

Introduction to JavaScript Training



with examples and
hands-on exercises

WEBUCATOR

Copyright © 2023 by Webucator. All rights reserved.

No part of this manual may be reproduced or used in any manner without written permission of the copyright owner.

Version: 3.3.0

The Authors

Nat Dunn

Nat Dunn is the founder of Webucator (www.webucator.com), a company that has provided training for tens of thousands of students from thousands of organizations. Nat started the company in 2003 to combine his passion for technical training with his business expertise, and to help companies benefit from both. His previous experience was in sales, business and technical training, and management. Nat has an MBA from Harvard Business School and a BA in International Relations from Pomona College.

Follow Nat on Twitter at [@natdunn](https://twitter.com/natdunn) and Webucator at [@webucator](https://twitter.com/webucator).

Brian Hoke

Brian Hoke joined Webucator as CEO in January, 2022. Brian Hoke was formerly Principal of Bentley Hoke, a web consultancy formed in Syracuse, New York, in 2000. The firm served the professional services, education, government, nonprofit, and retail sectors with a variety of development, design, and marketing services. Previously, Brian served as Director of Technology, Chair of the Computer and Information Science Department, and Dean of Students at Manlius Pebble Hill School, an independent day school in DeWitt, NY. Before that, Brian taught at Insitut auf dem Rosenberg, an international boarding school in St. Gallen, Switzerland. Brian holds degrees from Hamilton and Dartmouth colleges.

Class Files

Download the class files used in this manual at

<https://static.webucator.com/media/public/materials/classfiles/JSC101-3.3.0-introduction-to-javascript-training.zip>.

Errata

Corrections to errors in the manual can be found at <https://www.webucator.com/books/errata/>.

Table of Contents

LESSON 1. JavaScript Basics.....	1
JavaScript vs. EcmaScript.....	1
The HTML DOM.....	2
JavaScript Syntax.....	3
Accessing Elements.....	4
Where Is JavaScript Code Written?.....	5
JavaScript Objects, Methods and Properties.....	8
📄 Exercise 1: Alerts, Writing, and Changing Background Color.....	11
LESSON 2. Variables, Arrays, and Operators.....	15
JavaScript Variables.....	15
A Loosely Typed Language.....	16
Google Chrome DevTools.....	17
Storing User-Entered Data.....	21
📄 Exercise 2: Using Variables.....	25
Constants.....	26
Arrays.....	27
📄 Exercise 3: Working with Arrays.....	31
Associative Arrays.....	34
Playing with Array Methods.....	37
JavaScript Operators.....	38
The Modulus Operator.....	41
Playing with Operators.....	41
The Default Operator.....	44
📄 Exercise 4: Working with Operators.....	47
LESSON 3. JavaScript Functions.....	55
Global Objects and Functions.....	55
📄 Exercise 5: Working with Global Functions.....	58
User-defined Functions.....	64
📄 Exercise 6: Writing a JavaScript Function.....	68
Returning Values from Functions.....	74
LESSON 4. Built-In JavaScript Objects.....	75
String.....	75
Math.....	80
Date.....	83
Helper Functions.....	88
📄 Exercise 7: Returning the Day of the Week as a String.....	89

LESSON 5. Conditionals and Loops.....	93
Conditionals.....	93
Short-circuiting	98
Switch / Case.....	102
Ternary Operator.....	108
Truthy and Falsy.....	109
📄 Exercise 8: Conditional Processing.....	110
Loops.....	114
while and do...while Loops.....	114
for Loops.....	116
break and continue.....	118
📄 Exercise 9: Working with Loops.....	120
Array: forEach().....	123
LESSON 6. Event Handlers and Listeners.....	125
On-event Handlers.....	125
📄 Exercise 10: Using On-event Handlers.....	128
The addEventListener() Method.....	132
Anonymous Functions.....	138
Capturing Key Events.....	140
📄 Exercise 11: Adding Event Listeners.....	142
Benefits of Event Listeners.....	145
Timers.....	147
📄 Exercise 12: Typing Test.....	151

LESSON 7. The HTML Document Object Model.....	157
CSS Selectors.....	158
The innerHTML Property.....	162
Nodes, NodeLists, and HTMLCollections.....	163
Accessing Element Nodes.....	164
📄 Exercise 13: Accessing Elements	174
Dot Notation and Square Bracket Notation.....	176
Accessing Elements Hierarchically.....	180
📄 Exercise 14: Working with Hierarchical Elements	184
Accessing Attributes.....	189
Creating New Nodes.....	190
Focusing on a Field.....	191
Shopping List Application.....	192
📄 Exercise 15: Logging	194
📄 Exercise 16: Adding EventListeners	197
📄 Exercise 17: Adding Items to the List	201
📄 Exercise 18: Dynamically Adding Remove Buttons to the List Items	203
📄 Exercise 19: Removing List Items	205
📄 Exercise 20: Preventing Duplicates and Zero-length Product Names	207
Manipulating Tables.....	209
LESSON 8. CSS Object Model.....	217
Changing CSS with JavaScript.....	217
Hiding and Showing Elements.....	222
Checking and Changing Other Style Properties.....	226
Increasing and Decreasing Measurements.....	228
Custom data Attributes.....	232
Gotcha with fontWeight.....	237
Font Awesome.....	240
classList Property.....	242
📄 Exercise 21: Showing and Hiding Elements	243
LESSON 9. Errors and Exceptions.....	251
Runtime Errors.....	251
Globally Handled Errors.....	252
Structured Error Handling.....	253
📄 Exercise 22: Try/Catch	257

LESSON 1

JavaScript Basics

EVALUATION COPY: Not to be used in class.

Topics Covered

- ✓ The HTML DOM.
- ✓ JavaScript syntax rules.
- ✓ Inline JavaScript.
- ✓ JavaScript script blocks.
- ✓ Creating and linking to external JavaScript files.
- ✓ Working with JavaScript objects, methods, and properties.
- ✓ Referencing HTML elements.

Introduction

In this lesson, you will get comfortable with the basics of JavaScript.

EVALUATION COPY: Not to be used in class.



1.1. JavaScript vs. EcmaScript

We refer to the language you are learning as *JavaScript*, which is what it is usually called. However, *JavaScript* was invented by Netscape Communications and is now owned by Oracle Corporation¹. Microsoft calls its version of the language *JScript*. JavaScript and JScript are both implementations of *EcmaScript*, but everyone still refers to the language as JavaScript.

1. <https://en.wikipedia.org/wiki/JavaScript#Trademark>

❖ 1.1.1. What is ECMAScript?

ECMAScript, sometimes abbreviated as “ES”, is a scripting language specification maintained and trademarked by Ecma International (<https://www.ecma-international.org/mission/>), a Europe-based industry association dedicated to technology and communications standards. The specification for the most-recent standard version of ECMAScript can be found at:

<https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>

As we mentioned above, JavaScript – the scripting language you are learning here and whose code is run by the browsers you (or others) use to visit the pages you build – is an implementation of ECMAScript.

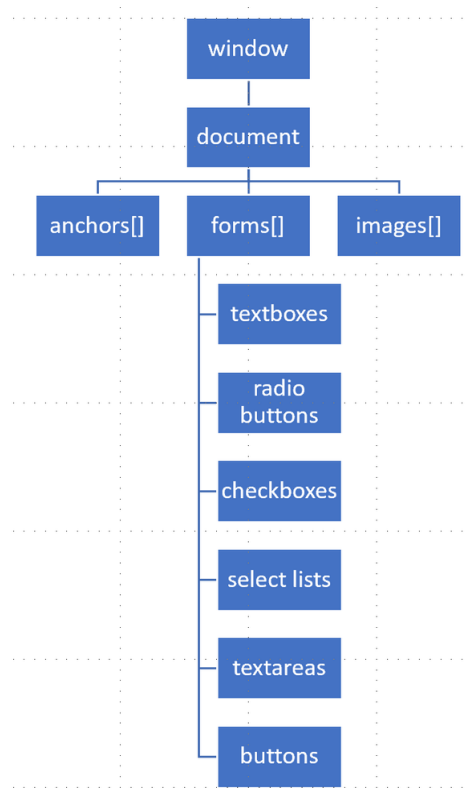
Keep in mind that ECMAScript evolves over time: new features are added, new syntax is adopted, etc. Like CSS, HTML, and other client-side technologies, JavaScript is an implementation of a standard (ECMAScript) by browsers - please be aware that all browsers won't implement (or implement in the same manner) all newer features of ECMAScript, and that later versions of browsers will implement newer features over time.

EVALUATION COPY: Not to be used in class.



1.2. The HTML DOM

The HTML Document Object Model (DOM) is the browser's view of an HTML page as an object hierarchy, starting with the browser window itself and moving deeper into the page, including all of the elements on the page and their attributes. Below is a simplified version of the HTML DOM:



As shown, the top-level object is window. The document object is a child of window and all the objects (i.e., elements) that appear on the page (e.g., forms, links, images, tables, etc.) are descendants of the document object. These objects can have children of their own. For example, form objects generally have several child objects, including text boxes, radio buttons, and select menus.

EVALUATION COPY: Not to be used in class.



1.3. JavaScript Syntax

❖ 1.3.1. Basic Rules

1. JavaScript statements end with semi-colons.
2. JavaScript is case sensitive.
3. JavaScript has two forms of comments:

- Single-line comments begin with a double slash (//).
- Multi-line comments begin with “/*” and end with “*/”.

```
// This is a single-line comment.
```

```
/*  
  This is  
  a multi-line  
  comment.  
*/
```

EVALUATION COPY: Not to be used in class.



1.4. Accessing Elements

❖ 1.4.1. Dot Notation

**Evaluation
Copy**

In JavaScript, elements (and other objects) can be referenced using dot notation, starting with the highest-level object (i.e., window). Objects can be referred to by name or id or by their position on the page. For example, if there is a form on the page named “loginform”, using dot notation you could refer to the form as follows:

```
window.document.loginform
```

Assuming that loginform is the first form on the page, you could also refer to it in this way:

```
window.document.forms[0]
```

A document can have multiple form elements as children. The number in the square brackets ([]) indicates the specific form in question. In programming speak, every document object contains a *collection* of forms. The length of the collection could be zero (meaning there are no forms on the page) or greater. In JavaScript, collections (and arrays) are zero-based, meaning that the first form on the page is referenced with the number zero (0) as shown in the syntax example above.

❖ 1.4.2. Square Bracket Notation

Objects can also be referenced using square bracket notation as shown below:

```
window['document']['loginform']  
  
// and  
  
window['document']['forms'][0]
```

Dot notation and square bracket notation are completely interchangeable. Dot notation is much more common; however, as we will see later in the course, there are times when it is more convenient to use square bracket notation.

EVALUATION COPY: Not to be used in class.



1.5. Where Is JavaScript Code Written?

JavaScript code can be written inline (e.g., within HTML attributes called *on-event handlers*), in script blocks, and in external JavaScript files. The page below shows examples of all three.

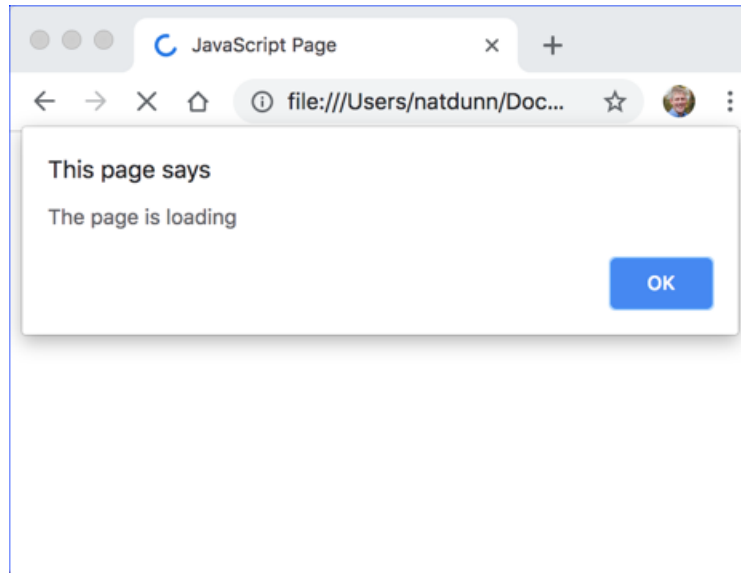
Demo 1.1: JavaScriptBasics/Demos/javascript.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      //Pop up an alert
10.     window.alert("The page is loading");
11. </script>
12. <title>JavaScript Page</title>
13. </head>
14. <body>
15. <main>
16.     <button onclick="document.body.style.backgroundColor = 'red';">
17.         Red
18.     </button>
19.     <button onclick="document.body.style.backgroundColor = 'white';">
20.         White
21.     </button>
22.     <button onclick="document.body.style.backgroundColor = 'green';">
23.         Green
24.     </button>
25.     <button onclick="document.body.style.backgroundColor = 'blue';">
26.         Blue
27.     </button>
28.     <script src="script.js"></script>
29. </main>
30. </body>
31. </html>
```

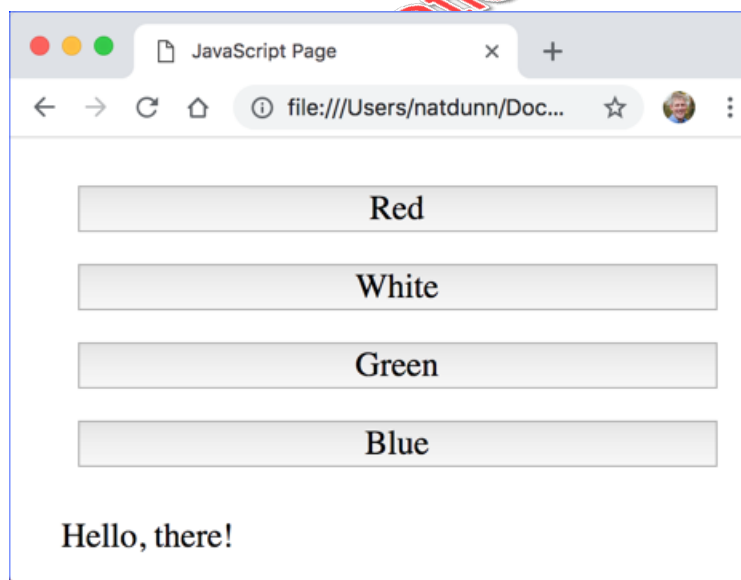
Demo 1.2: JavaScriptBasics/Demos/script.js

```
1.  /*
2.  This script simply outputs
3.  "Hello, there!"
4.  to the browser.
5.  */
6.  document.write("<p>Hello, there!</p>");
```

1. Open JavaScriptBasics/Demos/javascript.html in your browser. As the page loads, an alert will pop up that says “The page is loading” as shown below:

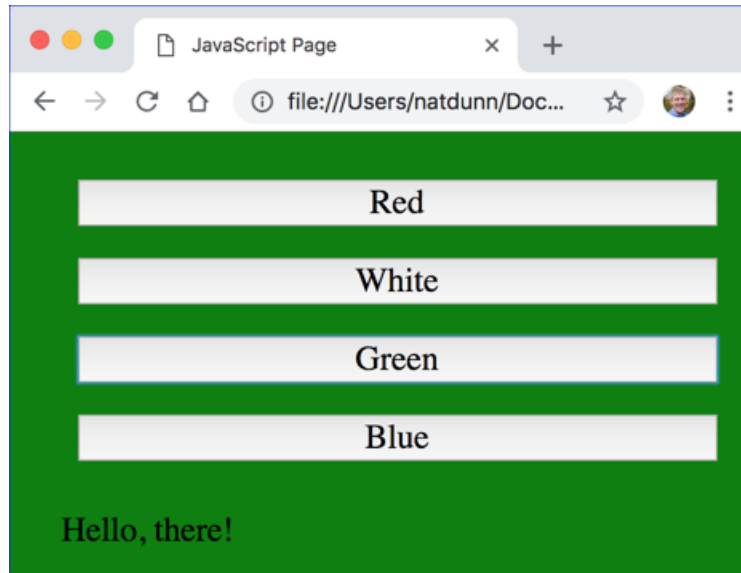


2. Click the **OK** button. The page will finish loading and will appear as follows:



The text "Hello, there!" is written dynamically by the code in `JavaScriptBasics/Demos/script.js`.

3. Click any one of the buttons. The background color of the page changes:



We will look at the code in this file and in `JavaScriptBasics/Demos/javascript.html` again shortly.

The Implicit window Object

The window object is always the implicit top-level object and therefore does not have to be included in references to objects. For example, `window.document.write()` can be shortened to `document.write()`. Likewise, `window.alert()` can be shortened to just `alert()`.

EVALUATION COPY: Not to be used in class.



1.6. JavaScript Objects, Methods and Properties

JavaScript is used to manipulate or get information about objects in the HTML DOM. Objects in an HTML page have methods (actions, such as opening a new window or submitting a form) and properties (attributes or qualities, such as color and size).

To illustrate objects, methods and properties, let's return to the code in `JavaScriptBasics/Demos/javascript.html` and `JavaScriptBasics/Demos/script.js`. You may find it useful to have those files open in your editor while reading this section.

❖ 1.6.1. Methods

Methods are the verbs of JavaScript. They cause things to happen.

`window.alert()`

HTML pages are read and processed from top to bottom. The JavaScript code in the initial `script` block at the top of `JavaScriptBasics/Demos/javascript.html` calls the `alert()` method of the `window` object. When the browser reads that line of code, it will pop up an alert box and will not continue processing the page until the user presses the OK button. Once the user presses the button, the alert box disappears and the rest of the page loads.

Note that, because `window` is the implicit top-level object, we could leave it off and just write `alert("The page is loading")`. And, in fact, this is the way it is usually done.

`document.write()`

The `write()` method of the `document` object is used to write out code to the page as it loads. In `JavaScriptBasics/Demos/script.js`, it simply writes out "Hello, there!"; however, it is more often used to write out dynamic data, such as the date and time on the user's machine.

The `document` object is a child of `window`, so we could write `window.document.write('some text')`, but again, `window` is implicit.

Arguments

Methods can take zero or more arguments separated by commas.

