10
Dynamic Element Manipulation Using the DOM

Learning the difficult concepts of the previous chapter will be rewarded in this chapter. We will take our DOM knowledge one step further and learn how to manipulate the DOM elements on the page with JavaScript. First, we need to learn how to navigate the DOM and select the elements we want. We will learn how we can add and change attributes and values, and how to add new elements to the DOM.

You will also learn how to add style to elements, which can be used to make items appear and disappear. Then we will introduce you to events and event listeners. We will start easy, but by the end of this chapter you will be able to manipulate web pages in many ways, and you will have the knowledge to create basic web apps. The sky is the limit after getting this skill down.

Along the way, we will cover the following topics:

- Basic DOM traversing
- Accessing elements in the DOM
- Element click handler
- This and the DOM
- Manipulating element style
- Changing the classes of an element
- Manipulating attributes
- Event listeners on elements
- Creating new elements

Note: exercise, project and self-check quiz answers can be found in the *Appendix*.

We have learned a lot about the DOM already. In order to interact with our web page and create a dynamic web page, we have to connect our JavaScript skills to the DOM.

Basic DOM traversing

We can traverse the DOM using the `document` object that we saw in the previous chapter. This document object contains all the HTML and is a representation of the web page. Traversing over these elements can get you to the element you need in order to manipulate it.

This is not the most common way to do it, but this will help understand how it works later. And sometimes, you might actually find yourself needing these techniques as well. Just don't panic: there are other ways to do it, and they will be revealed in this chapter!

Even for a simple HTML piece there are already multiple ways to traverse the DOM. Let's go hunting for treasure in our DOM. We start with this little HTML snippet:

```html
<!DOCTYPE html>
<html>
  <body>
    <h1>Let's find the treasure</h1>
    <div id="forest">
      <div id="tree1">
        <div id="squirrel"></div>
        <div id="flower"></div>
      </div>
      <div id="tree2">
        <div id="shrubbery">
          <div id="treasure"></div>
        </div>
        <div id="mushroom">
          <div id="bug"></div>
        </div>
      </div>
    </div>
  </body>
</html>
```

We now want to traverse the DOM of this snippet to find the treasure. We can do this by stepping into the document object and navigating our way from there onwards. It is easiest to do this exercise in the console in the browser, because that way you'll get direct feedback about where in the DOM you are.

We can start by using the `body` property from the document. This contains everything that's inside the `body` element. In the console, we'll type:

```
console.dir(document.body);
```

We should get a really long object. There are a few ways from this object to get to our treasure. To do so, let's discuss the children and `childNodes` property.

> `childNodes` is more a complete term than children. Children just contain all the HTML elements, so are really the nodes. `childNodes` also contain text nodes and comments. With children, however, you can use the ID, and therefore they are easier to use.

To get to the treasure using children you would have to use:

```
console.dir(document.body.children.forest.children.tree2.children.
shrubbery.children.treasure);
```

As you can see, on every element we select, we have to select the children again. So, first, we grab the children from the body, then we select `forest` from these children. Then from `forest`, we want to grab its children again, and from these children we want to select `tree2`. From `tree2` we want to grab the children again, from these children we need `shrubbery`. And then finally, we can grab the children from `shrubbery` and select `treasure`.

To get to the treasure using `childNodes` you would have to use your console a lot because text and comment nodes are also in there. `childNodes` is an array, so you will have to select the right index to select the right child. There is one advantage here: it is a lot shorter because you won't need to select the name separately.

```
console.dir(document.body.childNodes[3].childNodes[3].childNodes[1].
childNodes[1]);
```

You could also combine them:

```
console.dir(document.body.childNodes[3].childNodes[3].childNodes[1].
children.treasure);
```

There are many ways to traverse the document. Depending on what you need, you might have to use one specific way. For tasks that require DOM traversing, it is usually the case that if it is works, it is a good solution.

So far, we have seen how we can move down the DOM, but we can also move up. Every element knows its parent. We can use the `parentElement` property to move back up. For example, if we use the treasure HTML sample and type this into the console:

```
document.body.children.forest.children.tree2.parentElement;
```

We are back at `forest`, since that is the parent element of `tree2`. This can be very useful, in particular when combined with functions such as `getElementById()`, which we will see later in more detail.

Not only can we move up and down, we can also move sideways. For example, if we select `tree2` like this:

```
document.body.children.forest.children.tree2;
```

We can get to `tree1` using:

```
document.body.children.forest.children.tree2.previousElementSibling;
```

And from `tree1` we can get to `tree2` using:

```
document.body.children.forest.children.tree1.nextElementSibling;
```

As an alternative to `nextElementSibling`, which returns the next node that is an element, you could use `nextSibling`, which will return the next node of any type.

Practice exercise 10.1

In this exercise, experiment with traversing the DOM hierarchy. You can use this sample HTML website:

```
<!doctype html>
<html><head><title>Sample Webpage</title></head>
<body>
    <div class="main">
        <div>
            <ul >
                <li>One</li>
                <li>Two</li>
```

```
                  <li>Three</li>
              </ul>
          </div>
          <div>blue</div>
          <div>green</div>
          <div>yellow</div>
          <div>Purple</div>
      </div>
  </body>
  </html>
```

Take the following steps:

1. Create and open the above sample web page, or visit your favorite website, and open the document body in the console with `console.dir(document)`.
2. In the `body.children` property, select some of the child elements. View how they match the page content.
3. Navigate to and output the next nodes or elements into the console.

Selecting elements as objects

Now we know how to traverse the DOM, we can make changes to the elements. Instead of using `console.dir()`, we can just type in the path to the element we want to change. We now have the element as a JavaScript object, and we can make changes to all its properties. Let's use a simpler HTML page for this one.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Welcome page</h1>
    <p id="greeting">
      Hi!
    </p>
  </body>
</html>
```

We can traverse to the `p` element, for example, by using this code:

```
document.body.children.greeting;
```

This gives us the power to manipulate the properties of the element, and the element itself, directly! Let's execute this newly gained power in the next section.

Changing innerText

The `innerText` property focuses on the text between the opening and closing of the element, like so:

```
<element>here</element>
```

The retrieved value would be `here` as plain text. For example, if we go to the console and we type:

```
document.body.children.greeting.innerText = "Bye!";
```

The message that is displayed on the page changes from `Hi!` to `Bye!` immediately. `innerText` returns the content of the element as plain text, which is not a problem in this case because there is only text in there. However, if there is any HTML inside the element you need to select, or if you want to add HTML, you cannot use this method. It will interpret the HTML as text and just output it on the screen. So if we executed this:

```
document.body.children.greeting.innerText = "<p>Bye!</p>";
```

It will output to the screen `<p>Bye!</p>`, with the HTML around it, as if it was intended as a text string. To get around this, you need to use `innerHTML`.

Changing innerHTML

If you did not only want to work with plain text, or perhaps specify some HTML formatting with your value, you could use the `innerHTML` property instead. This property doesn't just process be plain text, it can also be inner HTML elements:

