weight	fontWeight	"bold"
font-size	fontSize	"12pt"
z-index	zIndex	"456"

**Table 8.10 DOM style property names**

A DOM object's **style** property is useful for setting new styles but has some issues when trying to examine pre-existing element styles defined in CSS, which we'll explore in a later section. In the next chapter we will learn a workaround for this limitation.