```
object.method(argument1, argument2);
```

The `alert()` and `write()` methods shown in the example above each take only one argument: the message to show or the HTML to write out to the browser.

❖ 1.6.2. Properties

Properties are the adjectives of JavaScript. They describe qualities of objects and, in some cases are writable (can be changed dynamically).

`document.body.style.backgroundColor`

The `body` object is a property of the `document` object, the `style` object is a property of the `body` object, and `backgroundColor` is a read-write property of the `style` object. To understand what's going on, it can be useful to read the dot notation from right to left: "The `backgroundColor` style of the `body` of the `document`."

Looking back at `JavaScriptBasics/Demos/javascript.html`, the four `button` elements use the `onclick` on-event handler to catch click events. When the user clicks a button, JavaScript is used to set the background of the `body` to a new color, in the same way that we might use CSS to style the page with `background-color:red` or `background-color:white`.



Exercise 1: Alerts, Writing, and Changing Background Color

⌚ 5 to 15 minutes

In this exercise, you will practice using JavaScript to pop up an alert, write text to the screen, and set the background color of the page.

1. Open `JavaScriptBasics/Exercises/alert-write-bgcolor.html` for editing.
2. In the head of the file, add a JavaScript alert which pops up the message “Welcome to my page!” when the page loads.
3. Add click handlers to the two buttons to allow the user to change the background color of the page to red or to blue.
4. In the script at the bottom of the page, use JavaScript to write the text “This text was generated by JavaScript.” to the page.
5. Test your solution in a browser.

Solution: JavaScriptBasics/Solutions/alert-write-bgcolor.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      window.alert("Welcome to my page!");
10. </script>
11. <title>Alert, Write, Change Background Color</title>
12. </head>
13. <body>
14. <main>
15.     <p>Click the button to turn the page:</p>
16.     <button onclick="document.body.style.backgroundColor = 'red'">
17.         Red
18.     </button>
19.     <p>Click the button to turn the page:</p>
20.     <button onclick="document.body.style.backgroundColor = 'blue'">
21.         Blue
22.     </button>
23.     <script>
24.         document.write('This text was generated by JavaScript');
25.     </script>
26. </main>
27. </body>
28. </html>
```

Code Explanation

1. In the head, we use `window.alert()` to generate the pop-up. We could have just used `alert()`.
 2. We use `document.write()` to write to the screen at the bottom of the page.
 3. We use `onclick="document.body.style.backgroundColor = 'red'"` and `onclick="document.body.style.backgroundColor = 'blue'"` to add click handlers to the buttons.
-

Conclusion

In this lesson, you have learned the basics of JavaScript. Now you're ready for more.

Evaluation
Copy

LESSON 2

Variables, Arrays, and Operators

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☑ Creating, reading, and modifying variables in JavaScript.
- ☑ JavaScript arrays.
- ☑ JavaScript operators.

Introduction

In this lesson, you will learn to work with variables, arrays, and operators.

EVALUATION COPY: Not to be used in class.



2.1. JavaScript Variables

Variables are used to hold data in memory. JavaScript variables are declared with the `let` keyword.

```
let age;
```

While this practice is discouraged, it is possible to declare multiple variables in a single step, like this:

```
let age, height, weight, dominantHand;
```

After a variable is declared, it can be assigned a value.

```
age = 18;
```

Variable declaration and assignment can be done in a single step.

```
let age = 18;
```

let versus var

If you have worked with JavaScript before, you may wonder why we are using `let` as opposed to the `var` keyword. Although `var` has not been officially deprecated, use of this keyword is discouraged primarily because variables defined with `let` cannot be accessed outside of the block where the variable is defined, thus reducing the likelihood of runtime errors caused by changing the value of a variable out of scope.² See the Mozilla documentation³ for details.

EVALUATION COPY: Not to be used in class.



2.2. A Loosely Typed Language

JavaScript is a loosely typed language. This means that you do not specify the data type of a variable when declaring it. It also means that a single variable can hold different data types at different times and that JavaScript can change the variable type on the fly.

For example, in the following block, the variable `age` is an *integer* and the variable `strAge` is a *string* (programming speak for text) because of the quotes.

```
let age = 18;  
let strAge = "18";
```

If you were to try to do a math function on `strAge` (e.g., multiply it by 4), a strongly typed (or *statically* typed) language would error saying you cannot multiply a string by a number. JavaScript would

2. You will learn more about scope when we cover functions.

3. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>

dynamically change `strAge` to an integer for the purposes of that operation. Although this is very convenient, it can also cause unexpected results, so be careful.

TypeScript

TypeScript⁴ is an open-source programming language developed by Microsoft. Developers writing in TypeScript compile their code to valid JavaScript, which they can use anywhere one might use JavaScript. A useful feature of TypeScript is **static typing**, meaning that a developer might specify the type of a given variable - to be a string, say, or a Boolean true/false variable - when declaring the variable. Violations of this static typing - trying to work with a number value as if it were a string value, for example - produces an error when compiling the TypeScript code into JavaScript, and thus adds a check against a dangerous bug creeping into the code.

EVALUATION COPY: Not to be used in class.



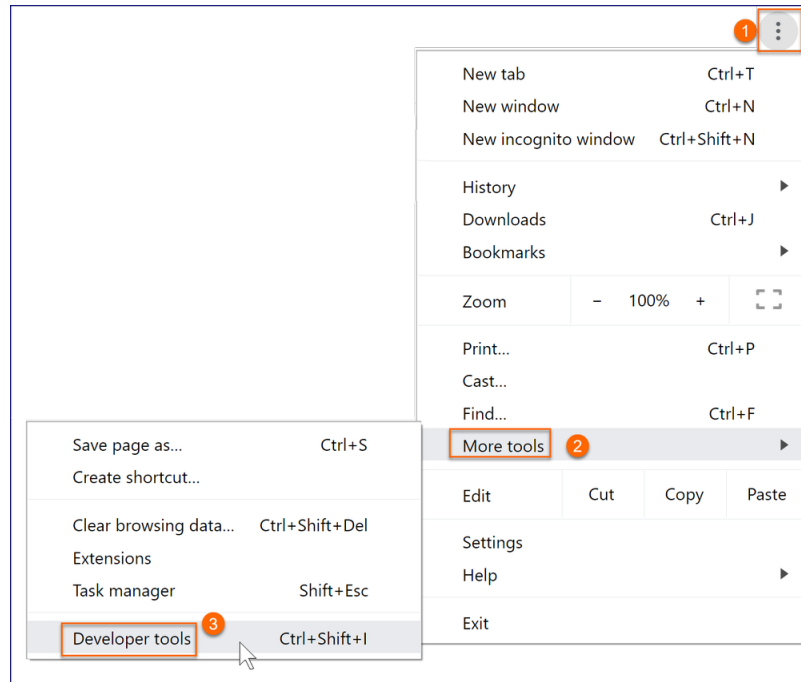
2.3. Google Chrome DevTools

Google Chrome DevTools is a set of tools to help web developers. We will use the Chrome DevTools Console to illustrate JavaScript's dynamic typing.

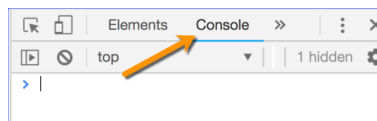
To open the Chrome DevTools Console:

1. Click the three-vertical-dot icon in the upper right of Google Chrome.
2. Select **More Tools**.
3. Select **Developer Tools**.

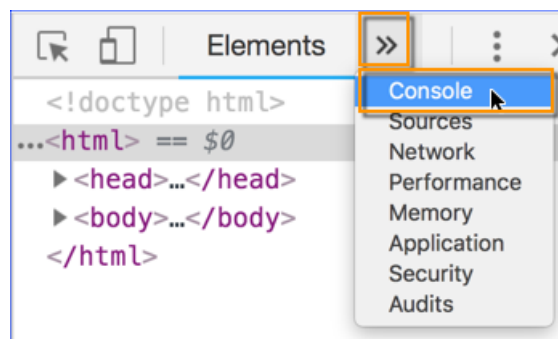
4. <https://www.typescriptlang.org/>



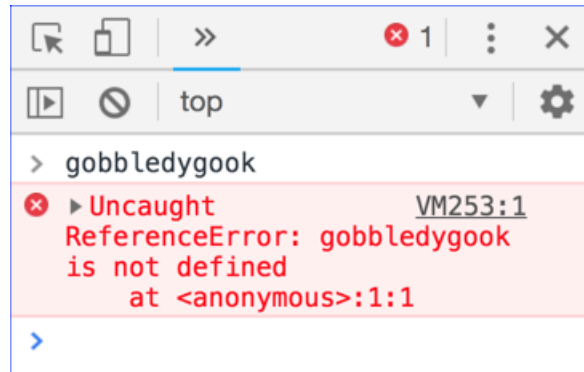
4. The tools will usually be docked on the right or bottom of your screen. Make sure that the Console is selected:



You may need to dropdown the menu to see the Console option:

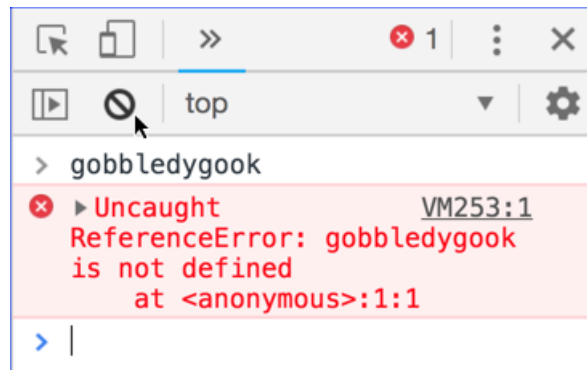


5. Now type “gobbledygook” in the Console and press **Enter**:

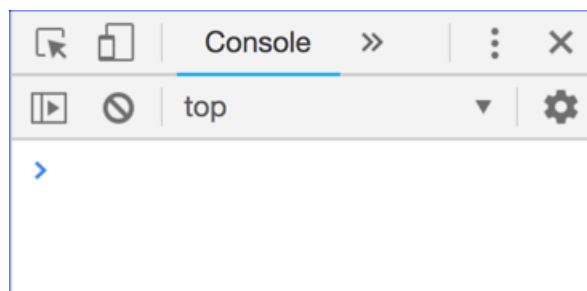


The word “gobbledygook” doesn’t mean anything in JavaScript and we have not defined a variable named “gobbledygook”, so we get an error.

6. To clear the Console, press the Clear Console icon:



7. You should now have a clear Console to start practicing JavaScript:



You can write and test JavaScript for a page directly in the Console. We will use it to show how JavaScript variables are dynamic:

1. Type `let age = 18;` and press **Enter** :

```
> let age = 18;
< undefined
> |
```

Don't worry about the "undefined" response. All that means is that your code didn't return anything.

2. Now type `age`; and press **Enter** :

```
> let age = 18;
< undefined
> age;
< 18
> |
```

This time it does return something – the value of `age`.

3. Let's subtract 2 from `age` and then add 2 to `age` :

```
> age - 2;
< 16
> age + 2;
< 20
```

That works as expected.

4. Now we will set `age` to `'18'` in single quotes. This makes `age` a string, which is programming-speak for text :

```
> age = '18';
< "18"
```

Notice that it returns `"18"`. At this point, a strongly typed programming language would have balked. It would have told us that `age` was declared as a number and cannot be assigned a string value. You may also notice that `"18"` in double quotes was returned even though we used single quotes when we set the value of `age`. Single and double quotes are interchangeable in JavaScript.

5. Now let's subtract 2 from `age` :

```
> age - 2;  
< 16
```

Notice that JavaScript understands that we want to treat `age` as a number and so it converts it to a number before doing the math.

6. Now let's add 2 to `age` :

```
> age + 2;  
< "182"
```

Oops! What happened? As it turns out, the plus operator (+) has multiple functions in JavaScript. In addition to adding numbers together, it can add strings together. In this case, because `age` is a string, it converts 2 to a string before doing the operation. So, it's adding "18" and "2" to give us "182".

The issue shown above does not come up often, but when it does, it can bite you. The best way to handle it is to make sure that when you are going to use a variable as a new type, you explicitly convert it to the new type. We will show how to do that later in the course.

❖ 2.3.1. Variable Naming

1. Variable names must begin with a letter, underscore (`_`), or dollar sign (`$`).
2. Variable names cannot contain spaces or special characters (other than the underscore and dollar sign).
3. Variable names can contain numbers (but not as the first character).
4. Variable names are case sensitive.
5. You cannot use keywords (e.g., `window` or `function`) as variable names.

EVALUATION COPY: Not to be used in class.



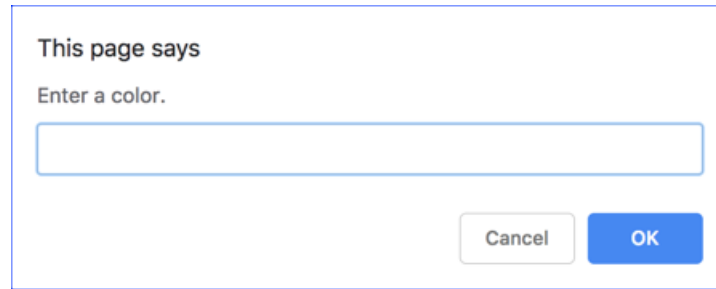
2.4. Storing User-Entered Data

The following example uses the `prompt()` method of the `window` object to collect user input. The value entered by the user is then assigned to a variable, which is accessed when the user clicks one of the `button` elements.

Demo 2.1: VariablesArraysOperators/Demos/variables.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      //Pop up a prompt
10.     let userColor = window.prompt("Enter a color.", "");
11. </script>
12. <title>JavaScript Variables</title>
13. </head>
14. <body>
15. <main>
16.     <button onclick="document.body.style.backgroundColor = 'red';">
17.         Red
18.     </button>
19.     <button onclick="document.body.style.backgroundColor = 'white';">
20.         White
21.     </button>
22.     <button onclick="document.body.style.backgroundColor = 'green';">
23.         Green
24.     </button>
25.     <button onclick="document.body.style.backgroundColor = 'blue';">
26.         Blue
27.     </button>
28.     <button onclick="document.body.style.backgroundColor = userColor;">
29.     <script>
30.         document.write(userColor);
31.     </script>
32.     </button>
33. </main>
34. </body>
35. </html>
```

As the page loads, a prompt pops up asking the user to enter a color.



This is done with the `prompt()` method of the window object. The `prompt()` method is used to get input from the user. It takes two arguments:

1. The message in the dialog box (e.g., "Enter a color.").
2. The default value that appears in the text box. In the example above this is an empty string (i.e., "").

If the **OK** button is pressed, the prompt returns the value entered in the text box. If the **Cancel** button, the prompt returns `null`.⁵ The line below assigns whatever is returned to the variable `userColor`.

```
let userColor = window.prompt("Enter a color.", "");
```

A script block with a call to `document.write()` is then used to output the color entered by the user. This output is contained within a `button` element, which has an `onclick` on-event handler that will be used to turn the background color of the page to the user-entered color.