```
document.body.children.greeting.innerHTML = "<b>Bye!</b>";
```

This will display **Bye!** in bold on the screen, having taken the `b` element into account rather than just printing it as if it were a single string value.

You were already promised that you could access elements in a more convenient way than traversing the DOM. Let's see how exactly in the next section.

Accessing elements in the DOM

There are multiple methods to select elements from the DOM. After getting the elements, we are able to modify them. In the following sections, we will discuss how to get elements by their ID, tag name, and class name, and by CSS selector.

Instead of traversing it step by step as we just did, we are going to use built-in methods that can go through the DOM and return the elements that match the specifications.

We are going to use the following HTML snippet as an example:

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Just an example</h1>
    <div id="one" class="example">Hi!</div>
    <div id="two" class="example">Hi!</div>
    <div id="three" class="something">Hi!</div>
  </body>
</html>
```

Let's start by accessing elements by ID.

Accessing elements by ID

We can grab elements by ID with the `getElementById()` method. This returns one element with the specified ID. IDs should be unique, as only one result will be returned from the HTML document. There are not a lot of rules for valid IDs; they cannot contain spaces and must be at least one character. As with the conventions for naming variables, it is a best practice to make it descriptive and avoid special characters.

If we want to select the element with an ID of `two` right away, we could use:

```
document.getElementById("two");
```

This would return the full HTML element:

```
<div id="two" class="example">Hi!</div>
```

To reiterate, if you have more than one element with the same ID, it will just give you back the first one it encounters. You should avoid this situation in your code though.

This is what the full file looks like with the JavaScript inside the HTML page, instead of simply querying the browser console:

```
<html>
  <body>
```

```
    <h1 style="color:pink;">Just an example</h1>
    <div id="one" class="example">Hi!</div>
    <div id="two" class="example">Hi!</div>
    <div id="three" class="something">Hi!</div>
  </body>
  <script>
    console.log(document.getElementById("two"));
  </script>
</html>
```

In this case, it would log the full HTML `div` with `id="two"` to the console.

Practice exercise 10.2

Try experimenting with getting elements by their IDs:

1. Create an HTML element and assign an ID in the element attribute.
2. Select the page element using its ID.
3. Output the selected page element into the console.

Accessing elements by tag name

If we ask for elements by tag name, we get an array as a result. This is because there could be more than one element with the same tag name. It will be a collection of HTML elements, or `HTMLCollection`, which is a special JavaScript object. It's basically just a list of nodes. Execute the following command in the console:

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

It will give back:

```
HTMLCollection(3) [div#one.example, div#two.example, div#three.
something, one: div#one.example, two: div#two.example, three:
div#three.something]
```

As you can see, all the elements in the DOM with the `div` tag are returned. You can read what the ID is and what the class is from the syntax. The first ones in the collection are the objects: `div` is the name, `#` specifies the ID, and `.` specifies the class. If there are multiple dots, there are multiple classes. Then you can see the elements again (`namedItems`), this time as key-value pairs with their ID as the key.

We can access them using the `item()` method to access them by index, like this:

```
document.getElementsByTagName("div").item(1);
```

This will return:

```
<div id="two" class="example">Hi!</div>
```

We can also access them by name, using the `namedItem()` method, like this:

```
document.getElementsByTagName("div").namedItem("one");
```

And this will return:

```
<div id="one" class="example">Hi!</div>
```

When there is only one match, it will still return an `HTMLCollection`. There is only one `h1` tag, so let's demonstrate this behavior:

```
document.getElementsByTagName("h1");
```

This will output:

```
HTMLCollection [h1]
```

Since `h1` doesn't have an ID or class, it is only `h1`. And since it doesn't have an ID, it is not a `namedItem` and is only in there once.

Practice exercise 10.3

Use JavaScript to select page elements via their tag name:

1. Start by creating a simple HTML file.
2. Create three HTML elements using the same tag.
3. Add some content within each element so you can distinguish between them
4. Add a script element to your HTML file, and within it select the page elements by tag name and store them in a variable as an array
5. Using the index value, select the middle element and output it into the console.

Accessing elements by class name

We can do something very similar for class names. In our example HTML, we have two different class names: `example` and `something`. If you get elements by class name, it gives back an HTMLCollection containing the results. The following will get all the elements with the class `example`:

```
document.getElementsByClassName("example");
```

This returns:

```
HTMLCollection(2) [div#one.example, div#two.example, one: div#one.
example, two: div#two.example]
```

As you can see, it only returned the `div` tags with the `example` class. It left out the `div` with the `something` class.

Practice exercise 10.4

Select all matching page elements using the class name of the element.

1. Create a simple HTML file to work on.
2. Add three HTML elements adding the same class to each. You can use different tags as long as the same element class is included. Add some content within each element so you can distinguish between them.
3. Add a script element to your file, and within it select the page elements by class name. Assign the resulting `HTMLCollection` values to a variable.
4. You can use an index value to select the individual `HTMLCollection` items, just as you would for array items. Starting with an index of 0, select one of the page elements with the class name and output the element into the console.

Accessing elements with a CSS selector

We can also access elements using a CSS selector. We do this with `querySelector()` and `querySelectorAll()`. We then give the CSS selector as an argument, and this will filter the items in the HTML document and only return the ones that satisfy the CSS selector.

The CSS selector might look a bit different than you might think at first. Instead of looking for a certain layout, we use the same syntax as we use when we want to specify a layout for certain elements. We haven't discussed this yet, so we will cover it here briefly.

If we state p as a CSS selector, it means all the elements with tag p. This would look like this:

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

If we say p.example, it means all the p tag elements with example as the class. They can also have other classes; as long as example is in there, it will match. We can also say #one, which means select all with an ID of one.

This method is the same result as getElementById(). Which one to use is a matter of taste when all you really need to do is select by ID—this is great input for a discussion with another developer. querySelector() allows for more complicated queries, and some developers will state that getElementById() is more readable. Others will claim that you might as well use querySelector() everywhere for consistency. It doesn't really matter at this point, but try to be consistent.

Don't worry too much about all these options for now; there are many, and you'll figure them out when you need them. This is how you can use the CSS selectors in JavaScript.

Using querySelector()

This first option will select the first element that matches the query. So, enter the following in the console, still using the HTML snippet introduced at the start of the section:

```
document.querySelector("div");
```

It should return:

```
<div id="one" class="example">Hi!</div>
```

It only returns the first div, because that's the first one it encounters. We could also ask for an element with the class .something. If you recall, we select classes using dot notation like this:

```
document.querySelector(".something");
```

This returns:

```
<div id="three" class="something">Hi!</div>
```

With this method, you can only use valid CSS selectors: elements, classes, and IDs.

