4. The `document` Object and the DOM

After Chapter 3, you should be able to write JavaScript programs that are integrated with a web page, communicate with the user, and solve problems. Congratulations! You still have a long way to go.

Up to this point, your interactions with the user have entirely made use of pop-up dialogs. While this style of communication is sometimes used in **web apps**, most input usually consists of mouse and keyboard **events** (i.e. moving the mouse, clicking, and typing) and most JavaScript output consists of modifications to the web page (e.g. changing colors and styles, revealing and hiding menus, changing the contents of an **element**, loading and displaying new content, etc.). We'll start with the output first in this chapter, then Chapter 5 will show you how to respond to keyboard and mouse input. By the end of these two chapters, you'll have the tools you need to write much more professional web apps.

4.1 The Document Object Model

Every JavaScript program running in a web browser has access to a `document` **object**. This object holds the browser's internal representation of the **Document Object Model** (**DOM**), and contains all the information the browser retrieves from the **HTML tags** and **attributes**, **CSS style rules**, images, and other components that make up the source code of the page. Understanding the DOM is key to becoming an effective JavaScript programmer.

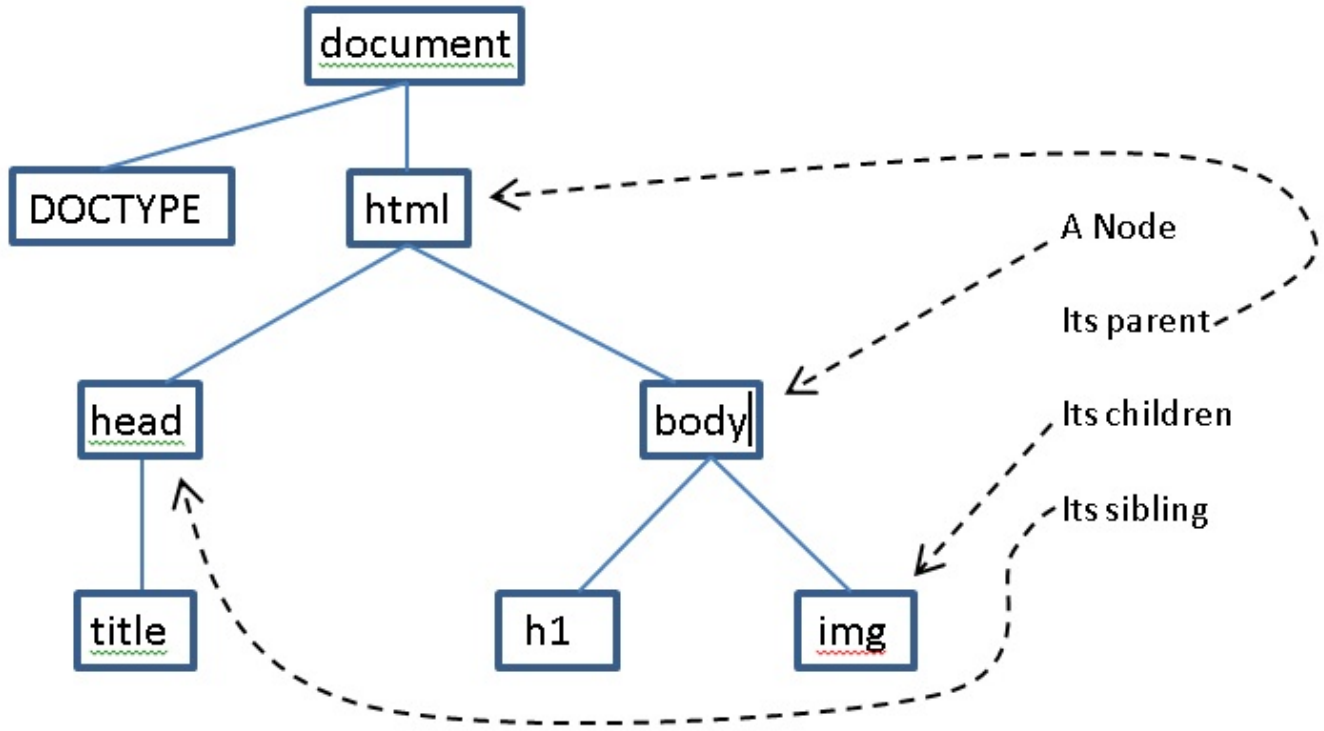
Like any bracketed structure, an HTML page can be viewed as a hierarchical family **tree** of **elements** containing other elements. When the browser reads an HTML source page, it constructs an object for each element, and links it to the elements it contains (the "children") and also links it to the element that contains it (the "parent").