```
<button  
  onclick="document.body.style.backgroundColor = userColor;">  
  <script>  
    document.write(userColor);  
  </script>  
</button>
```

Test this out:

1. Open `VariablesArraysOperators/Demos/variables.html` in your browser and enter "Yellow" in the prompt:

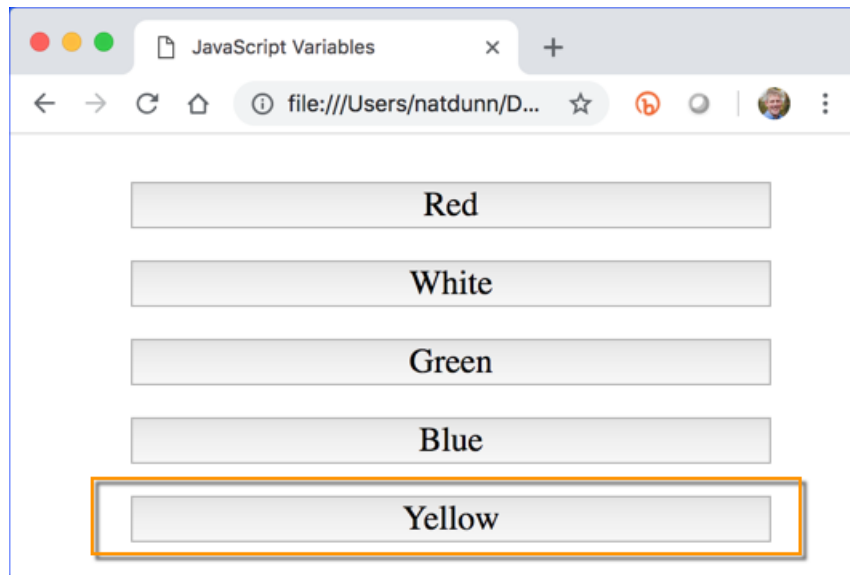
5. In JavaScript, `null` is a datatype with only one value: `null`. It represents a value that we don't know or that is missing.

This page says

Enter a color.


Cancel OK

- The resulting page should appear as follows:



- Click the "Yellow" button. The background should turn yellow.

Exercise 2: Using Variables

 5 to 15 minutes

In this exercise, you will practice using variables.

1. Open `VariablesArraysOperators/Exercises/variables.html` for editing.
2. Below the `ADD PROMPT HERE` comment, write code that will prompt the user for their first name and assign the result to a variable.
3. Add a button below the Ringo button that reads “Your Name”. Add functionality so that when this button is pressed an alert pops up showing the user’s first name.
4. Test your solution in a browser.

Exercise Code 2.1: `VariablesArraysOperators/Exercises/variables.html`

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     //ADD PROMPT HERE
10. </script>
11. <title>JavaScript Variables</title>
12. </head>
13. <body>
14. <main>
15.     <button onclick="alert('Paul');">Paul</button>
16.     <button onclick="alert('John');">John</button>
17.     <button onclick="alert('George');">George</button>
18.     <button onclick="alert('Ringo');">Ringo</button>
19.     <!--ADD BUTTON HERE-->
20. </main>
21. </body>
22. </html>
```

Solution: VariablesArraysOperators/Solutions/variables.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     let firstName = window.prompt("What's your name?", "");
10. </script>
11. <title>JavaScript Variables</title>
12. </head>
13. <body>
14. <main>
15.     <button onclick="alert('Paul');">Paul</button>
16.     <button onclick="alert('John');">John</button>
17.     <button onclick="alert('George');">George</button>
18.     <button onclick="alert('Ringo');">Ringo</button>
19.     <button onclick="alert(firstName);">Your Name</button>
20. </main>
21. </body>
22. </html>
```

EVALUATION COPY: Not to be used in class.



2.5. Constants

In programming, a constant is like a variable in that it is an identifier that holds a value, but, unlike variables, constants are not variable, they are constant. Good name choices, right?

Whereas variables are declared with the `let` keyword, constants are declared with the `const` keyword:

```
const NUM = 1;
```

Constants cannot be reassigned; that is, a later statement like `NUM = 2;` would fail, meaning that the value of `NUM` would remain 1; depending on how the browser you are using handles `const`, the later

statement may either cause the code to fail or simply not assign the new value to `NUM`. In Google Chrome, for example, trying to assign a new value to a constant will cause an error. We can see this using the Chrome DevTools Console:

```
> const NUM = 1;
< undefined
> NUM = 2;
✖ ▶ Uncaught TypeError: VM395:1
  Assignment to constant
  variable.
    at <anonymous>:1:5
```

While constants can be declared with uppercase or lowercase names, the convention is to use all-uppercase names for constants in the global scope⁶, so they are easily distinguishable from variables. Constants in the function scope are named using lowerCamelCase, just like variables.

Constants in this Course

In this course, we often write small bits of code in the global scope (i.e., not within curly braces) that would normally be locally scoped in real code. In these cases, we use lowerCamelCase for our constant names.

EVALUATION COPY: Not to be used in class.



2.6. Arrays

An array is a grouping of objects that can be accessed through subscripts. At its simplest, an array can be thought of as a list. In JavaScript, the first element of an array is considered to be at position zero (0), the second element at position one (1), and so on. Arrays are useful for storing related sets of data.⁷

Arrays are declared using the `new` keyword and should be defined as constant:

```
const myArray = new Array();
```

6. You will learn more about scope when we cover functions.

7. Unlike in some languages, values in JavaScript arrays do not all have to be of the same data type.

It is also possible and very common to use the `[]` literal to declare a new Array object:

```
const myArray = [];
```

When constants are not constant

When you declare a constant, you create a pointer to a specific object. You may not change the pointer (i.e., you cannot assign a new value to a constant), but you can change the object that is assigned to the constant (e.g., the items in the array).

Values are assigned to arrays as follows:

```
myArray[0] = value1;  
myArray[1] = value2;  
myArray[2] = value3;
```

Evaluation
Copy

Arrays can be declared with initial values.

```
const myArray = new Array(value1, value2, value3);
```

Or, using the `[]` notation:

```
const myArray = [value1, value2, value3];
```

The following example is similar to the previous one, except that it prompts the user for four different colors and places each into the `colors` array. It then displays the values in the `colors` array in the buttons and assigns them to `document.body.style.backgroundColor` when the user clicks the buttons.

Demo 2.2: VariablesArraysOperators/Demos/arrays.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.      //Pop up four prompts and create an array
10.     const colors = new Array();
11.     colors[0] = prompt("Choose a color.", "");
12.     colors[1] = prompt("Choose a color.", "");
13.     colors[2] = prompt("Choose a color.", "");
14.     colors[3] = prompt("Choose a color.", "");
15. </script>
16. <title>JavaScript Arrays</title>
17. </head>
18. <body>
19. <main>
20.     <button onclick="document.body.style.backgroundColor = colors[0];">
21.         <script>
22.             document.write(colors[0]);
23.         </script>
24.     </button>
25.     <button onclick="document.body.style.backgroundColor = colors[1];">
26.         <script>
27.             document.write(colors[1]);
28.         </script>
29.     </button>
30.     <button onclick="document.body.style.backgroundColor = colors[2];">
31.         <script>
32.             document.write(colors[2]);
33.         </script>
34.     </button>
35.     <button onclick="document.body.style.backgroundColor = colors[3];">
36.         <script>
37.             document.write(colors[3]);
38.         </script>
39.     </button>
40. </main>
41. </body>
42. </html>
```

As the page loads, an array called colors is declared.

```
colors = new Array();
```

The next four lines populate the array with user-entered values.

```
colors[0] = prompt("Choose a color.", "");  
colors[1] = prompt("Choose a color.", "");  
colors[2] = prompt("Choose a color.", "");  
colors[3] = prompt("Choose a color.", "");
```

The body of the page contains a paragraph with four `<button>` tags, the text of which is dynamically created with values from the `colors` array.

Exercise 3: Working with Arrays

 15 to 25 minutes

In this exercise, you will practice working with arrays.

1. Open `VariablesArrays0operators/Exercises/arrays.html` for editing.
2. Below the comment, declare a `rockStars` array and populate it with four values entered by the user.
3. Add functionality to the buttons, so that alerts pop up with values from the array when the buttons are clicked.
4. Test your solution in a browser. It should work as follows:
 - A. As the page loads, you should get four alerts (the values should be blank by default):

This page says

Who is your favorite rock star?

CancelOK

This page says

Your next favorite rock star?

CancelOK

This page says

Your next favorite rock star?

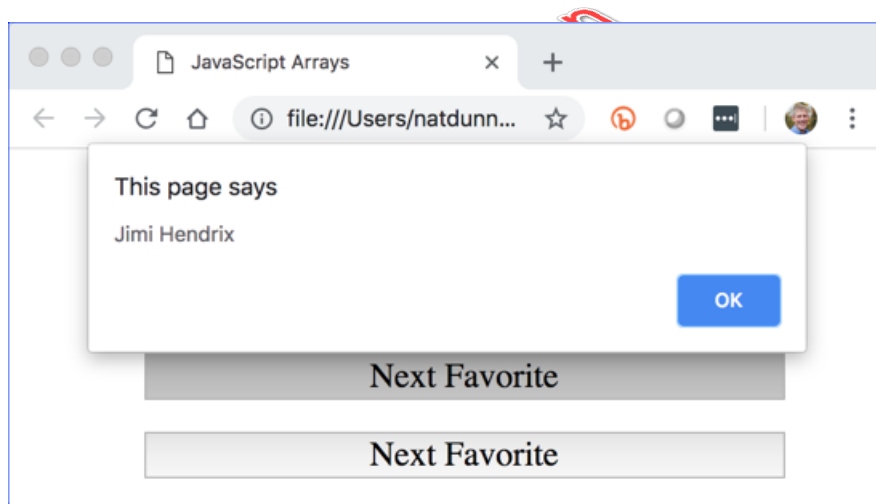
CancelOK

This page says

Your next favorite rock star?

CancelOK

- B. After responding to all the prompts, you should see four buttons on the page. When you click one of the buttons, it should alert one of your rock stars:



Exercise Code 3.1: VariablesArraysOperators/Exercises/arrays.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      /*
10.         Declare a rockStars array and populate it with
11.         four values entered by the user.
12.      */
13. </script>
14. <title>JavaScript Arrays</title>
15. </head>
16. <body>
17. <main>
18.     <button>Favorite</button>
19.     <button>Next Favorite</button>
20.     <button>Next Favorite</button>
21.     <button>Next Favorite</button>
22. </main>
23. </body>
24. </html>
```

Solution: VariablesArraysOperators/Solutions/arrays.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      const rockStars = new Array();
10.     rockStars[0] = prompt("Who is your favorite rock star?", "");
11.     rockStars[1] = prompt("Your next favorite rock star?", "");
12.     rockStars[2] = prompt("Your next favorite rock star?", "");
13.     rockStars[3] = prompt("Your next favorite rock star?", "");
14. </script>
15. <title>JavaScript Arrays</title>
16. </head>
17. <body>
18. <main>
19.     <button onclick="alert(rockStars[0]);">Favorite</button>
20.     <button onclick="alert(rockStars[1]);">Next Favorite</button>
21.     <button onclick="alert(rockStars[2]);">Next Favorite</button>
22.     <button onclick="alert(rockStars[3]);">Next Favorite</button>
23. </main>
24. </body>
25. </html>
```

EVALUATION COPY: Not to be used in class.



2.7. Associative Arrays

Whereas regular (or enumerated) arrays are indexed numerically, associative arrays are indexed using names as keys. The advantage of this is that the keys can be meaningful, which can make it easier to reference an element in an array. The following code, written in Chrome DevTools Console, illustrates how an associative array is used:


```

> const beatles = [];
< undefined
> beatles["singer1"] = "Paul";
< "Paul"
> beatles["singer2"] = "John";
< "John"
> beatles["guitarist"] = "George";
< "George"
> beatles["drummer"] = "Ringo";
< "Ringo"
> beatles;
< ▶ [singer1: "Paul", singer2: "John", guitarist: "George", drummer: "Ringo"]
> beatles["drummer"];
< "Ringo"

```

Arrays can also have subarrays. For example, rather than having “singer1” and “singer2” keys, it would be better to have a “singers” key that was an enumerated array. We could do that like this:

```

> const beatles = [];
< undefined
> beatles["singers"] = ["Paul", "John"];
< ▶ (2) ["Paul", "John"]
> beatles["guitarist"] = "George";
< "George"
> beatles["drummer"] = "Ringo";
< "Ringo"
> beatles;
< ▶ [singers: Array(2), guitarist: "George", drummer: "Ringo"]
> beatles["singers"];
< ▶ (2) ["Paul", "John"]
> beatles["singers"][0];
< "Paul"
> beatles["singers"][1];
< "John"

```

Notice how the singers are accessed first by the “singers” key of the `beatles` array and then by the index:

```
beatles['singers'][0];
```

❖ 2.7.1. Array Properties and Methods

The tables below show some of the most useful array properties and methods. All of the examples assume an array called `beatles` that holds “Paul”, “John”, “George”, and “Ringo”.

```
const beatles = ["Paul", "John", "George", "Ringo"];
```

Array Properties

Property	Description
length	Holds the number of elements in an array. beatles.length // 4

Array Methods

Property	Description
join(delimiter)	Returns a string comprised of the elements in the array. The elements are joined together by the delimiter. The default delimiter is a comma. beatles.join(":") // Paul:John:George:Ringo beatles.join() // Paul,John,George,Ringo
push()	Appends an element to an array. beatles.push("Steve")
pop()	Removes the last item in an array and returns its value. beatles.pop() // Returns Ringo
shift()	Removes the first item in an array and returns its value. beatles.shift() // Returns Paul
unshift()	Prepends one or more items to the beginning of an array. beatles.unshift('Paul')
slice(start, end)	Returns a subarray from start up to, but not including end. If end is left out, it includes the remainder of the array. beatles.slice(1, 3) //Returns [John, George]
splice(start, count)	Removes count items from start in the array and returns the resulting array. beatles.splice(1, 2) //Returns [Paul, Ringo]
sort()	Sorts an array alphabetically. beatles.sort() //Returns [George, John, Paul, Ringo] and sorts the array



2.8. Playing with Array Methods

Take some time to play around with these array methods in Chrome DevTools Console. Try your own things and/or follow along with the following code.

```
> const beatles = ['Paul', 'John', 'George', 'Ringo'];
< undefined
> beatles.length;
< 4
> beatles.join(':');
< "Paul:John:George:Ringo"
> beatles.push('Steve');
< 5
> // Notice that beatles now contains Steve as the last element.
< undefined
> // Let's pop it off.
  beatles.pop();
< "Steve"
> // Now we're back to the original beatles:
  beatles;
< ▶ (4) ["Paul", "John", "George", "Ringo"]
> // shift() is like pop() but it returns and removes the first item:
  beatles.shift();
< "Paul"
> // Now beatles will be missing Paul:
  beatles;
< ▶ (3) ["John", "George", "Ringo"]
> // Let's use unshift to bring Paul back:
  beatles.unshift('Paul');
< 4
> beatles;
< ▶ (4) ["Paul", "John", "George", "Ringo"]
```

Note that some methods will return a value without modifying the existing array, while others will make changes to the existing array “in place”. For example, study the following code. Notice that `slice()` returns a new array without changing the existing array, whereas `splice()` and `sort()` make changes to the existing array.

```
> beatles.slice(1,3); // Does not change existing array
< ▶ (2) ["John", "George"]
> beatles;
< ▶ (4) ["Paul", "John", "George", "Ringo"]
> beatles.splice(1,2); // Changes existing array
< ▶ (2) ["John", "George"]
> beatles;
< ▶ (2) ["Paul", "Ringo"]
> beatles = ['Paul', 'John', 'George', 'Ringo'];
< ▶ (4) ["Paul", "John", "George", "Ringo"]
> beatles.sort();
< ▶ (4) ["George", "John", "Paul", "Ringo"]
> beatles;
< ▶ (4) ["George", "John", "Paul", "Ringo"]
```

slice() returns a new array, but the original beatles array still contains all 4 names.

splice() returns a new array, and changes the original beatles array, removing two of the names.

sort() changes the array "in place," meaning it returns the same array after modifying it.

Array Documentation

See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array for full documentation on Arrays.

EVALUATION COPY: Not to be used in class.



2.9. JavaScript Operators

Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)

Assignment Operators

Operator	Description
=	Assignment
+=	One step addition and assignment (a+=3 is the same as a=a+3)
-=	One step subtraction and assignment (a-=3 is the same as a=a-3)
=	One step multiplication and assignment (a=3 is the same as a=a*3)
/=	One step division and assignment (a/=3 is the same as a=a/3)
%=	One step modulus and assignment (a%=3 is the same as a=a%3)
++	Increment by one (a++ is the same as a=a+1 or a+=1)
--	Decrement by one (a-- is the same as a=a-1 or a-=1)

String Operators

Operator	Description
+	Concatenation
	<code>let greeting = "Hello " + firstname;</code>
+=	One step concatenation and assignment
	<code>let greeting = "Hello ";</code> <code>greeting += firstname;</code>

The following code, written in Chrome DevTools Console, shows examples of working with JavaScript arithmetic operators:

```

> let a=5, b=4;
< undefined
> a+b;
< 9
> a-b;
< 1
> a*b;
< 20
> a/b;
< 1.25
> a%b; // Modulus returns remainder
< 1
> c=a++; // Assigns a to c and then increments a by 1
< 5
> c;
< 5
> a;
< 6
> d = a--; // Assigns a to d and then decrements a by 1
< 6
> d;
< 6
> a;
< 5
> a = a + 2; // Adds 2 to a
< 7
> a+=2; // Adds 2 to a
< 9

```

And here we have examples of the concatenation operator:

```

> let greeting = 'Hello';
< undefined
> let firstName = 'Nat';
< undefined
> greeting + ', ' + firstName; // Concatenation
< "Hello, Nat"
> let fullGreeting = greeting + ', ' + firstName;
< undefined
> fullGreeting;
< "Hello, Nat"
> fullGreeting += '!';
< "Hello, Nat!"
> fullGreeting;
< "Hello, Nat!"