Practice exercise 10.5

Use `querySelector()` to enable single element selection:

1. Create another simple HTML file.
2. Create four HTML elements adding the same class to each. They can be different tag names as long as they have the class within the element attribute.
3. Add some content within each element so you can distinguish between them.
4. Within a script element, use `querySelector()` to select the first occurrence of the elements with that class and store it in a variable. If there is more than one matching result in `querySelector()`, it will return the first one.
5. Output the element into the console.

Using querySelectorAll()

Sometimes it is not enough to return only the first instance, but you want to select all the elements that match the query. For example when you need to get all the input boxes and empty them. This can be done with `querySelectorAll()`:

```
document.querySelectorAll("div");
```

This returns:

```
NodeList(3) [div#one.example, div#two.example, div#three.something]
```

As you can see, it is of object type `NodeList`. It contains all the nodes that match the CSS selector. With the `item()` method we can get them by index, just as we did for the `HTMLCollection`.

Practice exercise 10.6

Use `querySelectorAll()` to select all matching elements in an HTML file:

1. Create an HTML file and add four HTML elements, adding the same class to each one.
2. Add some content within each element so you can distinguish between them.
3. Within a script element, use `QuerySelectorAll()` to select all the matching occurrences of the elements with that class and store them in a variable.
4. Output all the elements into the console, first as an array and then looping through them to output them one by one.

Element click handler

HTML elements can do something when they are clicked. This is because a JavaScript function can be connected to an HTML element. Here is one snippet in which the JavaScript function associated with the element is specified in the HTML:

```
<!DOCTYPE html>
<html>
  <body>
    <div id="one" onclick="alert('Ouch! Stop it!')">Don't click here!
    </div>
  </body>
</html>
```

Whenever the text in the `div` gets clicked, a pop up with the text `Ouch! Stop it!` opens. Here, the JavaScript is specified directly after `onclick`, but if there is JavaScript on the page, you can also refer to a function that's in that JavaScript like this:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function stop(){
        alert("Ouch! Stop it!");
      }
    </script>
    <div id="one" onclick="stop()">Don't click here!</div>
  </body>
</html>
```

This code is doing the exact same thing. As you can imagine, with bigger functions this would be a better practice. The HTML can also refer to scripts that get loaded into the page.

There is also a way to add a click handler using JavaScript. We select the HTML element we want to add the click handler to, and we specify the onclick property.

Here is a HTML snippet:

```
<!DOCTYPE html>
<html>
  <body>
```

```
    <div id="one">Don't click here!</div>
  </body>
</html>
```

This code is at the moment not doing anything if you click it. If we want to dynamically add a click handler to the `div` element, we can select it and specify the property via the console:

```
document.getElementById("one").onclick = function () {
alert("Auch! Stop!");
}
```

As it's been added in the console, this functionality will be gone when you refresh the page.

This and the DOM

The `this` keyword always has a relative meaning; it depends on the exact context it is in. In the DOM, the special `this` keyword refers to the element of the DOM it belongs to. If we specify an `onclick` to send `this` in as an argument, it will send in the element the `onclick` is in.

Here is a little HTML snippet with JavaScript in the `script` tag:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function reveal(el){
        console.log(el);
      }
    </script>
    <button onclick="reveal(this)">Click here!</button>
  </body>
</html>
```

And this is what it will log:

```
<button onclick="reveal(this)">Click here!</button>
```

As you can see, it is logging the element it is in, the `button` element.

We can access the parent of `this` with a function like this:

```
function reveal(el){
    console.log(el.parentElement);
}
```

In the above example, the body is the parent of the button. So if we click the button with the new function, it will output:

```
<body>
    <script>
      function reveal(el.parentElement){
        console.log(el);
      }
    </script>
    <button onclick="reveal(this)">Click here!</button>
  </body>
```

We could output any other property of the element the same way; for example, `console.log(el.innerText);` would print the inner text value as we saw in the *Changing innerText* section.

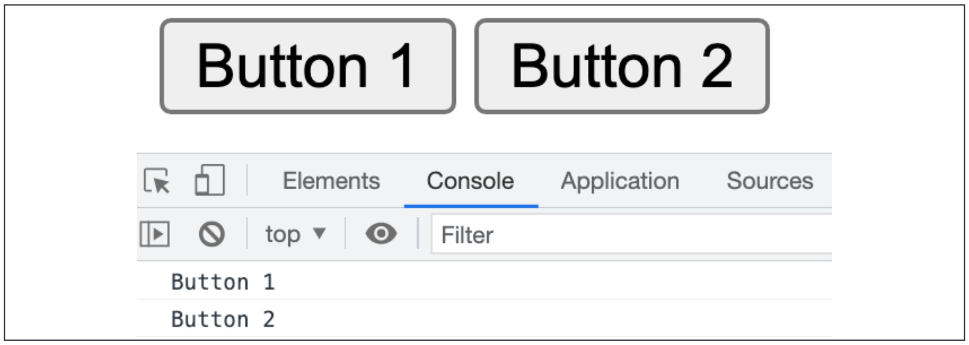
So, the `this` keyword is referring to the element, and from this element we can traverse the DOM like we just learned. This can be very useful, for example, when you need to get the value of an input box. If you send `this`, then you can read and modify the properties of the element that triggered the function.

Practice exercise 10.7

Create a button within a basic HTML document and add the `onclick` attribute. The example will demonstrate how you can reference object data with `this`:

1. Create a function to handle a click within your JavaScript code. You can name the function `message`.
2. Add this to the `onclick` function parameters sending the current element object data using `this`.
3. Within the `message` function, use `console.dir()` to output in the console the element object data that was sent to the function using `onclick` and `this`.
4. Add a second button to the page also invoking the same function on the click.

5. When the button is clicked, you should see the element that triggered the click in the console, like so:



```
Button 1
Button 2
```

Figure 10.1: Implementing the onclick attribute

Manipulating element style

After selecting the right element from the DOM, we can change the CSS style that applies to it. We can do this using the `style` property. This is how to do it:

1. Select the right element from the DOM.
2. Change the right property of the style property of this element.

We are going to make a button that will toggle the appearing and disappearing of a line of text. To hide something using CSS, we can set the `display` property of the element to `none`, like this for a `p` (paragraph) element:

```
p {
  display: none;
}
```

And we can toggle it back to visible using:

```
p {
  display: block;
}
```

We can add this style using JavaScript as well. Here is a little HTML and JavaScript snippet that will toggle the displaying of a piece of text:

```
<!DOCTYPE html>
<html>
```

```
    <body>
      <script>
        function toggleDisplay(){
          let p = document.getElementById("magic");
          if(p.style.display === "none") {
            p.style.display = "block";
          } else {
            p.style.display = "none";
          }
        }
      </script>
      <p id="magic">I might disappear and appear.</p>
      <button onclick="toggleDisplay()">Magic!</button>
    </body>
  </html>
```

As you can see, in the `if` statement we are checking for whether it is currently hiding, if it is hiding, we show it. Otherwise, we hide it. If you click the button and it is currently visible, it will disappear. If you click the button when the text is gone, it will appear.