Do It Yourself

Load **helloworld.html** from the **example pack**, and look at the page source:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <h1 id='message' class='heading'>Hello, World!</h1>
        <img src='images/smiley.jpg' alt='smile image'>
    </body>
</html>
```

In the example above, the document consists of a `<!DOCTYPE>` element and an `<html>` element. The `<html>` element contains a `<head>` and a `<body>`. The `<head>` contains a `<title>` element and the `<body>` contains an `<h1>` element and an `<img>` element. This set of relationships can be displayed in a tree diagram like the one shown below.



The items in the boxes are referred to as **nodes**. Each node is a JavaScript object that has been constructed by the browser to represent the corresponding **HTML element**. Each node object contains information about the attributes, CSS style and contents of the element it represents.

A node can have one parent (above it), any number of children directly below it, and any number of siblings (nodes with the same parent). The root node is at the top and the leaf nodes are the ones at the bottom with no children, which makes this a curious sort of upside down "tree". When you are using JavaScript in a web page, the built-in **global variable**, `document`, holds the root node of this tree.

Do It Yourself

Open **helloworld.html** from the **example pack**. Hit F12 and instead of the Console tab, go to the **Elements** tab. This is a representation of the browser's **DOM**.

Now right click the `<img>` element, select "add **attribute**", and give this image a style attribute (e.g. `style="width:100px"`). You should see the change reflected on the page immediately. Now check the page source, and you will see that it has not changed. Why not?

Now go to the Console, type `document`, and hit enter. Click the response to open the DOM and explore it. This `document` **object** gives you access to the browser's DOM from within a JavaScript program.

There is a lot more to say about the structure of the DOM than this, but the information in this section is good enough to get you started.

4.2 What is a JavaScript Object? (An Important Aside)

The last section introduced the notion of an **Object**, so we better say a few words about that before we

move on.

Just like in Java, a JavaScript object is a package of **variables** (**fields**) and code (**methods**) stored together under a single **reference** or name. In your experience with Java you have probably used objects before, and you may even have created your own objects.

For example, in JavaScript every **string** object has a field named `length` and a method named `charAt` (just like the Java `String` object). The code in the DIY box below creates a string and displays its length and first character. Note the use of the variable name, `s`, with the dot operator to access the variables and methods that belong to the object referenced by `s`.

Java Connection

Objects. JavaScript makes extensive use of **Objects**, and supports the creation of custom objects and an **object-oriented** style of programming. But JavaScript also supports other programming styles (e.g. **imperative**), and as you will see later, JavaScript objects are different in many ways from their Java counterparts. But for now you can treat JavaScript objects just like Java objects.

Do It Yourself

Try this in a console window or a `<script>` **element** of a web page. Predict the result of each line before you run it.

```
var s = "JavaScript is Cool. ";
alert(s.length);
alert(s.charAt(0));
```

The second line in the code above accesses the `length` field associated with the object `s`. The third line calls the `charAt` method with the parameter 0 to get the first character from the object `s`. (Remember that there is no `char` type in JavaScript so the **return value** is a string of length 1.)

Java Connection

Object Syntax. There are two ways of accessing an object's **fields** and **methods** in JavaScript.

Like Java, you can write `objectName.fieldName`.

Unlike Java, you can also write `objectName["fieldname"]`.

One thing that is a little different about JavaScript is that you can also use square bracket notation to access an object's variables and **functions**. For example, instead of `s.length` and `s.charAt`, you can write `s["length"]` and `s["charAt"]`, as shown in the DIY box below.

This square bracket object notation looks and behaves exactly like a data structure called an **associative array** (or sometimes a "map" or a "dictionary"). Indeed, there is really no difference between an object and an associative array in JavaScript, so both styles of referencing are included for the convenience of the programmer.

Do It Yourself

Try this in a console window or a `<script>` **element** of a web page.