```



2.10. The Modulus Operator

The *modulus* operator (%) is used to find the remainder after division:

```
5 % 2 // returns 1
11 % 3 // returns 2
22 % 4 // returns 2
22 % 3 // returns 1
10934 % 324 // returns 242
```

The modulus operator is useful for determining whether a number is even or odd:

```
1 % 2 // returns 1: odd
2 % 2 // returns 0: even
3 % 2 // returns 1: odd
4 % 2 // returns 0: even
5 % 2 // returns 1: odd
6 % 2 // returns 0: even
```

Evaluation
Copy



2.11. Playing with Operators

Take some time to play around with JavaScript operators in Chrome DevTools Console. Try your own things and/or follow along with the code in the preceding sections.

The file below illustrates the use of the concatenation operator and several math operators. It also illustrates a potential problem with loosely typed languages.

Demo 2.3: VariablesArraysOperators/Demos/operators.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      const userNum1 = window.prompt("Choose a number.", "");
10.     alert("You chose " + userNum1);
11.     const userNum2 = window.prompt("Choose another number.", "");
12.     alert("You chose " + userNum2);
13.     const numsAdded = userNum1 + userNum2;
14.     const numsSubtracted = userNum1 - userNum2;
15.     const numsMultiplied = userNum1 * userNum2;
16.     const numsDivided = userNum1 / userNum2;
17.     const numsModulused = userNum1 % userNum2;
18. </script>
19. <title>JavaScript Operators</title>
20. </head>
21. <body>
22. <main>
23.     <p>
24.         <script>
25.             document.write(userNum1 + " + " + userNum2 + " = ");
26.             document.write(numsAdded + "<br>");
27.             document.write(userNum1 + " - " + userNum2 + " = ");
28.             document.write(numsSubtracted + "<br>");
29.             document.write(userNum1 + " * " + userNum2 + " = ");
30.             document.write(numsMultiplied + "<br>");
31.             document.write(userNum1 + " / " + userNum2 + " = ");
32.             document.write(numsDivided + "<br>");
33.             document.write(userNum1 + " % " + userNum2 + " = ");
34.             document.write(numsModulused + "<br>");
35.         </script>
36.     </p>
37. </main>
38. </body>
39. </html>
```



This page is processed as follows:

1. The user is prompted for a number and the result is assigned to userNum1:

This page says

Choose a number.

Cancel OK

2. An alert pops up telling the user what number they entered. The concatenation operator (+) is used to combine two strings: “You chose ” and the number entered by the user. Note that all user-entered data is always treated as a string of text, even if the text consists of only digits:

This page says

You chose 5

OK

3. The user is prompted for another number and the result is assigned to `userNum2`:

This page says

Choose another number.

Cancel OK

4. Another alert pops up telling the user what number they entered:

This page says

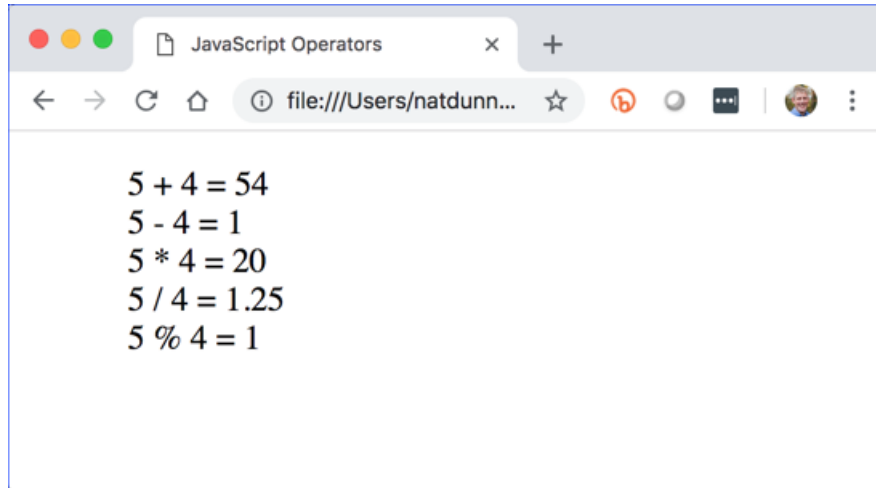
You chose 4

OK

5. Five constants are declared and assigned values :

```
const numsAdded = userNum1 + userNum2;  
const numsSubtracted = userNum1 - userNum2;  
const numsMultiplied = userNum1 * userNum2;  
const numsDivided = userNum1 / userNum2;  
const numsModulus = userNum1 % userNum2;
```

6. The values the constants contain are output to the browser:



So, $5 + 4$ is 54?? Well, only if 5 and 4 are strings, and, as stated earlier, all user-entered data is treated as a string. Don't worry. We will learn how to fix this problem soon.

EVALUATION COPY: Not to be used in class.



2.12. The Default Operator

Default Operator

Operator	Description
	Used to assign a default value. <code>const yourName = prompt("Your Name?", "") "Stranger";</code>

The following code sample shows how the default operator works:

Demo 2.4: VariablesArraysOperators/Demos/default.html

```
-----Lines 1 through 7 Omitted-----  
8.  <script>  
9.    const yourName = prompt("Your Name?", "") || "Stranger";  
10.  
11.    alert("Hi " + yourName + "!");  
12.  </script>  
-----Lines 13 through 20 Omitted-----
```

If the user presses **OK** without filling out the prompt or presses **Cancel**, the default value “Stranger” is assigned to the `yourName` constant.

Why do we need a default operator?

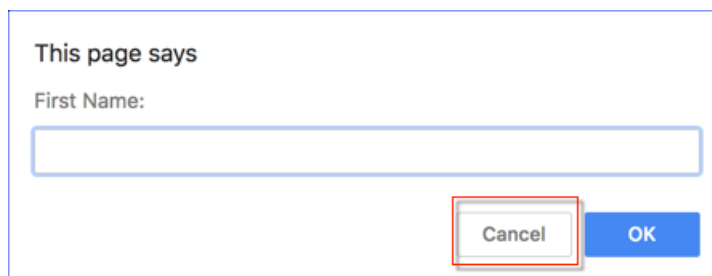
The default operator allows you to make sure that your variable contains a non-null value, so that you can perform operations on the variable with no errors. To illustrate, do the following in the Chrome DevTools Console:

1. Enter the following code and press **Enter**:

```
let firstName = prompt("First Name:", "");
```

This will cause a prompt to pop up.

2. Press the **Cancel** button. This will return null and assign it to `firstName`:



3. Enter the following code and press **Enter**:

```
let greeting = "Hello, " + firstName;
```

4. Then output `greeting` and you'll see this strange result:

```
> let firstName = prompt("First name:", "");  
< undefined  
> let greeting = "Hello, " + firstName;  
< undefined  
> greeting;  
< "Hello, null"
```

Now repeat the above, but start with:

```
let firstName = prompt("First Name:", "") || "Stranger";
```

This time, when you press **Cancel**, the default value of “Stranger” will be assigned to `firstName` and the concatenation operation will work fine:

```
> let firstName = prompt("First name:", "") || "Stranger";  
< undefined  
> let greeting = "Hello, " + firstName;  
< undefined  
> greeting;  
< "Hello, Stranger"
```

Exercise 4: Working with Operators

 15 to 25 minutes

In this exercise, you will practice working with JavaScript operators.

1. Open `VariablesArrays0operators/Exercises/operators.html` for editing.
2. Add code to prompt the user for the number of songs they have downloaded of their favorite and second favorite rock stars:

This page says

Who is your favorite rock star?

Cancel OK

This page says

How many Elvis songs do you have?

Cancel OK

This page says

And your next favorite rock star?

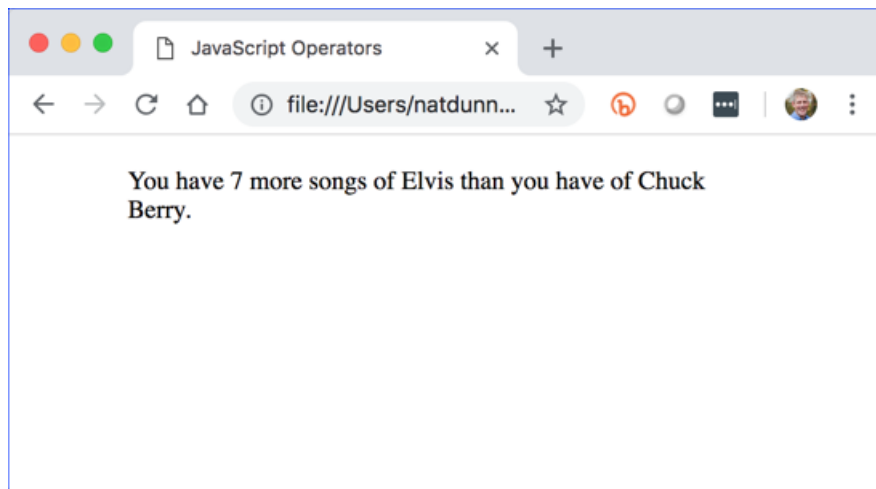
Cancel OK

This page says

How many Chuck Berry songs do you have?

CancelOK

3. In the body, let the user know how many more of their favorite rock star's songs they have than of their second favorite rock star's songs:



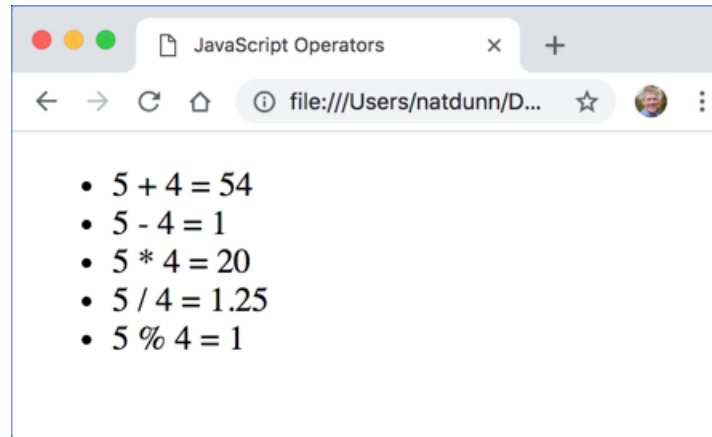
4. Test your solution in a browser.

Exercise Code 4.1: VariablesArraysOperators/Exercises/operators.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     const rockStars = [];
10.    rockStars[0] = prompt("Who is your favorite rock star?", "");
11.    /*
12.    Ask the user how many of this rockstar's songs they have downloaded
13.    and store the result in a variable.
14.    */
15.    rockStars[1] = prompt("And your next favorite rock star?", "");
16.    /*
17.    Ask the user how many of this rockstar's songs they have downloaded
18.    and store the result in a variable.
19.    */
20. </script>
21. <title>JavaScript Operators</title>
22. </head>
23. <body>
24. <main>
25. <!--
26.     Let the user know how many more of their favorite rock star's songs
27.     they have than of their second favorite rock star's songs.
28. -->
29. </main>
30. </body>
31. </html>
```

Challenge

1. Open VariablesArraysOperators/Exercises/operators-challenge.html for editing.
2. Modify it so that it outputs an unordered list as shown below:



Don't worry about the 54. We will learn how to fix the addition problem soon.

Solution: VariablesArraysOperators/Solutions/operators.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      const rockStars = [];
10.     const songTotals = [];
11.     rockStars[0] = prompt("Who is your favorite rock star?", "");
12.     songTotals[0] = prompt("How many " + rockStars[0] +
13.                            " songs do you have?", "");
14.     rockStars[1] = prompt("And your next favorite rock star?", "");
15.     songTotals[1] = prompt("How many " + rockStars[1] +
16.                            " songs do you have?", "");
17. </script>
18. <title>JavaScript Operators</title>
19. </head>
20. <body>
21. <main>
22.     <script>
23.         const diff = songTotals[0] - songTotals[1];
24.         document.write("You have " + diff + " more songs of " + rockStars[0]);
25.         document.write(" than you have of " + rockStars[1] + ".");
26.     </script>
27. </main>
28. </body>
29. </html>
```

Challenge Solution:

<VariablesArraysOperators/Solutions/operators-challenge.html>

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      const userNum1 = prompt("Choose a number.", "");
10.     alert("You chose " + userNum1);
11.     const userNum2 = prompt("Choose another number.", "");
12.     alert("You chose " + userNum2);
13.     const numsAdded = userNum1 + userNum2;
14.     const numsSubtracted = userNum1 - userNum2;
15.     const numsMultiplied = userNum1 * userNum2;
16.     const numsDivided = userNum1 / userNum2;
17.     const numsModulused = userNum1 % userNum2;
18. </script>
19. <title>JavaScript Operators</title>
20. </head>
21. <body>
22. <main>
23. <ul>
24.     <script>
25.         document.write("<li>" + userNum1 + " + " + userNum2 + " = ");
26.         document.write(numsAdded + "</li>");
27.         document.write("<li>" + userNum1 + " - " + userNum2 + " = ");
28.         document.write(numsSubtracted + "</li>");
29.         document.write("<li>" + userNum1 + " * " + userNum2 + " = ");
30.         document.write(numsMultiplied + "</li>");
31.         document.write("<li>" + userNum1 + " / " + userNum2 + " = ");
32.         document.write(numsDivided + "</li>");
33.         document.write("<li>" + userNum1 + " % " + userNum2 + " = ");
34.         document.write(numsModulused + "</li>");
35.     </script>
36. </ul>
37. </main>
38. </body>
39. </html>
```

Conclusion

In this lesson, you have learned to work with JavaScript variables, arrays and operators.

LESSON 3

JavaScript Functions

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☑ JavaScript's global functions and objects.
- ☑ Creating your own functions.
- ☑ Returning values from functions.

Introduction

In this lesson, you will learn to use some of JavaScript's built-in functions, and you will also learn to create your own.

EVALUATION COPY: Not to be used in class.



3.1. Global Objects and Functions

A “global” function or object is one that is accessible from anywhere. JavaScript has a number of global objects and functions. We will examine some of them in this section.

❖ 3.1.1. parseFloat(object)

The `parseFloat()` function takes one argument: an object, and attempts to return a floating point number, which is a decimal number. If it cannot, it returns `NaN`, for “Not a Number.”

Remember when we “add” two strings using the plus sign (+), the strings are concatenated together, as the following code illustrates:

```
const strNum1 = '1';
const strNum2 = '2';
const strSum = strNum1 + strNum2;
strSum; // will return "12"
```

Because `strNum1` and `strNum2` are both strings, the `+` operator concatenates them, resulting in `"12"`.

We can use `parseFloat()` to convert those strings to numbers before adding them:

```
const strNum1 = '1';
const strNum2 = '2';
const num1 = parseFloat(strNum1);
const num2 = parseFloat(strNum2);
const sum = num1 + num2;
sum; // will return 3
```

After the `parseFloat()` function has been used to convert the strings to numbers, the `+` operator performs addition, resulting in `3`.

If the value passed to `parseFloat()` doesn't start with a number, the function returns `NaN`:

```
parseFloat('I want 1.5 apples'); // will return NaN
```

❖ 3.1.2. `parseInt(object)`

The `parseInt()` function is similar to `parseFloat()`. It takes one argument: an object, and attempts to return an integer. If it cannot, it returns `NaN`, for “Not a Number.”

As you can see from the following code, `parseInt()` just strips everything to the right of the first integer it finds. If the value passed to `parseInt()` doesn't start with an integer, the function returns `NaN`:

```
parseInt('1'); // will return 1
parseInt('1.5'); // will return 1
parseInt('1.5 apples'); // will return 1
parseInt('I want 1.5 apples'); // will return NaN
```

❖ 3.1.3. isNaN(object)

The `isNaN()` function takes one argument: an object. The function checks if the object is *not* a number (or cannot be converted to a number). It returns `true` if the object is not a number and `false` if it is a number:

```
isNaN(4); // will return false  
isNaN('4'); // will return false  
isNaN('hello'); // will return true
```

As you can see from the code above, if the passed-in value is a number or can be converted into a number (e.g., 4 and '4'), `isNaN()` returns `false`. Otherwise (e.g., 'hello'), it returns `true`, meaning that it **is** indeed **Not a Number**.



Exercise 5: Working with Global Functions

🕒 10 to 15 minutes

In this exercise, you will practice working with JavaScript's global functions.