You can do all sorts of fun things using this style element. What do you think this does when you click the button?

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function rainbowify(){
        let divs = document.getElementsByTagName("div");
        for(let i = 0; i < divs.length; i++) {
          divs[i].style.backgroundColor = divs[i].id;
        }
      }
    </script>
    <style>
      div {
        height: 30px;
        width: 30px;
        background-color: white;
      }
    </style>
```

```
      <div id="red"></div>
      <div id="orange"></div>
      <div id="yellow"></div>
      <div id="green"></div>
      <div id="blue"></div>
      <div id="indigo"></div>
      <div id="violet"></div>
      <button onclick="rainbowify()">Make me a rainbow</button>
    </body>
  </html>
```

This is what you see when you first open the page:

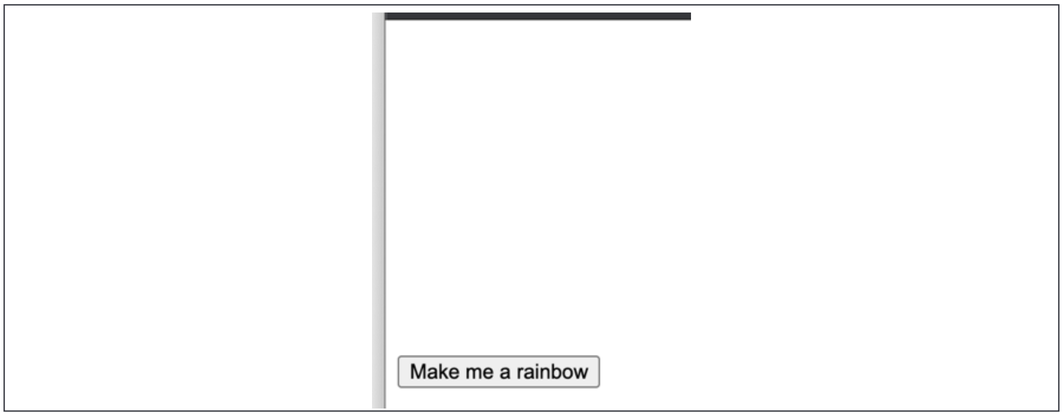


Figure 10.2: A button that will do wonderful things when it is clicked

And when you click the button:

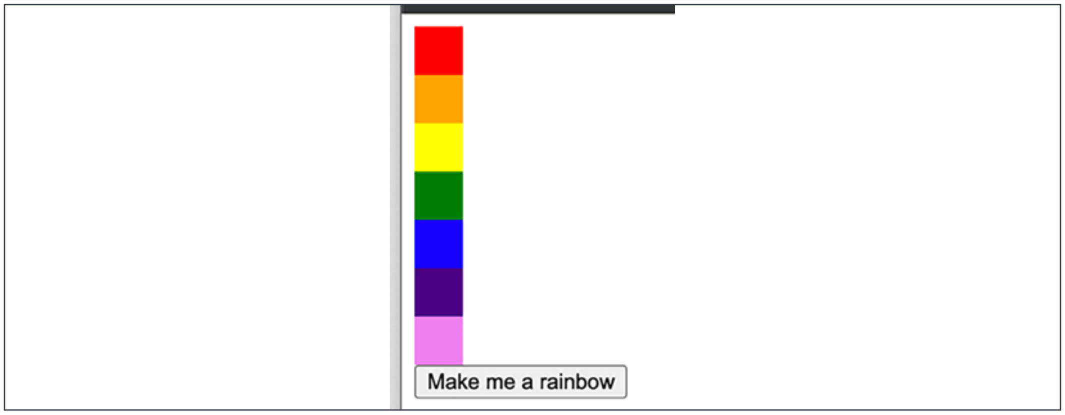


Figure 10.3: Beautiful rainbow made by JavaScript at the click of a button

Let's go over this script to see how works. First of all, there are a few div tags in the HTML that all have the ID of a certain color. There is a style tag specified in HTML, which gives a default layout to these div tags of 30px by 30px and a white background.

When you click the button, the rainbowify() JavaScript function is executed. In this function the following things are happening:

1. All the div elements get selected and stored in an array, divs.
2. We loop over this divs array.
3. For every element in the divs array, we are setting the backgroundColor property of style to the ID of the element. Since all the IDs represent a color, we see a rainbow appear.

As you can imagine, you can really have a lot of fun playing around with this. With just a few lines of code, you can make all sorts of things appear on the screen.

Changing the classes of an element

HTML elements can have classes, and as we have seen, we can select elements by the name of the class. As you may remember, classes are used a lot for giving elements a certain layout using CSS.

With JavaScript, we can change the classes of HTML elements, and this might trigger a certain layout that is associated with that class in CSS. We are going to have a look at adding classes, removing classes, and toggling classes.

Adding classes to elements

This might sound a bit vague, so let's have a look at an example where we are going to add a class to an element, which in this case will add a layout and make the element disappear.

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function disappear(){
        document.getElementById("shape").classList.add("hide");
      }
    </script>
    <style>
      .hide {
        display: none;
      }

      .square {
        height: 100px;
        width: 100px;
        background-color: yellow;
      }

      .square.blue {
        background-color: blue;
      }
    </style>
    <div id="shape" class="square blue"></div>

    <button onclick="disappear()">Disappear!</button>
  </body>
</html>
```

In this example, we have some CSS specified in the `style` tag. Elements with the `hide` class have a `display: none` style, meaning they are hidden. Elements with the `square` class are 100 by 100 pixels and are yellow. But when they have both the `square` and `blue` class, they are blue.

When we click on the **Disappear!** button, the `disappear()` function gets called. This one is specified in the script tag. The `disappear()` function changes the classes by getting the `classList` property of the element with the ID `shape`, which is the square we are seeing. We are adding the `hide` class to the `classList` and because of this, the elements get the `display: none` layout and we can no longer see it.

Removing classes from elements

We can also remove a class. If we remove the `hide` class from the `classList`, for example, we could see our element again because the `display: none` layout no longer applies.

In this example, we are removing another class. Can you figure out what will happen if you press the button by looking at the code?