```
var s = "JavaScript is Cool. ";
alert( s["length"] );
alert( s["charAt"](0) );
```

This square bracket style of accessing an object's **fields** and **methods** might seem strange, but it has its uses. For example, try the following (after typing the lines above). Make sure you type something legal like "length" when prompted.

```
var fieldName = prompt("type a field name");
alert( s[fieldName] );
```

4.3 Retrieving and Manipulating a DOM Node

There are lots of ways to access and dig through the **DOM tree** from within a JavaScript program, but by far the easiest way to retrieve a **node** is by using its `id` **attribute**. To do this, use the `getElementById` **method** (note the lower-case `d`) of the `document` **object** to reach into the DOM and grab the Node you want. Once you have it, you can access and/or change the contents of the **element**, its style information, or any of its attributes.

Any element you retrieve using `document.getElementById` will be an object of type `Node`. Node objects contain a number of **fields**, each containing information associated with the corresponding **HTML element**. These fields can be used to make changes to the DOM long after a page has been loaded.

Here are some of the more useful `node` fields...

`innerHTML`:	a String representing the contents of the node as HTML
`style`:	the CSS style information associated with the node
`className`:	the HTML class attribute
plus…	there will be one field for every attribute specified in the corresponding tag in the original HTML source code (e.g. `id`, `src`, `href`, `value`, `type`, etc.)

Do It Yourself

Open **helloworld.html** example from the **example pack**, go to the console and try the following commands, noting the **return values** in each case.

```
var node = document.getElementById("message");
node;
node.innerHTML;
node.style;
node.className;
```

Now add an `id` **attribute** to the `<img>` **element**, either by changing the source code, or by changing the **DOM** through the Elements view (see a previous DIY box). Now use JavaScript **statements** similar to the ones above to retrieve the `src` and `alt` attributes of the image.

An alternative to the `getElementById` method is `querySelector`. You can use `querySelector` with any **CSS selector** and it will return the first matching element (or `null` if no elements match).

For example, `document.querySelector("h1")` returns the first `<h1>` element in the DOM, `document.querySelector("h1.main")` returns the first `<h1>` element with a **class** of `main`, and so on. And of course, `document.querySelector("#myId")` is equivalent to `document.getElementById("myID")`.

Do It Yourself

Open the browser console for this page (the one you're reading) and try the following to get the first DIY box and the first Java Connection box in this textbook

```
document.querySelector("div.DIY")
document.querySelector("div.JC")
```

The `querySelector` method can sometimes come in handy, but we will continue to just use `getElementById` in the examples that follow.

4.3.1 Changing an Element's `innerHTML`

The `innerHTML` property returns a **string** representation of the content of the node. The content is everything that appears between the opening and closing tags that define that node. Changing this string is probably the easiest way to rewrite part of the DOM.

(Be careful with case here. If you type `innerHtml` instead of `innerHTML`, the command may fail without even giving you an error message!)

The DIY example below shows you how to change the text contents of an element.

Do It Yourself

Load **innerHTMLExample1.html** from the **example pack**, and let it do its thing. After you press OK, you will see the text of the heading change.

The relevant code is shown below:

```
<h1 id="heading">A Simple Example</h1>
<p>Here is some text in a paragraph.</p>
<script type="text/javascript">
    alert("Press OK to see me change my own heading.");
    document.getElementById("heading").innerHTML = "Done";
</script>
```

See if you can rewrite this code so that instead of just rewriting the heading, it also rewrites the `<p>` **element**.

You can also place **HTML tags** into the `innerHTML` field of a node. When you do that, the browser will read the tags, create node objects for the corresponding HTML elements, and add them to the DOM, as demonstrated in the DIY example below. If the `innerHTML` of the element you rewrite already contained other HTML elements, the corresponding nodes will be discarded from the DOM and replaced with new ones.

Do It Yourself

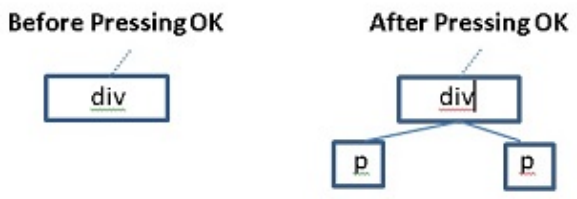
Load the file **innerHTMLExample2.html** from the **example pack** into a browser, and let it do its thing. After you press OK, you will see the div change to have two new paragraphs added as contents. Hit F12 and go to **Elements** to verify that two new **nodes** have been added to the **DOM**.

The relevant code is shown below:

```
<div id='target'>Here is some text in a div.</div>
<script type='text/javascript'>
    alert("Press OK to see me change my heading.");
    document.getElementById("target").innerHTML =
    "<p>First paragraph</p><p>Second paragraph</p>";
</script>
```

Rewrite the code so that it asks the user for some HTML and then inserts whatever the user types into the `<div>`.

Here is a diagram of the relevant part of the DOM before and after executing the script in the DIY example above.



<exercises section='4.3.1'>

1. *From the first DIY box above:* Modify the code for **innerHTMLExample1.html** so that it rewrites the <p> **element** as well as the <h1> element.

2. *From the second DIY box above:*

   a. Modify **innerHTMLExample2.html** so that it prompts the user for some text, and then changes the contents of the to match what they typed.

   b. Test your code and try entering HTML at the prompt and investigate what happens to the **DOM**.

   c. Why do you have to create the script after the element? What happens if you place it earlier in the file?

   d. Modify the script again so that it asks the user to enter a positive integer. Then alter the contents of the element to display the integers from 1 up to the number the user entered. Each integer should be contained in a separate <p> element. (Note this will require some **string** processing with a **loop**.)

3. Get **addition.html** from the **example pack**. Read the comment header in the page source and follow the instructions to complete the program.

</exercises>

4.3.2 Changing an Element's `style`

Changing an element's CSS information can be done most easily by accessing the `style` field of the corresponding DOM node. This `style` object contains all the in-line CSS information for the Node, where each field of the object corresponds to a CSS property.

Do It Yourself

Load **innerHTMLExample1.html** from the **example pack**, and press OK when prompted. Now hit F12 to open the console and type the following:

```
var e = document.getElementById("heading");
e.style.color = "red";
```

Now try changing other CSS styles in the same way. Add a border, change the font, position the header below the paragraph, etc.

Now go to the **Elements** view and look at the <h1> element. You will see style information in the element's `style` **attribute**. Note that the page source does not contain this information.

In addition to assigning or changing style information, you can also read style information from the fields of the `style` object, but you cannot read any CSS properties that were defined as part of a **style sheet** in this way.

Do It Yourself

Load **innerHTMLExample2.html** from the **example pack**, and press OK when prompted. Note that the `<div>` **element** is red with a border.

Now hit F12 to open the console and type the following:

```
var e = document.getElementById("target");
e.style.color;
e.style.border;
```

Why does one of the above lines give you a value but not the other? View the original page source and see if you can figure it out.

The only wrinkle in accessing the style object above is that you can get into problems with the dot notation for hyphenated style properties like `background-color`. The problem is that the JavaScript expression `e.style.background-color` looks like we are subtracting the **variable** color from the field `e.style.background`. The solution implemented in most browsers is to convert hyphenated-property-names to **camelCasePropertyNames** (e.g. `border-radius-top-left` becomes `borderRadiusTopLeft`).

Do It Yourself

Load **innerHTMLExample1.html** from the **example pack**, and press OK when prompted. Now hit F12 to open the console and type the following:

```
var e = document.getElementById("header");
e.style.backgroundColor = "red";
```

Now change some other hyphenated properties, like `font-size`, `border-radius`, etc.

Note: Adding Fields to Objects

It is important to understand that when you assign new style information to a node you are actually adding new fields to the `style` object. This is something you can't do in languages like Java.

Unlike Java, JavaScript allows you to create a new field in any object at any time simply by assigning something to a new field name. This is very convenient, but it also means that some errors are hard to catch. For example, if you misspell a field or get the case wrong JavaScript will modify the object by adding a new field without warning you about what has happened.

Java Connection

Bad Field Names. In Java, if you mistype an object's **field** name, you will get a **syntax error** every time. In JavaScript if you try to read a non-existent field, you will get a runtime error in the console. But if you try to assign to a non-existent field, JavaScript will create a new field for you without any errors or warnings. This makes JavaScript **objects** very flexible, but it also makes some errors very hard to find.

For example if you type `node.innerHtml = "new content"`, the `innerHTML` field of the `node` object will not change because you got the case wrong on its name. But you will not get any error or warning messages either since JavaScript just adds a new `innerHtml` field to the `node` object. Something similar will happen if you type `node.style.colour="red"` (because of the Canadian spelling).

Note: Use of Square Brackets

Don't forget that you can access an objects' fields using the dot notation or **associative array** notation (see the section on JavaScript objects above). Many JavaScript programmers choose to use associative **array** notation for style properties. So in the most recent DIY box above, instead of `e.style.backgroundColor` many JavaScript programmers would prefer `e.style["backgroundColor"]`. Which method you use to access the `style` object's fields is up to you, but you need to know about the two alternatives in order to read other people's code effectively.

Do It Yourself

Go back and redo all the other DIY boxes in this section, changing the dot notation to **associative array** notation for all style properties. Every example should behave in exactly the same way as before.

<exercises section='4.3.2'>

1. Create a web page with a single "hello world" paragraph. Create a script on the page after this **element** that prompts the user for a color, and then sets the color of the "hello world" element according to what they typed. Why do you have to create the script after the `<p>` element? What happens if you place it earlier in the file?
2. Modify the script from question 1 so that the user can specify both the style property to change and the value (note that you will have to use square bracket notation to access the style property the user chooses).
3. Go back to the **addition.html** file from the **example pack**. Change the file so that the result **field** changes to Green if the user gets your question right, otherwise it shoudl be Red.
4. Modify **addition.html** again so that if the user fails to enter a valid number, the `<h1>` element turns into a solid red box with a thick black border around it.

</exercises>

4.3.3 Changing an Element's `class`

Sometimes you may want to change a large number of property/value pairs at once. In this case, a good option might be to define styles for two different classes, and then change the `className` field of the element. This will automatically update the style of the element to match its new class, effectively changing many styles at once.

Do It Yourself

Load **classNameChangeExample.html** from the **example pack**. Now hit F12 to open the console and type the following:

```
document.getElementById("maindiv").className="classtwo"
```

You should see a big change. Take a look at the source code for this file to see how this change happened. Can you use a single command to change it back to the way it was originally?

(Advanced Programming Note: If you need an object to have multiple classes associated with it, you can use the `classList` api for that object. You won't find this on w3schools, so here's a link to the **Mozilla Developer Network ClassList [https://developer.mozilla.org/en-US/docs/Web/API/Element.classList]** page.)

<exercises section='4.3.3'>

1. Go back to **addition.html** and create **CSS rules** for two **classes** named `correct` and `incorrect`. The `correct` class should make an **element** big, centered in the middle of the screen, with a border and a happy background color. The `incorrect` class should make an element small, positioned in the lower right, with sad colors. Then modify the script to change the class of the result element appropriately depending on whether the user gets your question right or wrong.

</exercises>

4.3.4 Changing Other Attributes

You can also access, change or add any other attributes of an element by accessing the fields for those attributes. For example, if you want to change an image, you can modify its `src` attribute by accessing the `src` field of the corresponding node. Or if you want to change where a link, you can modify the `href` attribute of the corresponding node.

Do It Yourself

Open the chrome console for this page and execute the following command to retrieve the **node** corresponding to this DIY box:

```
node = document.getElementById("testDIY");
```

Now add a `title` attribute:

```
node.title="Read Me!";
```

Now hover the mouse over this DIY box to see the effect of this **statement**. Then right-click and choose "Inspect **Element**" to see the **attribute** you added to the **DOM**.

This link goes to Google. It has an `id` of "testLink". Can you enter commands in the browser console to redirect the link to **facebook [http://www.facebook.com]** instead?

<exercises section='4.3.4'>

1. Get the file **mood.html** from the **example pack** and follow the instructions in the comments at the top. Here are the image files you will need (right click and save them to an appropriate folder):

</exercises>