1. Open `JavaScriptFunctions/Exercises/built-in-functions.html` for editing.
2. As the code is currently written (see below), it will concatenate the user-entered numbers rather than add them. Fix this so that it outputs the sum of the two numbers entered by the user.

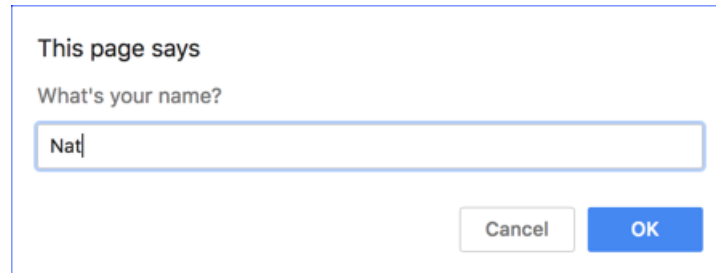
Exercise Code 5.1: `JavaScriptFunctions/Exercises/built-in-functions.html`

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     let userNum1;
10.    let userNum2;
11.    let numsAdded;
12.    userNum1 = window.prompt("Choose a number.", "");
13.    alert("You chose " + userNum1);
14.    userNum2 = window.prompt("Choose another number.", "");
15.    alert("You chose " + userNum2);
16.    numsAdded = userNum1 + userNum2;
17. </script>
18. <title>JavaScript Built-in Functions</title>
19. </head>
20. <body>
21. <p>
22.     <script>
23.         document.write(userNum1 + " + " + userNum2 + " = ");
24.         document.write(numsAdded);
25.     </script>
26. </p>
27. </body>
28. </html>
```


Challenge

Create a new HTML file that prompts the user for

1. Their name:



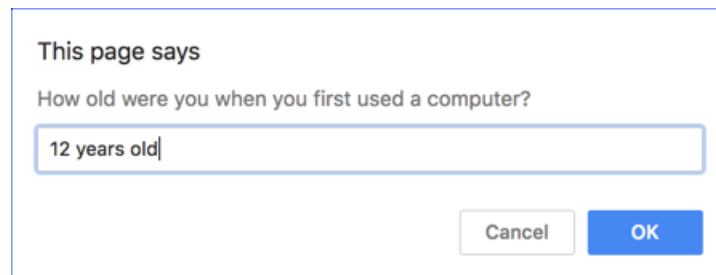
This page says

What's your name?

Nat

Cancel OK

The age at which they first worked on a computer:



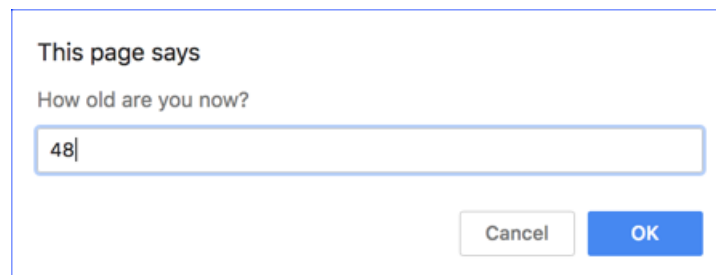
This page says

How old were you when you first used a computer?

12 years old

Cancel OK

And their current age:



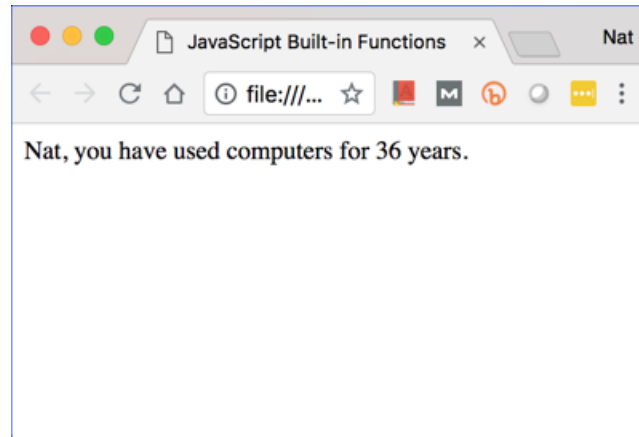
This page says

How old are you now?

48

Cancel OK

After gathering this information, write out to the page how many years they have been working on a computer:



Notice that the program is able to deal with numbers followed by strings (e.g., “12 years old”).

Solution: JavaScriptFunctions/Solutions/built-in-functions.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      let userNum1;
10.     let userNum2;
11.     let numsAdded;
12.     userNum1 = window.prompt("Choose a number.", "");
13.     alert("You chose " + userNum1);
14.     userNum2 = window.prompt("Choose another number.", "");
15.     alert("You chose " + userNum2);
16.     numsAdded = parseFloat(userNum1) + parseFloat(userNum2);
17. </script>
18. <title>JavaScript Built-in Functions</title>
19. </head>
20. <body>
21. <main>
22.     <p>
23.         <script>
24.             document.write(userNum1 + " + " + userNum2 + " = ");
25.             document.write(numsAdded);
26.         </script>
27.     </p>
28. </main>
29. </body>
30. </html>
```

Challenge Solution:

JavaScriptFunctions/Solutions/built-in-functions-challenge.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     const userName = prompt("What's your name?");
10.    const age1 = prompt('How old were you when you first used a computer?');
11.    const age2 = prompt('How old are you now?');
12.    const diff = parseFloat(age2) - parseFloat(age1);
13. </script>
14. <title>JavaScript Built-in Functions</title>
15. </head>
16. <body>
17. <main>
18.     <p>
19.         <script>
20.             document.write(userName + ', you have used '
21.                             + 'computers for ' + diff + ' years. ');
22.         </script>
23.     </p>
24. </main>
25. </body>
26. </html>
```

Code Explanation

You may have noticed that we are not including the second argument, which is the default value, for `prompt()` in the challenge solution. While these could be written as `const age2 = prompt("How old are you now?", "");`, this is not necessary as an empty string is the default value.

EVALUATION COPY: Not to be used in class.



3.2. User-defined Functions

Writing functions makes it possible to reuse code for common tasks. Functions can also be used to hide complex code. For example, an experienced developer can write a function for performing a complicated task. Other developers do not need to know how that function works; they only need to know how to call it.

❖ 3.2.1. Function Syntax

JavaScript functions generally appear in the head of the page or in external JavaScript files. A function is written using the `function` keyword followed by the name of the function.

```
function doSomething() {  
    //function statements go here  
}
```

As you can see, the body of the function is contained within curly brackets (`{}`). The following example demonstrates the use of simple functions:

Demo 3.1: JavaScriptFunctions/Demos/simple-functions.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     function changeBgRed() {
10.         document.body.style.backgroundColor = "red";
11.     }
12.
13.     function changeBgWhite() {
14.         document.body.style.backgroundColor = "white";
15.     }
16. </script>
17. <title>JavaScript Simple Functions</title>
18. </head>
19. <body>
20.     <button onclick="changeBgRed();">Red</button>
21.     <button onclick="changeBgWhite();">White</button>
22. </body>
23. </html>
```

When the user clicks one of the buttons, the event is captured by the onclick event handler and the corresponding function is called.

❖ 3.2.2. Passing Values to Functions

The functions above aren't very useful because they always do the same thing. Every time we wanted to add another color, we would have to write another function. Also, if we want to modify the behavior, we will have to do it in each function. The following example shows how to create a single function to handle changing the background color.

Demo 3.2: JavaScriptFunctions/Demos/passing-values.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     function changeBg(color) {
10.         document.body.style.backgroundColor = color;
11.     }
12. </script>
13. <title>Passing Values</title>
14. </head>
15. <body>
16.     <button onclick="changeBg('red');">Red</button>
17.     <button onclick="changeBg('white');">White</button>
18. </body>
19. </html>
```

As you can see, when calling the `changeBg()` function, we pass a value (e.g., `'red'`), which is assigned to the `color` variable. We can then refer to the `color` variable throughout the function. Variables created in this way are called “parameters” and the values passed to them are called “arguments”. A function can have any number of parameters, separated by commas.

Adding parameters to functions makes them more flexible and, thus, more useful; as you saw above, we can call the `changeBg()` function many times, passing to it a different color as needed. We can make our functions even more useful by providing default values for parameters so that, if the function is called without an argument, we assign some default value to the parameter. Here’s how we might modify our earlier example:

Demo 3.3:

JavaScriptFunctions/Demos/passing-values-default-param.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     function changeBg(color='blue') {
10.         document.body.style.backgroundColor = color;
11.     }
12. </script>
13. <title>Passing Values - Default Param</title>
14. </head>
15. <body>
16. <p>
17.     <button onclick="changeBg('red');">Red</button>
18.     <button onclick="changeBg('white');">White</button>
19.     <button onclick="changeBg();">Blue (no param)</button>
20. </p>
21. </body>
22. </html>
```

We've added a default value for `changeBg`'s `color` parameter, giving it the value `'blue'` if no value is supplied when the function is called. We've also added a third button on which the user can click; here we call `changeBg()` (without a parameter for `color`) and thus get the default color `'blue'`.

A Note on Variable Scope

A variable's "scope" is the context in which the variable can be referenced. Variables created by passing arguments to function parameters are local to the function, meaning that they cannot be accessed outside of the function. The same is true for variables declared within a function using the `let` keyword.

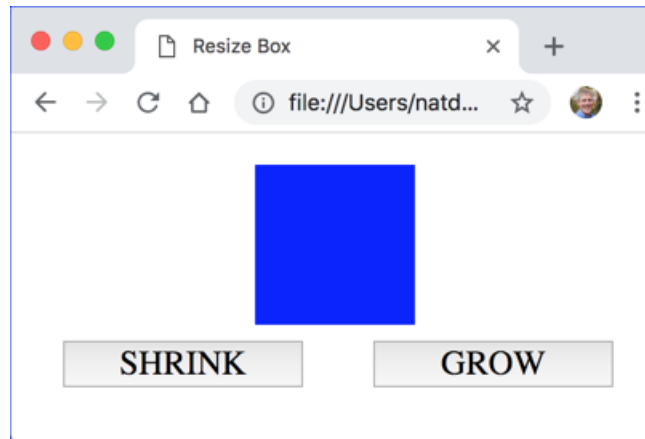
Variables declared with `let` outside of a function can only be used in the block of code in which the variable is defined.



Exercise 6: Writing a JavaScript Function

🕒 15 to 25 minutes

In this exercise, you will modify a page called `resize-box.html`, which will contain a box and two buttons for resizing the box:



1. Open `JavaScriptFunctions/Exercises/resize-box.html` for editing.
2. Notice that the page has a `div` with the id “box” and width and height styles set.
3. The page also contains two buttons that call `resizeBox()` passing in `-10` and `10` for the change argument.
4. Write a function called `resizeBox()` that has one parameter: `change`, which is the amount the width and height of the box should be changed. The default value of `change` should be `10`. The `resizeBox()` function will need to do the following:

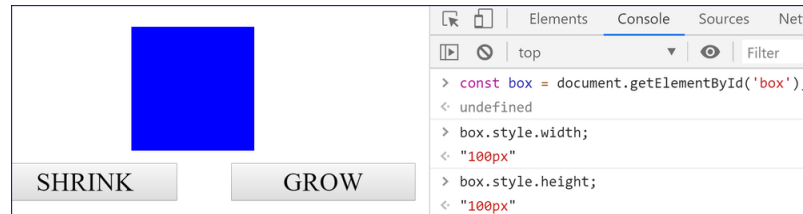
- A. Declare a constant `box` that holds the “box” `div`. You will do this using `document.getElementById()`, which is a method for accessing elements on the page by their id value:

```
const box = document.getElementById('box');
```

- B. Declare a constant `w` that holds the current width of the box. You will do this with the following line of code:

```
const w = box.style.width;
```

Note that the value will be a string ending in “px” as shown below. This is because width and height style values take a number and a unit.



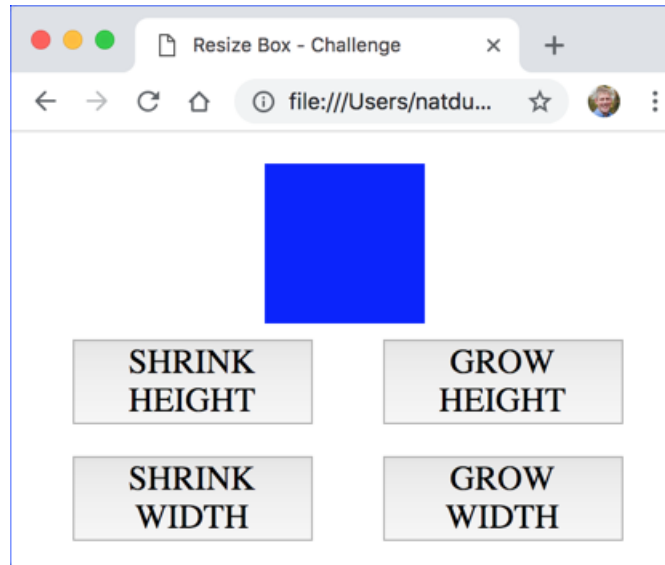
- C. Just as you did for width, declare a constant `h` that holds the current height of the box.
- D. Declare variables `wNew` and `hNew` that contain the new width and height values. Note that you will need to add the value of `change` to the current values of `w` and `h`, but before doing so, you will need to strip off the “px” from `w` and `h` and convert those values to numbers. You can do that with `parseInt()`.
- E. Assign the new values of `w` and `h` to `box.style.width` and `box.style.height`. Note that you will need to append (concatenate) “px” back to those values.

Exercise Code 6.1: JavaScriptFunctions/Exercises/resize-box.html

```
1. <!DOCTYPE html>  
2. <html lang="en">  
3. <head>  
4. <meta charset="UTF-8">  
5. <meta name="viewport" content="width=device-width,initial-scale=1">  
6. <link rel="stylesheet" href="../normalize.css">  
7. <link rel="stylesheet" href="../styles.css">  
8. <script>  
9.     // Write your code here  
10. </script>  
11. <title>Resize Box</title>  
12. </head>  
13. <body id="resize-box">  
14. <main>  
15.     <div id="box" style="width:100px; height:100px;  
16.         background-color:blue;"></div>  
17.     <button onclick="resizeBox(-10)">SHRINK</button>  
18.     <button onclick="resizeBox(10)">GROW</button>  
19. </main>  
20. </body>  
21. </html>
```

Challenge

Add separate buttons for changing height and width:



As we haven't learned to write conditional code yet, you will need to write separate functions; for example, `resizeBoxHeight()` and `resizeBoxWidth()`.

Solution: JavaScriptFunctions/Solutions/resize-box.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.  function resizeBox(change=10) {
10.    const box = document.getElementById('box');
11.    const w = box.style.width;
12.    const h = box.style.height;
13.    const wNew = parseInt(w) + change;
14.    const hNew = parseInt(h) + change;
15.    box.style.width = wNew + 'px';
16.    box.style.height = hNew + 'px';
17.  }
18. </script>
19. <title>Resize Box</title>
20. </head>
21. <body id="resize-box">
22. <main>
23.   <div id="box" style="width:100px; height:100px;"></div>
24.   <button onclick="resizeBox(-10)">SHRINK</button>
25.   <button onclick="resizeBox(10)">GROW</button>
26. </main>
27. </body>
28. </html>
```

Challenge Solution:

JavaScriptFunctions/Solutions/resize-box-challenge.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.  function resizeHeight(change=10) {
10.     const box = document.getElementById('box');
11.     const h = box.style.height;
12.     const hNew = parseInt(h) + change;
13.     box.style.height = hNew + 'px';
14.  }
15.
16.  function resizeWidth(change=10) {
17.     const box = document.getElementById('box');
18.     const w = box.style.width;
19.     const wNew = parseInt(w) + change;
20.     box.style.width = wNew + 'px';
21.  }
22. </script>
23. <title>Resize Box - Challenge</title>
24. </head>
25. <body id="resize-box">
26. <main>
27.   <div id="box" style="width:100px; height:100px;"></div>
28.   <button onclick="resizeHeight(-10)">SHRINK HEIGHT</button>
29.   <button onclick="resizeHeight(10)">GROW HEIGHT</button><br>
30.   <button onclick="resizeWidth(-10)">SHRINK WIDTH</button>
31.   <button onclick="resizeWidth(10)">GROW WIDTH</button>
32. </main>
33. </body>
34. </html>
```

EVALUATION COPY: Not to be used in class.