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function change(){
        document.getElementById("shape").classList.remove("blue");
      }
    </script>
    <style>
      .square {
        height: 100px;
        width: 100px;
        background-color: yellow;
      }

      .square.blue {
        background-color: blue;
      }
    </style>
    <div id="shape" class="square blue"></div>

    <button onclick="change()">Change!</button>
  </body>
</html>
```

When the button gets pressed, the change function gets triggered. This function removes the `blue` class, which removes the blue background color from the layout, leaving us with the yellow background color and the square will turn yellow.

You may wonder why the square was blue in the first place since it had two layouts for `background-color` assigned to it with the CSS. This happens with a points system. When a styling is more specific, it gets more points. So, specifying two classes with no space in between means that it applies to elements with these two classes. This is more specific than pointing at one class.

Referring to an ID in CSS, `#nameId`, gets even more points and would be prioritized over class-based layouts. This layering allows for less duplicate code, but it can become messy, so always make sure to combine the CSS and the HTML well to get the desired layout.

Toggling classes

In some cases, you would want to add a class when it doesn't already have that particular class, but remove it when it does. This is called toggling. There is a special method to toggle classes. Let's change our first example to toggle the `hide` class so the class will appear when we press the button the second time, disappear the third time, and so on. The `blue` class was removed to make it shorter; it's not doing anything in this example other than making the square blue.

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function changeVisibility(){
        document.getElementById("shape").classList.toggle("hide");
      }
    </script>
    <style>
      .hide {
        display: none;
      }

      .square {
        height: 100px;
        width: 100px;
        background-color: yellow;
      }
    </style>
    <div id="shape" class="square"></div>

    <button onclick="changeVisibility()">Magic!</button>
  </body>
</html>
```

Pressing the `Magic!` button will add the class to the `classList` when it isn't there and remove it when it is there. This means that you can see the result every time you press the button. The square keeps appearing and disappearing.

Manipulating attributes

We have seen already that we can change the class and style attributes, but there is a more general method that can be used to change any attribute. Just a quick reminder, attributes are the parts in HTML elements that are followed by equals signs. For example, this HTML link to Google:

```
<a id="friend" class="fancy boxed" href="https://www.google.com">Ask my
friend here.</a>
```

The attributes in this example are `id`, `class`, and `href`. Other common attributes are `src` and `style`, but there are many others out there.

With the `setAttribute()` method, we can add or change attributes on an element. This will change the HTML of the page. If you inspect the HTML in the browser you will see that the changed attributes are visible. You can do this from the console and see the result easily, or write another HTML file with this built in as a function. In this HTML snippet, you will see it in action:

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function changeAttr(){
        let el = document.getElementById("shape");
        el.setAttribute("style", "background-color:red;border:1px solid
black");
        el.setAttribute("id", "new");
        el.setAttribute("class", "circle");

      }
    </script>
    <style>
      div {
        height: 100px;
        width: 100px;
        background-color: yellow;
      }
```

```
        .circle {
          border-radius: 50%;
        }
      </style>
      <div id="shape" class="square"></div>

      <button onclick="changeAttr()">Change attributes...</button>
    </body>
  </html>
```

This is the page before clicking the button:

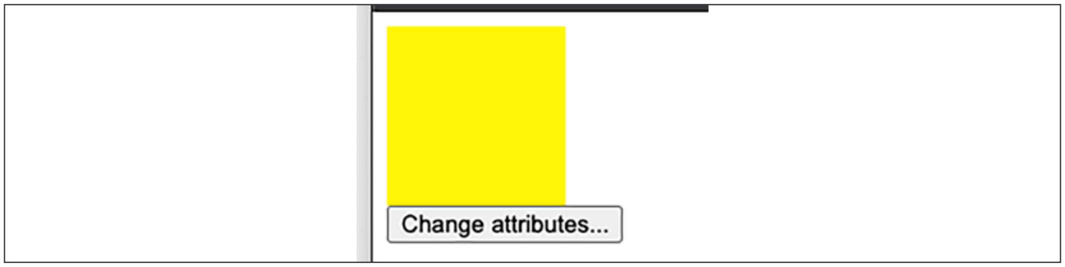


Figure 10.4: Page with a yellow square div

After clicking the button, the HTML of the `div` becomes:

```
  <div id="new" class="circle" style="background-color:red;border:1px
  solid black"></div>
```

As you can see, the attributes are changed. The `id` has changed from `shape` to `new`. The `class` has changed from `square` to `circle` and a `style` has been added. It will look like this:

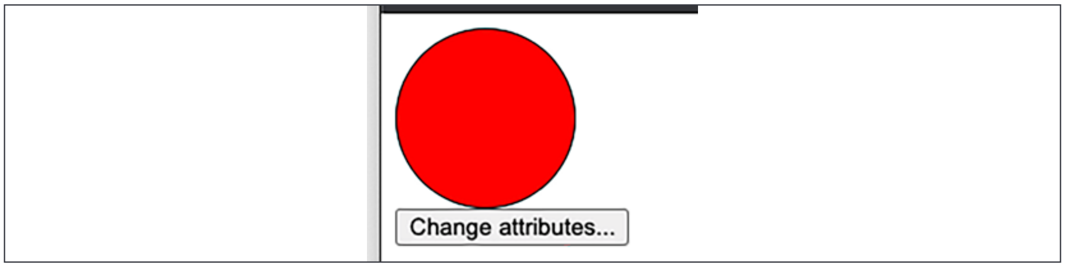


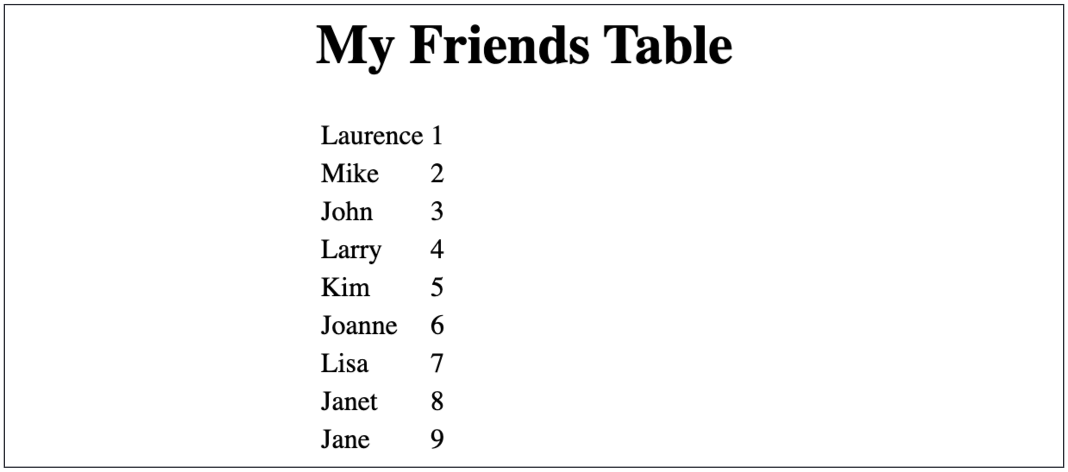
Figure 10.5: Page with a red circle with a black line around it

This is a very powerful tool that can be used to interact with the DOM in very many ways. Think, for example, of a tool that can be used to create images, or even postcards. Beneath the surface, there is a lot of manipulating going on.

It is important to note here that JavaScript interacts with the DOM and not with the HTML file—therefore, the DOM is the one that gets changed. If you click the button again, you'll get an error message in the console because no element with `id="shape"` is found in the DOM, and as a result we try to call a method on a null value.

Practice exercise 10.8

Creating custom attributes: using an array of names, the following code will update the element's HTML, adding HTML code using the data from the array. The items within the array will be output to the page as HTML code. The user will be able to click the page elements and they will display the page element attribute values.



My Friends Table

Laurence	1
Mike	2
John	3
Larry	4
Kim	5
Joanne	6
Lisa	7
Janet	8
Jane	9

Figure 10.6: Creating custom attributes with an array of names

As the HTML will start getting more complex from now on, and we're only trying to test your JavaScript, we will provide HTML templates to use where needed. You can use the following HTML template and provide your answer as the completed `script` element:

```
<!DOCTYPE html>
<html>
<head>
    <title>Complete JavaScript Course</title>
</head>
<body>
```

```
    <div id="message"></div>
    <div id="output"></div>
    <script>

    </script>
</body>
</html>
```

Take the following steps:

1. Create an array of names. You can add as many as you want—all the string values will be output onto the page within a table.
2. Select the page elements as JavaScript objects.
3. Add a function and also invoke that function within the JavaScript code. The function can be called `build()` as it will be building the page content. Within the `build` function, you will be setting up the HTML in a table.
4. Create a table named `html`, and within the tags, loop through the contents of the array and output the results into the `html` table.
5. Add a class called `box` to one of the cells that has the index value of the item from the array, adding the same class to the elements for each additional row.
6. As you create the HTML for the elements within the `tr` element, create an attribute called `data-row` in the main `row` element that includes the index value of the item from the array. In addition, add another attribute within the element called `data-name` that will contain the text output.
7. Within the attribute of the same `tr` element, also add `onclick` to invoke a function named `getData` passing the current element object as `this` into the function parameter.
8. Add the table of HTML code to the page.
9. Create a function named `getData` that will be invoked once the HTML `tr` elements are clicked. Once the `tr` element is clicked, use `getAttribute` to get the attribute values of the row value and the contents of the text output and store them in different variables.
10. Using the values in the attributes stored in the preceding step, output the values into the `message` element on the page.
11. Once the user clicks the element on the page, it will display the details coming from the element attributes within the element with the `id` of `message`.

Event listeners on elements

Events are things that happen on a web page, like clicking on something, moving the mouse over an element, changing an element, and there are many more. We have seen how to add an `onclick` event handler already. In the same way, you can add an `onchange` handler, or an `onmouseover` handler. There is one special condition, though; one element can only have one event handler as an HTML attribute. So, if it has an `onclick` handler, it cannot have an `onmouseover` handler as well. At this point, we have only seen how to add event listeners using HTML attributes like this:

```
<button onclick="addRandomNumber()">Add a number</button>
```

There is a way to register event handlers using JavaScript as well. We call these event listeners. Using event listeners, we can add multiple events to one element. This way, JavaScript is constantly checking, or listening, for certain events to the elements on the page. Adding event listeners is a two-step process:

1. Select the element you want to add an event to
2. Use the `addEventListener("event", function)` syntax to add the event

Even though it is two steps, it can be done in one line of code:

```
document.getElementById("square").addEventListener("click",
changeColor);
```

This is getting the element with the ID `square` and adding the `changeColor` function as the event for whenever it gets clicked. Note that when using event listeners, we remove the `on` prefix from the event type. For example, `click` here references the same event type as `onclick`, but we have removed the `on` prefix.

Let's consider another way to add an event listener (don't worry, we will review these methods in detail in *Chapter 11*, *Interactive Content and Event Listeners*) by setting the `event` property of a certain object to a function.

There is a fun fact here—event listeners often get added during other events!

We could reuse our trusty `onclick` listener in this context, but another common one is when the web page is done loading with `onload`:

```
window.onload = function() {
  // whatever needs to happen after loading
  // for example adding event listeners to elements
}
```

This function will then be executed. This is common for `window.onload`, but less common for many others, such as `onclick` on a `div` (it is possible though). Let's look at an example of the first event listener we looked at on a web page. Can you figure out what it will be doing when you click on the square?

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      window.onload = function() {
        document.getElementById("square").addEventListener("click",
changeColor);
      }
      function changeColor(){
        let red = Math.floor(Math.random() * 256);
        let green = Math.floor(Math.random() * 256);
        let blue = Math.floor(Math.random() * 256);
        this.style.backgroundColor = `rgb(${red}, ${green}, ${blue})`;
      }
    </script>
    <div id="square" style="width:100px;height:100px;background-
color:grey;">Click for magic</div>
  </body>
</html>
```

The web page starts with a gray square with the text `Click for magic` in it. After the web page is done loading, an event gets added for this square. Whenever it gets clicked, the `changeColor` function will be executed. This function uses random variables to change the color using RGB colors. Whenever you click the square, the color gets updated with random values.

You can add events to all sorts of elements. We have only used the `click` event so far, but there are many more. For example, `focus`, `blur`, `focusin`, `focusout`, `mouseout`, `mouseover`, `keydown`, `keypress`, and `keyup`. These will be covered in the next chapter, so keep going!

Practice exercise 10.9

Try an alternative way to implement similar logic to *Practice exercise 10.7*. Use the following HTML code as a template for this exercise, and add the contents of the `script` element:

```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
</head>
<body>
    <div>
        <button>Button 1</button>
        <button>Button 2</button>
        <button>Button 3</button>
    </div>
    <script>

    </script>
</body>
</html>
```

Take the following steps:

1. Select all the page buttons into a JavaScript object.
2. Loop through each button, and create a function within the button scope called output.
3. Within the `output()` function, add a `console.log()` method that outputs the current object's `textContent`. You can reference the current parent object using the `this` keyword.
4. As you loop through the buttons attach an event listener that when clicked invokes the `output()` function.

Creating new elements

In this chapter, you have seen so many cool ways to manipulate the DOM already. There is still an important one missing, the creation of new elements and adding them to the DOM. This consists of two steps, first creating new elements and second adding them to the DOM.