3.3. Returning Values from Functions

The return keyword is used to return values from functions as the following example illustrates:

Demo 3.4: JavaScriptFunctions/Demos/return-value.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.  function setBgColor() {
10.    const bg = prompt("Set Background Color:", "");
11.    document.body.style.backgroundColor = bg;
12.  }
13.
14.  function getBgColor() {
15.    return document.body.style.backgroundColor;
16.  }
17. </script>
18. <title>Returning a Value</title>
19. </head>
20. <body>
21.   <button onclick="setBgColor()">Set Background Color</button>
22.   <button onclick="alert(getBgColor())">Get Background Color</button>
23. </body>
24. </html>
```

When the user clicks the “Get Background Color” button, an alert pops up with a value returned from the `getBgColor()` function. This is a very simple example. Generally, functions that return values are a bit more involved. We’ll see many more functions that return values throughout the course.

Conclusion

In this lesson, you have learned to work with JavaScript’s global functions and to create functions of your own.

LESSON 4

Built-In JavaScript Objects

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☑ Built-in String object.
- ☑ Built-in Math object.
- ☑ Built-in Date object.

Introduction

JavaScript has some predefined, built-in objects that enable you to work with Strings and Dates, and perform mathematical operations.

EVALUATION COPY: Not to be used in class.



4.1. String

In JavaScript, there are two types of string data types: primitive strings and *String* objects. String objects have many methods for manipulating and parsing strings of text. Because these methods are available to primitive strings as well, in practice, there is no need to differentiate between the two types of strings.

Some common string properties and methods are shown below. In all the examples, the constant `myStr` contains “Webucator”:

```
const myStr = 'Webucator';
```

Common String Properties

Property	Description
length	Read-only value containing the number of characters in the string. myStr.length; // returns 9

Try the following out in the Chrome DevTools Console:

```
const myStr = 'Webucator';  
myStr.length; // will return 9
```

Evaluation
Copy

Spend some time going through methods in the table below and trying them out in the Chrome DevTools Console. Note that most programming languages have similar string methods, though they may use different names. Some of the string methods will seem obscure (“When would I use that?”). Don’t worry too much about that. The most important takeaway is to understand that there are a lot of built-in methods for working with strings and to get some practice using them.

Common String Methods

Method	Description
charAt(position)	Returns the character at the specified position.
	<pre>myStr.charAt(4); // returns 'c'</pre> <pre>myStr.charAt(0); // returns 'W'</pre>
indexOf(substr, startPos)	Searches from startPos (or the beginning of the string, if startPos is not supplied) for substr. Returns the first position at which substr is found or -1 if substr is not found.
	<pre>myStr.indexOf("cat"); // returns 4</pre> <pre>myStr.indexOf("cat", 5); // returns -1</pre>
lastIndexOf(substr, endPos)	Searches from endPos (or the end of the string, if endPos is not supplied) for substr. Returns the last position at which substr is found or -1 if substr is not found.
	<pre>myStr.lastIndexOf("cat"); // returns 4</pre> <pre>myStr.lastIndexOf("cat", 5); // returns 4</pre>
substring(startPos, endPos)	Returns the substring beginning at startPos and ending with the character before endPos. endPos is optional. If it is excluded, the substring continues to the end of the string.
	<pre>myStr.substring(4, 7); // returns cat</pre> <pre>myStr.substring(4); // returns cator</pre>
slice(startPos, endPos)	Same as substring(startPos, endPos).
	<pre>myStr.slice(4, 7); // returns cat</pre>
slice(startPos, posFromEnd)	posFromEnd is a negative integer. Returns the substring beginning at startPos and ending posFromEnd characters from the end of the string.
	<pre>myStr.slice(4, -2); // returns cat</pre>
split(delimiter)	Returns an array by splitting a string on the specified delimiter.
	<pre>const s = "A,B,C,D"; const a = s.split(","); document.write(a[2]); // returns C</pre>

Method	Description
toLowerCase()	Returns the string in all lowercase letters.
	<code>myStr.toLowerCase(); // returns webucator</code>
toUpperCase()	Returns the string in all uppercase letters.
	<code>myStr.toUpperCase(); // returns WEBUCATOR</code>
trim()	Removes leading and trailing whitespace.
	<code>' Webucator '.trim(); // returns Webucator with no spaces around it</code>

Below are the same methods from the table above shown in the Chrome DevTools Console:

```

> const myStr = 'Webucator';
< undefined
> myStr.charAt(4);
< "c"
> myStr.indexOf('cat');
< 4
> myStr.indexOf('dog'); // Returns -1 (not found)
< -1
> myStr.lastIndexOf('cat');
< 4
> 'banana'.indexOf('an'); // First occurrence of 'an'
< 1
> 'banana'.lastIndexOf('an'); // Last occurrence of 'an'
< 3
> myStr.substring(4,7);
< "cat"
> myStr.substring(4);
< "cator"
> myStr.slice(4,7);
< "cat"
> myStr.slice(4,-2);
< "cat"
> myStr.toLowerCase();
< "webucator"
> myStr.toUpperCase();
< "WEBUCATOR"
> ' webucator '.trim();
< "webucator"

```

Splitting a String

The `split()` method returns an array by splitting a string on the specified delimiter (separator). The following code illustrates this:

```
const s = "A,B,C,D";  
const a = s.split(",");  
a[2]; // returns C
```

Try it out in the Chrome DevTools Console:

```
> const s = 'A,B,C,D';  
< undefined  
> const a = s.split(',');  
< undefined  
> a;  
< ▼ (4) ["A", "B", "C", "D"] ⓘ  
  0: "A"  
  1: "B"  
  2: "C"  
  3: "D"  
  length: 4  
  __proto__: Array(0)  
> a[2];  
< "C"
```

Converting an Object to a String

To convert an object to a string, pass it to `String()`. For example:

```
> let i = 10;  
< undefined  
> typeof i;  
< "number"  
> i = String(i);  
< "10"  
> typeof i;  
< "string"
```

String Documentation

See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String for full documentation on Strings.



4.2. Math

The `Math` object's properties and methods are accessed directly (e.g., `Math.PI`) and are used for performing complex math operations. Some common math properties and methods are shown below:

Common Math Properties

Property	Description
<code>Math.PI</code>	The value of Pi (π)
	<code>Math.PI</code> ; //3.141592653589793
<code>Math.SQRT2</code>	Square root of 2.
	<code>Math.SQRT2</code> ; //1.4142135623730951

Try the following out in the Chrome DevTools Console:

```
> Math.PI;  
◀ 3.141592653589793  
> Math.SQRT2;  
◀ 1.4142135623730951
```

Spend some time going through methods in the table below and trying them out in the Chrome DevTools Console.

Common Math Methods

Method	Description
Math.abs(number)	Absolute value of number.
	Math.abs(-12); // returns 12
Math.ceil(number)	number rounded up.
	Math.ceil(5.4); // returns 6
Math.floor(number)	number rounded down.
	Math.floor(5.6); // returns 5
Math.max(numbers)	Highest Number in numbers.
	Math.max(2, 5, 9, 3); // returns 9
Math.min(numbers)	Lowest Number in numbers.
	Math.min(2, 5, 9, 3); // returns 2
Math.pow(number, power)	number to the power of power.
	Math.pow(2, 5); // returns 32
Math.round(number)	Rounded number.
	Math.round(2.5); // returns 3
Math.random()	Random number between 0 and 1.
	Math.random(); // Returns random number from 0 to 1

Below are the same methods from the table above shown in the Chrome DevTools Console:

> Math.abs(-12);	< 12
> Math.ceil(5.4);	< 6
> Math.floor(5.6);	< 5
> Math.max(2,5,9,3);	< 9
> Math.min(2,5,9,3);	< 2
> Math.pow(2,5);	< 32
> Math.round(2.5);	< 3
> Math.random();	< 0.7473928751077172

Method for Generating Random Integers

Because `Math.random()` returns a decimal value greater than or equal to 0 and less than 1, we can use the following code to return a random integer between `low` and `high`, inclusively (meaning the low and high values are included):

```
function randInt(low, high) {  
  const rndDec = Math.random();  
  const rndInt = Math.floor(rndDec * (high - low + 1) + low);  
  return rndInt;  
}
```

And here it is in the Chrome DevTools Console:


```

> function randInt(low, high) {
    const rndDec = Math.random();
    const rndInt = Math.floor(rndDec * (high - low + 1) + low);
    return rndInt;
}
< undefined
> randInt(1, 10);
< 6
> randInt(1, 100);
< 77

```

Math Documentation

See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math for full documentation on Math.

EVALUATION COPY: Not to be used in class.

Evaluation
Copy

4.3. Date

The Date object has methods for manipulating dates and times. JavaScript stores dates as the number of milliseconds since January 1, 1970.

The Epoch

The **epoch** is the moment that a computer or computer language considers time to have started. JavaScript considers the epoch to be January 1, 1970 at midnight (1970-01-01 00:00:00)

The following code samples show the different methods of creating date objects, all of which involve passing arguments to the `Date()` *constructor* (a special function for creating objects):

New Date object with current date and time

```

const now = new Date();
now; // returns Thu Nov 11, 2021 18:40:31 GMT-0500 (Eastern Standard Time)

```

New Date object with specific date and time

```
// Syntax: new Date('month dd, yyyy hh:mm:ss')
const moonLanding = new Date('July 21, 1969 16:18:00');
moonLanding; // returns Mon Jul 21, 1969 16:18:00 GMT-0400 (Eastern Daylight Time)

// Alternative Syntax: new Date(year, month, day, hours, min, sec, millisec)
const moonLanding = new Date(1969, 6, 21, 16, 18, 0, 0);
moonLanding; // returns Mon Jul 21, 1969 16:18:00 GMT-0400 (Eastern Daylight Time)
```

A few things to note:

1. To create a `Date` object containing the current date and time, the `Date()` constructor takes no arguments.
2. When passing the date as a string to the `Date()` constructor, the time portion is optional. If it is not included, it defaults to `00:00:00`. Also, other date formats are acceptable (e.g., `'6/21/1969'` and `'06-21-1969'`).
3. When passing date parts to the `Date()` constructor, `dd`, `hh`, `mm`, `ss`, and `ms` are all optional. The default for `dd` is 1; the other parameters default to 0.
4. Months are numbered from 0 (January) to 11 (December). In the example above, 6 represents July.

Some common date methods are shown below. In all the examples, the variable `moonLanding` contains the date `Mon Jul 21, 1969 16:18:00 GMT-0400 (Eastern Daylight Time)`.

Common Date Methods

Method	Description
getDate()	Returns the day of the month (1-31).
	<code>moonLanding.getDate();</code> <code>// returns 21</code>
getDay()	Returns the day of the week as a number (0-6, 0=Sunday, 6=Saturday).
	<code>moonLanding.getDay();</code> <code>// returns 1</code>
getMonth()	Returns the month as a number (0-11, 0=January, 11=December).
	<code>moonLanding.getMonth();</code> <code>// returns 6</code>
getFullYear()	Returns the four-digit year.
	<code>moonLanding.getFullYear();</code> <code>// returns 1969</code>
getHours()	Returns the hour (0-23).
	<code>moonLanding.getHours();</code> <code>// returns 16</code>
getMinutes()	Returns the minute (0-59).
	<code>moonLanding.getMinutes();</code> <code>// returns 18</code>
getSeconds()	Returns the second (0-59).
	<code>moonLanding.getSeconds();</code> <code>// returns 0</code>
getMilliseconds()	Returns the millisecond (0-999).
	<code>moonLanding.getMilliseconds();</code> <code>// returns 0</code>
getTime()	Returns the number of milliseconds since midnight January 1, 1970.
	<code>moonLanding.getTime();</code> <code>// returns -14096520000. It's negative, because it's before the epoch.</code>

Method	Description
getTimezoneOffset()	Returns the time difference in minutes between the user's computer and GMT.
	<pre>moonLanding.getTimezoneOffset(); // returns 240</pre>
toLocaleString()	Returns the Date object as a string.
	<pre>moonLanding.toLocaleString(); // returns '7/21/1969, 4:18:00 PM'</pre>
toLocaleDateString()	Returns the date portion of a Date object as a string.
	<pre>moonLanding.toLocaleDateString(); // returns '7/21/1969'</pre>
toLocaleTimeString()	Returns the time portion of a Date object as a string.
	<pre>moonLanding.toLocaleTimeString(); // returns '4:18:00 PM'</pre>
toGMTString()	Returns the Date object as a string in GMT timezone.
	<pre>moonLanding.toGMTString(); // returns 'Mon, 21 Jul 1969 20:18:00 GMT'</pre>

Below are the same methods from the table above shown in the Chrome DevTools Console:

```
> const moonLanding = new Date(1969, 6, 21, 16, 18, 0, 0);
< undefined
> moonLanding;
< Mon Jul 21 1969 16:18:00 GMT-0400 (Eastern Daylight Time)
> moonLanding.getDate();
< 21
> moonLanding.getDay();
< 1
> moonLanding.getMonth();
< 6
> moonLanding.getFullYear();
< 1969
> moonLanding.getHours();
< 16
> moonLanding.getMinutes();
< 18
> moonLanding.getSeconds();
< 0
> moonLanding.getMilliseconds();
< 0
> moonLanding.getTime();
< -14096520000
> moonLanding.getTimezoneOffset();
< 240
> moonLanding.toLocaleString();
< '7/21/1969, 4:18:00 PM'
> moonLanding.toLocaleDateString();
< '7/21/1969'
> moonLanding.toLocaleTimeString();
< '4:18:00 PM'
> moonLanding.toGMTString();
< 'Mon, 21 Jul 1969 20:18:00 GMT'
```

Date Documentation

See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date for full documentation on Date.

Let's see how we can use dates to build useful helper functions.

EVALUATION COPY: Not to be used in class.



4.4. Helper Functions

Some languages have functions that return the month as a string. JavaScript doesn't have such a built-in function. The following sample shows a user-defined "helper" function that handles this and how the `getMonth()` method of a `Date` object can be used to get the month.

Demo 4.1: BuiltInObjects/Demos/month-as-string.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.      function monthAsString(num) {
10.          const months = ["January", "February", "March", "April",
11.                          "May", "June", "July", "August", "September",
12.                          "October", "November", "December"];
13.          return months[num-1];
14.      }
15.
16.      function enterMonth() {
17.          const userMonth = prompt("What month were you born?", "");
18.          alert("You were born in " + monthAsString(userMonth) + ".");
19.      }
20.
21.      function getCurrentMonth() {
22.          const today = new Date();
23.          alert(monthAsString(today.getMonth()+1));
24.      }
25.  </script>
-----Lines 26 through 34 Omitted-----
```

Run this page in your browser and then click the buttons to see how they work.

Exercise 7: Returning the Day of the Week as a String

⌚ 15 to 25 minutes

In this exercise, you will create a function that returns the day of the week as a string.

1. Open `BuiltInObjects/Exercises/date-udfs.html` for editing.
2. Write a `dayAsString()` function that returns the day of the week as a string, with "1" returning "Sunday", "2" returning "Monday", etc.
3. Write an `enterDay()` function that prompts the user for the day of the week (as a number) and then alerts the string value of that day by calling the `dayAsString()` function.
4. Write a `getCurrentDay()` function that alerts today's actual day of the week according to the user's machine.
5. Add a **CHOOSE DAY** button that calls the `enterDay()` function.
6. Add a **GET CURRENT DAY** button that calls the `getCurrentDay()` function.
7. Test your solution in a browser.