This is not as hard as it may seem. The following JavaScript does just that:

```
let el = document.createElement("p");
el.innerText = Math.floor(Math.random() * 100);
document.body.appendChild(el);
```

It creates an element of type `p` (paragraph). This is a `createElement()` function that is on the `document` object. Upon creation, you need to specify what type of HTML element you would want to create, which in this case is a `p`, so something like this:

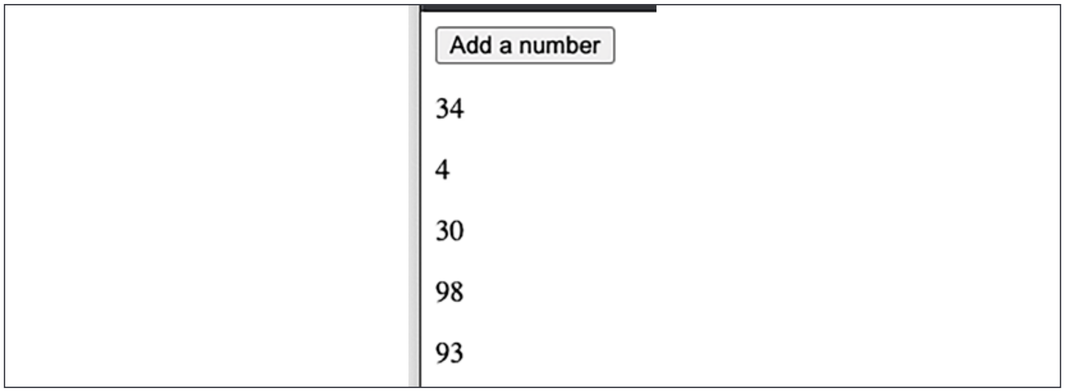
```
<p>innertext here</p>
```

And as `innerText`, it is adding a random number. Next, it is adding the element as a new last child of the body. You could also add it to another element; just select the element you want to add it to and use the `appendChild()` method.

Here, you can see it incorporated in a HTML page. This page has a button, and whenever it gets pressed, the `p` gets added.

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      function addRandomNumber(){
        let el = document.createElement("p");
        el.innerText = Math.floor(Math.random() * 100);
        document.body.appendChild(el);
      }
    </script>
    <button onclick="addRandomNumber()">Add a number</button>
  </body>
</html>
```

Here is a screenshot of this page after having pressed the button five times.



```
34

4

30

98

93
```

Figure 10.7: Random numbers after pressing the button five times

Once we refresh the page, it's empty again. The file with the source code doesn't change and we're not storing it anywhere.

Practice exercise 10.10

Shopping list: Using the following HTML template, update the code to add new items to the list of items on the page. Once the button is clicked, it will add a new item to the list of items:

```
<!DOCTYPE html>
<html>
<head>
    <title>Complete JavaScript Course</title>
    <style>
    </style>
</head>
<body>
    <div id="message">Complete JavaScript Course</div>
    <div>
        <input type="text" id="addItem">
        <input type="button" id="addNew" value="Add to List"> </div>
    <div id="output">
        <h1>Shopping List</h1>
        <ol id="sList"> </ol>
    </div>
    <script>

    </script>
</body>
</html>
```

Take the following steps:

1. Select the page elements as JavaScript objects.
2. Add an `onclick` event listener to the add button. Once the button is clicked, it should add the contents of the input field to the end of the list. You can call the function `addOne()`.
3. Within `addOne()`, create `li` elements to append to the main list on the page. Add the input value to the list item text content.
4. Within the `addOne()` function, get the current value of the `addItem` input field. Use that value to create a `textNode` with that value, adding it to the list item. Append the `textNode` to the list item.

Chapter projects

Collapsible accordion component

Build a collapsing and expanding accordion component that will open page elements, hiding and showing content when the title tab is clicked. Using the following HTML as a template, add the completed `script` element and create the desired functionality with JavaScript:

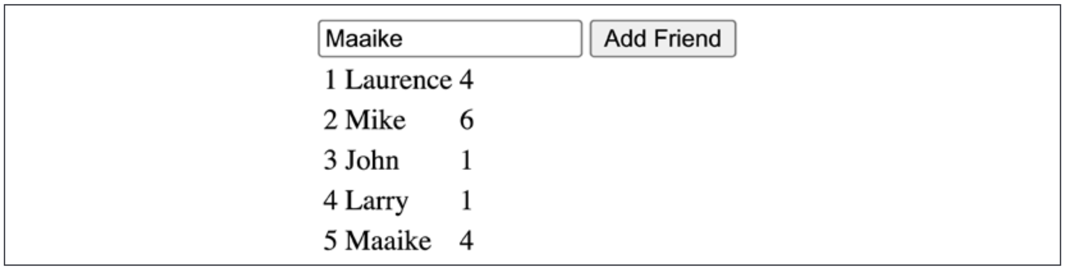
```
<!doctype html>
<html>
<head>
    <title>JS Tester</title>
    <style>
        .active {
            display: block !important;
        }
        .myText {
            display: none;
        }
        .title {
            font-size: 1.5em;
            background-color: #ddd;
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="title">Title #1</div>
        <div class="myText">Just some text #1</div>
        <div class="title">Title #2</div>
        <div class="myText">Just some text #2</div>
        <div class="title">Title #3</div>
        <div class="myText">Just some text #3</div>
    </div>
    <script>

    </script>
</body>
</html>
```

Take the following steps:

1. Using `querySelectorAll()`, select all the elements with a class of `title`.
2. Using `querySelectorAll()`, select all the elements with a class of `myText`. This should be the same number of elements as the `title` elements.
3. Iterate through all the `title` elements and add event listeners that, once clicked, will select the next element siblings.
4. Select the element on the `click` action and toggle the `classlist` of the element with the `class` of `active`. This will allow the user to click the element and hide and show the below content.
5. Add a function that will be invoked each time the elements are clicked that will remove the `class` of `active` from all the elements. This will hide all the elements with `myText`.

Interactive voting system

The below code will create a dynamic list of people that can be clicked, and it will update the corresponding value with the number of times that name was clicked. It also includes an input field that will allow you to add more users to the list, each of which will create another item in the list that can be interacted with the same as the default list items.



1	Laurence	4
2	Mike	6
3	John	1
4	Larry	1
5	Maaike	4

Figure 10.8: Creating an interactive voting system

Use the following HTML code as a template to add JavaScript to, and provide your answer as the completed `script` element.

```html
<!DOCTYPE html>
<html>
<head>
    <title>Complete JavaScript Course</title>
</head>
<body>
    <div id="message">Complete JavaScript Course</div>
    <div>
        <input type="text" id="addFriend">
        <input type="button" id="addNew" value="Add Friend">
    </div>
    <table id="output"></table>
    <script>

    </script>
</body>
</html>
```

Take the following steps:

1. Create an array of people's names called `myArray`. This will be the default original list of names.
2. Select the page elements as JavaScript objects so they can easily be selected within the code.
3. Add event listener to the **Add Friend** button. Once clicked, this will get the value from the input field and pass the values to a function that will add the friend list to the page. Additionally, add the new friend's name into the people's names array you created. Get the current value in the input field and push that value into the array so the array matches the values on the page.
4. Run a function to build the content on the page, using the `forEach()` loop get all the items within the array and add them to the page. Include `0` as a default for the vote count, as all individuals should start on zero votes.
5. Create a main function that will create the page elements, starting with the parent table row, `tr`. Then create three table cell, `td`, elements. Add content to the table cells, including the vote count in the last column, the person name in the middle, and the index plus 1 in the first column.
6. Append the table cells to the table row and append the table row to the output area on the page.
7. Add an event listener that will increase the vote counter for that row when the user clicks.
8. Get the text content from the last column in the row. It should be the value of the current counter. Increment the counter by one and make sure the datatype is a number so you can add to it.
9. Update the last column with the new click counter.

Hangman game

Create a Hangman game using arrays and page elements. You can use the following HTML template:

```
<!doctype html>
<html><head>
    <title>Hangman Game</title>
    <style>
        .gameArea {
            text-align: center;
            font-size: 2em;
        }
        .box,
        .boxD {
            display: inline-block;
            padding: 5px;
        }
        .boxE {
            display: inline-block;
            width: 40px;
            border: 1px solid #ccc;
            border-radius: 5px;
            font-size: 1.5em;
        }
    </style>
</head>
<body>
    <div class="gameArea">
        <div class="score"> </div>
        <div class="puzzle"></div>
        <div class="letters"></div>
        <button>Start Game</button>
    </div>
    <script>

    </script>
</body>
</html>
```

Take the following steps:

1. Set up an array that contains some words or phrases that you want to use in the game.
2. In JavaScript, create a main game object containing a property to contain the current word solution, and another property to contain the array of letters of the solution. It should also create an array to contain the page elements and correspond with the values of the index values of each letter from the solution, and finally add a property to count the number of letters left to solve and end the game when needed.
3. Select all the page elements into variables so they are easier to access in the code.

4. Add an event listener to the **Start Game** button, making it clickable, and when it gets clicked it should launch a function called `startGame()`.
5. Within `startGame()`, check if the `words` array has any words left. If it does, then hide the button by setting the `.display` object to `none`. Clear the game contents and set the total to `0`. Within the current word in the game object, assign a value, which should be the response of `shift()` from the array containing the in-game words.
6. In the game solution, convert the string into an array of all the characters in the word solution using `split()`.
7. Create a function called `builder()` that can be used to build the game board. Invoke the function within the `startGame()` function once all the game values are cleared and set.
8. Create a separate function that you can use to generate page elements. In the parameters, get the type of element, the parent that the new element will be appended to, the output content for the new element, and a class to add to the new element. Using a temporary variable, create the element, add the class, append to the parent, set the `textContent`, and return the element.
9. In the `builder()` function, which will also get invoked once `startGame()` is run, clear the `innerHTML` from the letters and puzzle page elements.
10. Iterate through the game solution array, getting each letter of the solution. Use the `builder()` function to generate page elements, add an output value of `-`, set a class, and append it to the main puzzle page element.
11. Check if the value is blank, and if it is, clear `textContent` and update the border to white. If it's not blank, increment the total so that it reflects the total number of letters that must be guessed. Push the new element into the game puzzle array.
12. Create a new function to update the score so that you can output the current number of letters left. Add it to the `builder()` function.
13. Create a loop to represent the 26 letters of the alphabet. You can generate the letter by using an array containing all the letters. The string method `fromCharCode()` will return the character from the numeric representation.
14. Create elements for each letter, adding a `class` of `box` and appending it to the `letters` page element. As each element gets created, add an event listener that runs a function called `checker()`.
15. Once the letter gets clicked, we need to invoke the `checker()` function, which will remove the main class, add another class, remove the event listener, and update the background color. Also invoke a function called `checkLetter()`, passing the value of the clicked letter into the argument.

16. The `checkLetter()` function will loop through all the solution letters. Add a condition to check if the solution letter is equal to the letter selected by the player. Make sure to convert the inputted letter to uppercase so that you can match the letters accurately. Update the matching letters in the puzzle using the game puzzle array and the index from the letter in the solution. The index values will be the same on each, which provides an easy way to match the visual representation with what is in the array.
17. Subtract one from the game global object that tracks the total letters left to be solved, invoke the `updatescore()` function to check if the game is over, and update the score. Set the `textContent` of the puzzle to the letter removing the original dash.
18. In the `updatescore()` function, set the score to the number of letters left. If the total left is less than or equal to zero, the game is over. Show the button so that the player has an option for the next phrase.

Self-check quiz

1. What output will the following code produce?

```
    <div id="output">Complete JavaScript Course </div>
     <script>
         var output = document.getElementById('output');
         output.innerText = "Hello <br> World";
     </script>
```

2. What output will be seen within the browser page?

```
     <div id="output">Complete JavaScript Course </div>
     <script>
         document.getElementById('output').innerHTML = "Hello
 <br> World";
     </script>
```

3. What will be seen in the input field from the following code?

```
     <div id="output">Hello World</div>
     <input type="text" id="val" value="JavaScript">
     <script>
         document.getElementById('val').value = document.
 getElementById('output').innerHTML;
     </script>
```

4. In the following code, what is output into the console when the element with the word `three` gets clicked? What is the output when the element with the word `one` gets clicked?

```
<div class="holder">
    <div onclick="output('three')">Three
        <div onclick="output('two')">Two
            <div onclick="output('one')">One</div>
            </div>
        </div>
    </div>
<script>
    function
    output(val) {
        console.log(val);
    }
</script>
```

5. What line of code needs to be added to remove the event listener when the button is clicked in the following code?

```
<div class="btn">Click Me</div>
<script>
    const btn = document.querySelector(".btn");
    btn.addEventListener("click", myFun);
    function myFun() {
        console.log("clicked");

    }
</script>
```

Summary

In this chapter, we really took our web skills to the next level. Manipulating the DOM allows all kinds of interactions with the web page, meaning that the web page is no longer a static event.

We started off by explaining the dynamic web and how to traverse the DOM. After having walked over the elements manually, we learned that there's an easier way to select elements in the DOM with the `getElementBy…()` and the `querySelector()` methods. After having selected them, we had the power to modify them, add new elements to them, and do all sorts of things using the elements we selected. We started with some more basic HTML handlers, and we could assign a function to, for example, the `onclick` attribute of the HTML element.

We also accessed the clicked element using the `this` argument that was sent in as a parameter, and we could modify it in different ways, for example, by changing the `style` property. We also saw how to add classes to an element, create new elements, and add them to the DOM. And finally, we worked with event listeners on elements that really took our dynamic web pages to the next level. With event listeners, we can specify more than one event handler for a certain element. All these new skills allow us to create amazing things in the web browser. You can actually create complete games now!

The next chapter will take your event handler skills to the next level and will enhance your ability to create interactive web pages even further (and make it a bit easier as well!).