Solution: BuiltInObjects/Solutions/date-udfs.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     function monthAsString(num) {
10.         const months = [];
11.         months[0] = "January";
12.         months[1] = "February";
13.         months[2] = "March";
14.         months[3] = "April";
15.         months[4] = "May";
16.         months[5] = "June";
17.         months[6] = "July";
18.         months[7] = "August";
19.         months[8] = "September";
20.         months[9] = "October";
21.         months[10] = "November";
22.         months[11] = "December";
23.
24.         return months[num-1];
25.     }
26.
27.     function dayAsString(num) {
28.         const weekdays = [];
29.         weekdays[0] = "Sunday";
30.         weekdays[1] = "Monday";
31.         weekdays[2] = "Tuesday";
32.         weekdays[3] = "Wednesday";
33.         weekdays[4] = "Thursday";
34.         weekdays[5] = "Friday";
35.         weekdays[6] = "Saturday";
36.
37.         return weekdays[num-1];
38.     }
39.
40.     function enterMonth() {
41.         const userMonth = prompt("What month were you born?", "");
42.         alert("You were born in " + monthAsString(userMonth) + ".");
43.     }
44.
```




```
45.     function getCurrentMonth() {
46.         const today = new Date();
47.         alert(monthAsString(today.getMonth()+1));
48.     }
49.
50.     function enterDay() {
51.         const userDay = prompt("What day of the week is it?", "");
52.         alert("Today is " + dayAsString(userDay) + ".");
53.     }
54.
55.     function getCurrentDay() {
56.         const today = new Date();
57.         alert(dayAsString(today.getDay()+1));
58.     }
59. </script>
60. <title>Date UDFs</title>
61. </head>
62. <body>
63. <main>
64.     <button onclick="enterMonth()">CHOOSE MONTH</button>
65.     <button onclick="getCurrentMonth()">GET CURRENT MONTH</button>
66.     <hr>
67.     <button onclick="enterDay()">CHOOSE DAY</button>
68.     <button onclick="getCurrentDay()">GET CURRENT DAY</button>
69. </main>
70. </body>
71. </html>
```

Conclusion

In this lesson, you have learned to work with some of JavaScript's most useful built-in objects.

LESSON 5

Conditionals and Loops

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☒ if - else if - else blocks.
- ☒ switch / case blocks.
- ☒ Loops.

Introduction

In this lesson, you will learn to branch your code using if and switch conditions, and to use different types of loops.

EVALUATION COPY: Not to be used in class.



5.1. Conditionals

There are two types of conditionals in JavaScript:

1. if - else if - else
2. switch / case

❖ 5.1.1. if - else if - else Conditions

```
if (conditions) {  
    statements;  
} else if (conditions) {  
    statements;  
} else {  
    statements;  
}
```

Like with functions, each part of the if - else if - else block is contained within curly brackets ({}). There can be zero or more else if blocks. The else block is optional.

Comparison Operators

Operator	Description
==	Equals
!=	Doesn't equal
===	Strictly equals
!==	Doesn't strictly equal
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to

Note the difference between == (equals) and === (strictly equals). For two objects to be **strictly equal** they must be of the same value **and** the same type, whereas to be **equal** they must only have the same value. See the code samples below:

```

> 0 == false;
< true
> 0 === false;
< false
> 5 == '5';
< true
> 5 === '5';
< false
> 0 == ''; // Both are falsy
< true
> 0 === '';
< false

```

Notice that 0 is equal to, but not *strictly equal to*, an empty string. Both these values are *falsy*, meaning that when they are treated as Booleans, they are considered to be false. More on this soon.

It is almost always better to use the strictly equals operator (===) and the corresponding doesn't strictly equal operator (!==) as these help avoid unanticipated errors.

Logical Operators

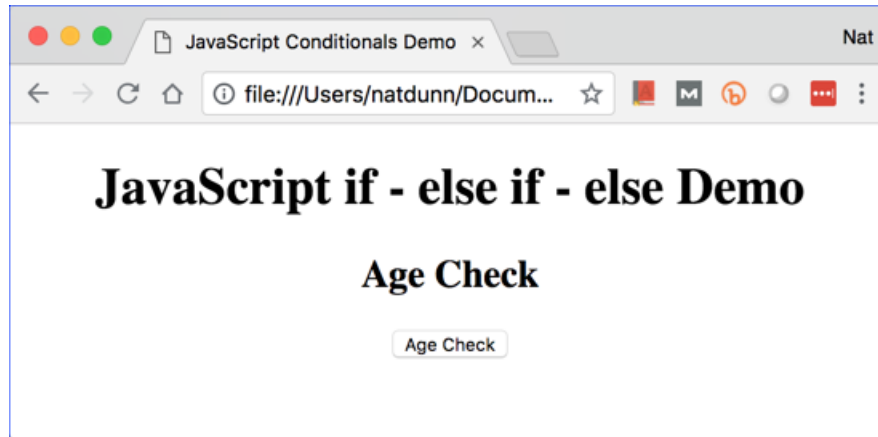
Operator	Description	Example
&&	and	(a == b && c != d)
	or	(a == b c != d)
!	not	!(a == b c == d)

The following example shows a function using an if - else if - else condition.

Demo 5.1: ConditionalsAndLoops/Demos/if-else-if-else.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      function checkAge() {
10.         const age = prompt("Your age?", "") || "";
11.
12.         if (age >= 21) {
13.             alert("You can vote and drink!");
14.         } else if (age >= 18) {
15.             alert("You can vote, but can't drink.");
16.         } else {
17.             alert("You cannot vote or drink.");
18.         }
19.     }
20. </script>
21. <title>JavaScript Conditionals Demo</title>
22. </head>
23. <body>
24. <main>
25.     <h1>JavaScript if - else if - else Demo</h1>
26.     <h2>Age Check</h2>
27.     <button onclick="checkAge()">Age Check</button>
28. </main>
29. </body>
30. </html>
```

The display of the page is shown below:

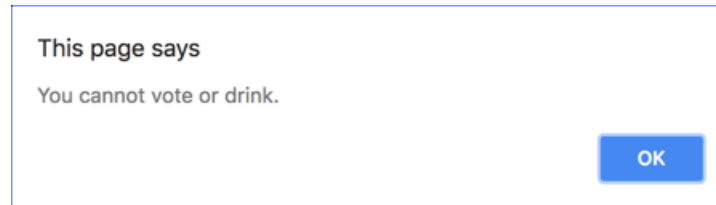


When the user clicks the **Age Check** button, the following prompt pops up:

A dialog box with a title 'This page says'. Below the title is the text 'Your age?'. There is a text input field with a cursor inside. At the bottom right are two buttons: 'Cancel' and 'OK'.

After the user enters their age, an alert pops up. The text of the alert depends on the user's age. The three possibilities are shown below:

An alert dialog box with a title 'This page says'. The text inside is 'You can vote and drink!'. There is a single 'OK' button at the bottom right.An alert dialog box with a title 'This page says'. The text inside is 'You can vote, but can't drink.'. There is a single 'OK' button at the bottom right.



Compound Conditions

Compound conditions are conditions that check for multiple things. See the following sample:

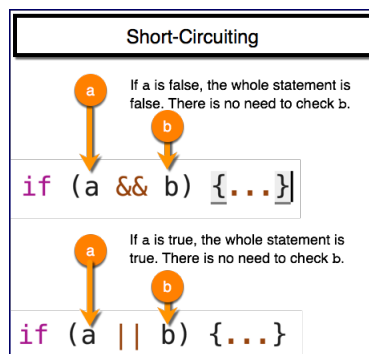
```
if (age > 18 && isCitizen) {  
  alert("You can vote!");  
}  
  
if (age >= 16 && (isCitizen || hasGreenCard)) {  
  alert("You can work in the United States");  
}
```

EVALUATION COPY: Not to be used in class.

EVALUATION COPY

5.2. Short-circuiting

JavaScript is lazy (or efficient) about processing compound conditions. As soon as it can determine the overall result of the compound condition, it stops looking at the remaining parts of the condition:



Short-circuiting is useful for checking that a variable is of the right data type before you try to manipulate it.

To illustrate, take a look at the following sample:

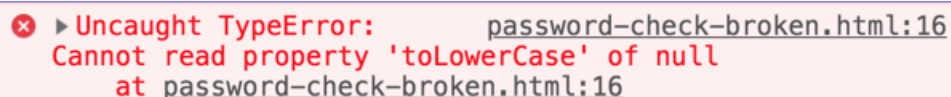
Demo 5.2: ConditionalsAndLoops/Demos/password-check-broken.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     const userPass = prompt("Password:", ""); //ESC here causes problems
10.    const pw = "xyz";
11. </script>
12. <title>Password Check</title>
13. </head>
14. <body>
15. <main>
16.     <script>
17.         if (userPass.toLowerCase() === pw) {
18.             document.write("<h1>Welcome!</h1>");
19.         } else {
20.             document.write("<h1>Bad Password!</h1>");
21.         }
22.     </script>
23. </main>
24. </body>
25. </html>
```

Everything works fine as long as the user does what you expect. However, if the user clicks the **Cancel** button when prompted for a password, the value `null` will be assigned to `userPass`. Because `null` is not a string, it does not have the `toLowerCase()` method. So the following line will result in a JavaScript error:

```
if (userPass.toLowerCase() === pw)
```

You can see the error in Chrome DevTools Console:



✖ ▶ Uncaught TypeError: password-check-broken.html:16
Cannot read property 'toLowerCase' of null
at password-check-broken.html:16

This can be fixed by using `typeof` (described below) to first check if `userPass` is a string as shown in the following sample:

The `typeof` Operator

The `typeof` operator is used to find out the type of a piece of data. The following screenshot shows what the `typeof` operator returns for different data types:



```
> typeof false;
< "boolean"
> typeof 5;
< "number"
> typeof 'hello';
< "string"
> typeof [];
< "object"
> typeof function() {};
< "function"
> typeof alert;
< "function"
> typeof window.document;
< "object"
> typeof null;
< "object"
> typeof undefined;
< "undefined"
> typeof foo; // We haven't defined this variable
< "undefined"
```

Demo 5.3: ConditionalsAndLoops/Demos/password-check.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      const userPass = prompt("Password:", "");
10.     const pw = "xyz";
11. </script>
12. <title>Password Check</title>
13. </head>
14. <body>
15. <main>
16.     <script>
17.         if (typeof userPass === "string" && userPass.toLowerCase() === pw) {
18.             document.write("<h1>Welcome!</h1>");
19.         } else {
20.             document.write("<h1>Bad Password!</h1>");
21.         }
22.     </script>
23. </main>
24. </body>
25. </html>
```

Now, if the user presses **Cancel** and `userPass` gets `null`, this check will fail: `typeof userPass === "string"`. Because the `if` condition uses `&&` requiring that both conditions are true for the whole statement to be true, there is no reason to check the second condition if the first condition is false. So, JavaScript short circuits, meaning it immediately returns `false` without wasting time checking the second condition.

Short circuiting also works with **or** conditions (e.g., `if (a or b)`). In this case, the whole statement is true if either side of the `or` condition is true. So, if `a` is true, there is no reason to check `b`. JavaScript will short circuit and return `true`.

EVALUATION COPY: Not to be used in class.



5.3. Switch / Case

```
switch (expression) {  
    case value :  
        statements;  
    case value :  
        statements;  
    default :  
        statements;  
}
```

Evaluation
Copy

Like if - else if - else statements, switch / case statements are used to run different code at different times. Unlike if statements, switch / case statements are limited to checking for equality. Each case is checked to see if the *expression* matches the *value*.

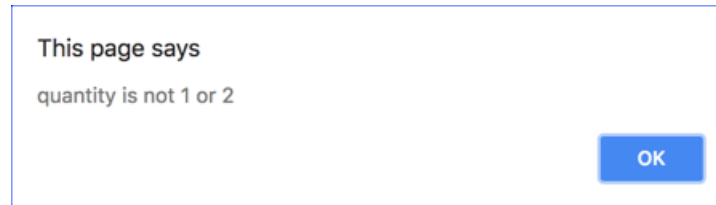
Take a look at the following example:

Demo 5.4: ConditionalsAndLoops/Demos/switch-without-break.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      const quantity = 1;
10.     switch (quantity) {
11.         case 1 :
12.             alert("quantity is 1");
13.         case 2 :
14.             alert("quantity is 2");
15.         default :
16.             alert("quantity is not 1 or 2");
17.     }
18. </script>
19. <title>Switch</title>
20. </head>
21. <body>
22. <main>
23.     <p>Nothing to show here.</p>
24. </main>
25. </body>
26. </html>
```

When you run this page in a browser, you'll see that all three alerts pop up, even though only the first case is a match:





That's because if a match is found, none of the remaining cases are checked and all the remaining statements in the switch block are executed. To stop this process, you can insert a `break` statement, which will end the processing of the switch statement.

The corrected code is shown in the following example:

Demo 5.5: ConditionalsAndLoops/Demos/switch-with-break.html


```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      const quantity = 1;
10.     switch (quantity) {
11.         case 1 :
12.             alert("quantity is 1");
13.             break;
14.         case 2 :
15.             alert("quantity is 2");
16.             break;
17.         default :
18.             alert("quantity is not 1 or 2");
19.     }
20. </script>
21. <title>Switch</title>
22. </head>
23. <body>
24. <main>
25.     <p>Nothing to show here.</p>
26. </main>
27. </body>
28. </html>
```

The following example shows how a switch / case statement can be used to decide what math operation to perform:

Evaluation
copy

Demo 5.6: ConditionalsAndLoops/Demos/do-math.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      function doMath(operator) {
10.         const n1 = parseFloat(document.getElementById('n1').value);
11.         const n2 = parseFloat(document.getElementById('n2').value);
12.         let result;
13.         switch (operator) {
14.             case "+":
15.                 result = n1 + n2;
16.                 break;
17.             case "-":
18.                 result = n1 - n2;
19.                 break;
20.             case "*":
21.                 result = n1 * n2;
22.                 break;
23.             case "/":
24.                 result = n1 / n2;
25.                 break;
26.             default:
27.                 alert("Bad operator");
28.         }
29.         alert(n1 + operator + n2 + '=' + result);
30.     }
31. </script>
32. <title>doMath</title>
33. </head>
34. <body>
35. <main>
36.     <label for="n1">First Number:</label> <input id="n1">
37.     <label for="n2">Second Number:</label> <input id="n2">
38.     <button onclick="doMath('+')">Add</button>
39.     <button onclick="doMath('-')">Subtract</button>
40.     <button onclick="doMath('*')">Multiply</button>
41.     <button onclick="doMath('/')">Divide</button>
42. </main>
43. </body>
44. </html>
```



Use Case for switch Without break

In most cases, you will include `break` statements in your `switch` conditions; however, there are cases when it makes sense to continue to execute all the subsequent statements in a `switch` condition after a match has been found. Consider the following, in which permissions are being added to an array based on a user's role:

```
const role = 'Admin';
const permissions = [];
switch (role) {
  case 'SuperAdmin':
    permissions.push('delete');
  case 'Admin':
    permissions.push('update');
  case 'Contributor':
    permissions.push('create');
  default:
    permissions.push('read');
}
console.log(permissions);
```

The code above will log (3) ['update', 'create', 'read'] to the console. That's because `role` is set to 'Admin'. The logic works as follows:

1. Does `role` contain 'SuperAdmin'? No, it does not. So, it doesn't push 'delete' onto the `permissions` array.
2. Does `role` contain 'Admin'? Yes, it does. So, it pushes 'update' onto the `permissions` array.
3. Then, it stops checking the cases, because it already found the match. And it continues executing all the statements until it finds a `break` or it reaches the end of the `switch` statement. In this case, there are no `break` statements, so it pushes 'create' and 'read' onto the `permissions` array.

The result is that SuperAdmin will get all permissions. Admin will get *update*, *create*, and *read* permissions. Contributor will get *create* and *read* permissions. All others will only get *read* permissions.

Order of Conditions

In conditional statements it's generally a good practice to test for the most likely cases/matches first so the browser can find the correct code to execute more quickly.



5.4. Ternary Operator

The ternary operator provides a shortcut for if conditions. The syntax is as follows:

```
const constName = (condition) ? valueIfTrue : valueIfFalse;
```

For example:

```
const evenOrOdd = (number % 2 === 0) ? "even" : "odd";
```

The following code sample shows how the ternary operator works:

Demo 5.7: ConditionalsAndLoops/Demos/ternary.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.    const num = parseInt(prompt("Enter a number.", ""));
10.
11.    //without ternary
12.    if (num % 2 === 0) {
13.        alert(num + " is even.");
14.    } else {
15.        alert(num + " is odd.");
16.    }
17.
18.    //with ternary
19.    const term = num % 2 === 0 ? "even" : "odd";
20.    alert(num + " is " + term);
21. </script>
-----Lines 22 through 29 Omitted-----
```

The first block shows a regular if-else statement.

The second block shows how to accomplish the same thing in a couple of lines of code with the ternary operator.



5.5. Truthy and Falsy

JavaScript has a boolean data type, which has only two possible values: `true` or `false`. In addition, every value and expression in JavaScript can be converted to `true` or `false`.

When a non-boolean literal value, variable, or expressions is used in a boolean context (e.g, an if condition or with the *default* operator), it is implicitly converted to a boolean. This process is called *Type Coercion*. For example, look at the following code, which uses the default operator:

```
const a = 1 || 2;
```

The value `1` is interpreted as `true`, so `a` will get `1`. Non-boolean values that are treated as `true` when used in a boolean context are said to be *truthy*.

Now examine the following code:

```
const a = 0 || 2;
```

The value `0` is interpreted as `false`, so `a` will get `2`. Non-boolean values that are treated as `false` when used in a boolean context are said to be *falsy*.

The only *falsy* values are:

1. `0`, but not `"0"`, which is a string.
2. `""` – a zero-length string.
3. `null`
4. `undefined`
5. `NaN` – a special number value that means “Not a Number”. For example, `NaN` is the result of dividing `0` by `0` or finding the square root of a negative number (e.g. `Math.sqrt(-1)`).

All other values are *truthy*.



Exercise 8: Conditional Processing

⌚ 20 to 30 minutes

In this exercise, you will practice using conditional processing.

1. Open `ConditionalsAndLoops/Exercises/conditionals.html` for editing.
2. Notice that there is an `onclick` event handler on the button that calls the `greetUser()` function. Create this function in the script block.
3. The function should do the following:
 - A. Ask (via a prompt) if the user is right- or left-handed.
 - B. If the user enters a value other than “right” or “left”, prompt again.
 - C. Ask (via a prompt) for the user’s last name.
 - D. If the user leaves the last name blank, prompt again.
 - E. If the user enters a number for the last name, alert that a last name can’t be a number and prompt again.
 - F. After collecting the user’s dominant hand and last name:
 - If the dominant hand is valid, pop up an alert that greets the user appropriately (e.g., “Hello Lefty Smith!”)
 - If the dominant hand is not valid, pop up an alert that reads something like “XYZ is not a valid value for dominant hand!”
4. Test your solution in a browser.

Challenge

1. Allow the user to enter the dominant hand in any case (e.g., left, Left, LEFT, right, Right, RIGHT).
2. If the user enters a last name that does not start with a capital letter, prompt to try again.

Solution: ConditionalsAndLoops/Solutions/conditionals.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.  function greetUser() {
10.    let dominantHand;
11.    let lastName;
12.
13.    dominantHand = prompt("Are you left- or right-handed?", "") || "";
14.    if (dominantHand !== "right" && dominantHand !== "left") {
15.      dominantHand = prompt("Try again: right or left?", "") || "";
16.    }
17.
18.    lastName = prompt("What's your last name?", "") || "";
19.    if (lastName.length === 0) {
20.      lastName = prompt("No last name? Please re-enter:", "") || "";
21.    } else if (!isNaN(lastName)) {
22.      lastName = prompt("Names aren't numbers. Re-enter:", "") || "";
23.    }
24.
25.    switch (dominantHand) {
26.      case "right" :
27.        alert("Hello Righty " + lastName + "!");
28.        break;
29.      case "left" :
30.        alert("Hello Lefty " + lastName + "!");
31.        break;
32.      default :
33.        alert(dominantHand + " is not a valid value for dominant hand!");
34.    }
35.  }
36. </script>
-----Lines 37 through 44 Omitted-----
```

Challenge Solution:

ConditionalsAndLoops/Solutions/conditionals-challenge.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.  function greetUser() {
10.    let dominantHand;
11.    let lastName;
12.
13.    dominantHand = prompt("Are you left- or right-handed?", "") || "";
14.    dominantHand = dominantHand.toLowerCase();
15.    if (dominantHand !== "right" && dominantHand !== "left") {
16.      dominantHand = prompt("Try again: right or left?", "") || "";
17.    }
18.
19.    lastName = prompt("What's your last name?", "") || "";
20.    const firstLetter = lastName.substring(0, 1);
21.    if (lastName.length === 0) {
22.      lastName = prompt("No last name? Please re-enter:", "") || "";
23.    } else if (!isNaN(lastName)) {
24.      lastName = prompt("Names aren't numbers. Re-enter:", "") || "";
25.    } else if (firstLetter === firstLetter.toLowerCase()) {
26.      lastName = prompt("Names begin with capital letters. Re-enter:", "") || "";
27.    }
28.
29.    switch (dominantHand) {
30.      case "right" :
31.        alert("Hello Righty " + lastName + "!");
32.        break;
33.      case "left" :
34.        alert("Hello Lefty " + lastName + "!");
35.        break;
36.      default :
37.        alert(dominantHand + " is not a valid value for dominant hand!");
38.    }
39.  }
40. </script>
-----Lines 41 through 48 Omitted-----
```

EVALUATION COPY: Not to be used in class.



5.6. Loops

There are several types of loops in JavaScript:

- while
- do...while
- for
- for...in
- for...of

EVALUATION COPY: Not to be used in class.



5.7. while and do...while Loops

❖ 5.7.1. while Loop Syntax

```
while (conditions) {  
  statements;  
}
```

The while loop first checks one or more conditions and then executes the statements in its body as long as those conditions are true. Something, usually a statement within the while block, must cause the condition to change so that it eventually becomes false and causes the loop to end. Otherwise, you get stuck in an infinite loop, which can bring down the browser.

Here is an example of a while loop:

```
let i=0;  
while (i < 5) {  
  console.log(i);  
  i++; // changing value of i  
}
```

And here's the above code executed at Chrome DevTools Console:


```
> let i = 0;
  while (i < 5) {
    console.log(i);
    i++;
  }
0
1
2
3
4
```

❖ 5.7.2. do...while Loop Syntax

```
do {
  statements;
} while (conditions);
```

The **do...while** loop checks the conditions *after* each execution of the statements in the body. Again, something, usually a statement within the do block, must cause the condition to change so that it eventually becomes false and causes the loop to end.

Here is an example of a do...while loop:

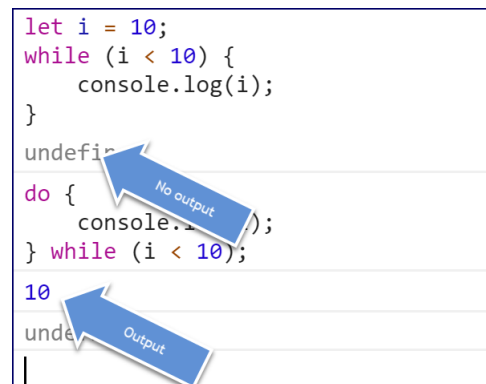
```
let i=0;
do {
  console.log(i);
  i++; // changing value of i
} while (i < 5);
```

And here's the above code executed at Chrome DevTools Console:

```
> let i = 0;
  do {
    console.log(i);
    i++;
  } while (i < 5);
0
1
2
3
4
```

Unlike with while loops, the statements in do...while loops will always execute at least one time because the conditions are not checked until the end of each iteration. The following code illustrates this:

```
let i = 10;
while (i < 10) {
  console.log(i);
}
undefined
do {
  console.log(i);
} while (i < 10);
10
undefined
|
```



EVALUATION COPY: Not to be used in class.



5.8. for Loops

❖ 5.8.1. for Loop Syntax

```
for (initialization; conditions; change) {
  statements;
}
```

In for loops, the initialization, conditions, and change are all placed up front and separated by semi-colons. This makes it easy to remember to include a change statement that will eventually cause the loop to end.

for loops are often used to iterate through arrays. The length property of an array can be used to check how many elements the array contains. For example:

```
const fruit = ['Apples', 'Oranges', 'Bananas', 'Pears'];
for (let i=0; i<fruit.length; i++) {
  console.log(fruit[i]);
}
```

And here's the above code executed at Chrome DevTools Console:

```
> const fruit = ['Apples', 'Oranges', 'Bananas', 'Pears'];  
for (let i=0; i < fruit.length; i++) {  
  console.log(fruit[i]);  
}
```

Apples
Oranges
Bananas
Pears

❖ 5.8.2. for...of Loop Syntax

```
for (let item of iterable) {  
  statement;  
}
```

for...of loops are used to loop through any *iterable object* – usually arrays, but there are other types of iterable objects as well. For example:

```
const fruit = ['Apples', 'Oranges', 'Bananas', 'Pears'];  
for (let i of fruit) {  
  console.log(i);  
}
```

And here's the above code executed at Chrome DevTools Console:

```
> const fruit = ['Apples', 'Oranges', 'Bananas', 'Pears'];  
for (let i of fruit) {  
  console.log(i);  
}
```

Apples
Oranges
Bananas
Pears

❖ 5.8.3. for...in Loop Syntax

```
for (let item in object) {  
  statements;  
}
```

for...in loops are used to loop through object properties. A common mistake is to use this type of loop to iterate through arrays. Most of the time, this will work fine, but for reasons that are beyond the scope of this course, you should avoid using for...in loops to iterate through arrays. We cover the syntax here only because you are likely to see this type of loop used incorrectly and we want you to be able to recognize it. If you would like to learn more why it should be avoided, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#Arrays.

EVALUATION COPY: Not to be used in class.

Evaluation
Copy

5.9. break and continue

The break statement is used to break out of a loop, usually when some condition is met.

```
for (let item of object) {  
  doSomething(item);  
  if (conditions) {  
    break;  
    // loop will stop executing  
    // and afterLoop() will run  
  }  
}  
afterLoop();
```

The following code illustrates how break works:

```
> const fruit = ['Apples', 'Oranges', 'Bananas', 'Pears'];
  for (let i of fruit) {
    console.log(i);
    if (i.indexOf('an') >= 0) {
      break;
    }
  }
Apples
Oranges
```

Notice that the Bananas and Pears do not get logged, because the loop is broken as soon as Oranges, which contains “an” is found.

The `continue` statement is used to move on to the next iteration of the loop. It is used when a condition is met that makes it unnecessary to run the rest of the code in the loop body for that iteration.

```
for (let item of object) {
  doSomething(item);
  if (conditions) {
    continue;
    // loop will move on to next item
    // doSomethingElse() won't be executed for this item
  }
  doSomethingElse(item);
}
```

The following code illustrates how `continue` works:

```
> const fruit = ['Apples', 'Oranges', 'Bananas', 'Pears'];
  for (let i of fruit) {
    if (i.indexOf('an') >= 0) {
      continue;
    }
    console.log(i);
  }
Apples
Pears
```

Notice that the Oranges and Bananas do not get logged, because both contain “an”, and when that condition is met, the loop moves on to the next iteration.



Exercise 9: Working with Loops

⌚ 20 to 30 minutes

In this exercise, you will practice working with loops.

1. Open `ConditionalsAndLoops/Exercises/loops.html` for editing. You will see that this file is similar to the solution to the challenge from the last exercise.
2. Declare an additional variable called `greeting`.
3. Create an array called `presidents` that contains the last names of four or more past presidents.
4. Currently, the user only gets two tries to enter a valid `dominantHand` and `lastName`. Modify the code so that, in both cases, the user continues to get prompted until the data is valid.
 - A. For `dominantHand`, the first prompt should be “Are you left- or right-handed?” Each subsequent prompt should be “Try again: right or left?”
 - B. For `lastName`, it should just continue prompting “What’s your last name?” until the user enters a valid last name.
5. Change the `switch` block so that it assigns an appropriate value (e.g., “Hello Lefty Smith”) to the `greeting` variable rather than popping up an alert.
6. After the `switch` block, write code that alerts the user by name if they have the same last name as a president. There is no need to alert those people who have non-presidential names.

Challenge

1. For those people who do not have presidential names, pop up an alert that tells them their names are not presidential.

Solution: ConditionalsAndLoops/Solutions/loops.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.  function greetUser() {
10.    let dominantHand;
11.    let lastName;
12.    let greeting;
13.    const presidents = ["Washington", "Jefferson", "Lincoln", "Kennedy"];
14.
15.    dominantHand = prompt("Are you left- or right-handed?", "") || "";
16.    dominantHand = dominantHand.toLowerCase();
17.    while (dominantHand !== "right" && dominantHand !== "left") {
18.      dominantHand = prompt("Try again: right or left?", "") || "";
19.    }
20.
21.    do {
22.      lastName = prompt("What's your last name?", "") || "";
23.    } while (lastName.length === 0
24.      || !isNaN(lastName)
25.      || lastName.substring(0, 1) === lastName.substring(0, 1).toLowerCase())
26.
27.    switch (dominantHand) {
28.      case "right" :
29.        greeting = "Hello Righty " + lastName + "!";
30.        break;
31.      default : // If not right, must be left
32.        greeting = "Hello Lefty " + lastName + "!";
33.    }
34.
35.    for (let lName of presidents) {
36.      if (lName === lastName) {
37.        alert(greeting + ' Your name is presidential!');
38.        break; // No need to keep looking after we've found a match
39.      }
40.    }
41.  }
42. </script>
-----Lines 43 through 50 Omitted-----
```

Challenge Solution:

ConditionalsAndLoops/Solutions/loops-challenge.html

```
-----Lines 1 through 34 Omitted-----  
35.     let match = false;  
36.     for (let lName of presidents) {  
37.         if (lName === lastName) {  
38.             alert(greeting + ' Your name is presidential!');  
39.             match = true;  
40.             break; // No need to keep looking after we've found a match  
41.         }  
42.     }  
43.     if (!match) {  
44.         alert(greeting + ' Your name is not presidential!');  
45.     }  
-----Lines 46 through 55 Omitted-----
```

EVALUATION COPY: Not to be used in class.

Evaluation
Copy

5.10. Array: forEach()

Another way to loop through arrays is to use the array's built-in `forEach()` method.

```
myArray.forEach( function(item) {  
    doSomething(item);  
});
```

Each item of the array is passed to the function one by one. For example:

```
const fruit = ['Apples', 'Oranges', 'Bananas', 'Pears'];  
fruit.forEach(function(item) {  
    console.log(item);  
});
```

And here's the above code executed at Chrome DevTools Console:

```
> const fruit = ['Apples', 'Oranges', 'Bananas', 'Pears'];  
fruit.forEach( function(item) {  
    console.log(item);  
});
```

Apples

Oranges

Bananas

Pears

Conclusion

In this lesson, you learned:

- To work with if-else if-else conditions.
- To work with switch / case conditionals.
- To work with several types of loops.

LESSON 6

Event Handlers and Listeners

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☒ Understanding on-event handlers.
- ☒ Commonly-used on-event handlers.
- ☒ `addEventListener()`.
- ☒ Benefits of event listeners.

Introduction

On-event handlers allow us to listen for user actions and to respond to those events with custom code.

EVALUATION COPY: Not to be used in class.



6.1. On-event Handlers

On-event handlers are attributes that force an element to “listen” for a specific event to occur.

We might, for instance, listen for a user to click a specific `div` element, listen for a form submission, or listen for the user to pass their mouse over any `input` element of a given class.

The table below lists commonly-used HTML on-event handlers with descriptions:

HTML On-event Handlers

On-event Handler	Description
onblur	The element lost the focus.
onchange	The element value was changed.
onclick	A pointer button was clicked.
ondblclick	A pointer button was double-clicked.
onfocus	The element received the focus.
onkeydown	A key was pressed down.
onkeypress	A key was pressed and released.
onkeyup	A key was released.
onload	The document has been loaded.
onmousedown	A pointer button was pressed down.
onmousemove	A pointer was moved within the element.
onmouseout	A pointer was moved off of the element.
onmouseover	A pointer was moved onto the element.
onmouseup	A pointer button was released over the element.
onreset	The form was reset.
onselect	Some text was selected.
onsubmit	The form was submitted.

❖ 6.1.1. The getElementById() Method

A very common way to reference HTML elements is by their id using the `getElementById()` method of the document object as shown in the following example. Once we have the element – that is, once we get a given `div`, `p`, `input` or other DOM element via the `getElementById()` method – we can then listen for events on that element. Let's look at an example:

Demo 6.1: EventHandlers/Demos/get-element-by-id.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.  function changeBg(id, color) {
10.    document.getElementById(id).style.backgroundColor = color;
11.  }
12. </script>
13. <title>getElementById()</title>
14. </head>
15. <body>
16. <main>
17.   <button onclick="changeBg('divRed','red')">Red</button>
18.   <button onclick="changeBg('divOrange','orange')">Orange</button>
19.   <button onclick="changeBg('divGreen','green')">Green</button>
20.   <button onclick="changeBg('divBlue','blue')">Blue</button>
21.   <div id="divRed">Red</div>
22.   <div id="divOrange">Orange</div>
23.   <div id="divGreen">Green</div>
24.   <div id="divBlue">Blue</div>
25. </main>
26. </body>
27. </html>
```

Clicking the buttons sets the style of the corresponding `div` element, whose `id` is gotten via a call to `getElementById()` in the `changeBg()` function.



Exercise 10: Using On-event Handlers

🕒 15 to 25 minutes

In this exercise, you will use on-event handlers to allow the user to change the background color of the page.

1. Open EventHandlers/Exercises/color-changer.html for editing.
2. Modify the page so that...
 - When the “Red” button is *clicked*, the background color turns red.
 - When the “Green” button is *double-clicked*, the background color turns green.
 - When the “Orange” button is *clicked down*, the background color turns orange and when the button is released (onmouseup), the background color turns white.
 - When the mouse hovers over the “pink” link, the background color turns pink. When it hovers off, the background color turns white.

Exercise Code 10.1: EventHandlers/Exercises/color-changer.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <title>Color Changer</title>
9. </head>
10. <body>
11. <main>
12.   <button>
13.     Click to turn the page red.
14.   </button>
15.   <button>
16.     Double-click to turn the page green.
17.   </button>
18.   <button>
19.     Click and hold to turn the page orange.
20.   </button>
21.   <a href="#">Hover over to turn page pink.</a>
22. </main>
23. </body>
24. </html>
```

Challenge

1. Add functionality so that when the user presses any key, the background color turns white.

Solution: EventHandlers/Solutions/color-changer.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.  function changeBg(color) {
10.     document.body.style.backgroundColor = color;
11.  }
12. </script>
13. <title>Color Changer</title>
14. </head>
15. <body>
16. <main>
17.   <button onclick="changeBg('red')">
18.     Click to turn the page red.
19.   </button>
20.   <button ondblclick="changeBg('green')">
21.     Double-click to turn the page green.
22.   </button>
23.   <button onmousedown="changeBg('orange')"
24.     onmouseup="changeBg('white')">
25.     Click and hold to turn the page orange.
26.   </button>
27.   <a href="#"
28.     onmouseover="changeBg('pink')"
29.     onmouseout="changeBg('white')">Hover over to turn page pink.</a>
30. </main>
31. </body>
32. </html>
```

Challenge Solution:

EventHandlers/Solutions/color-changer-challenge.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.  function changeBg(color) {
10.    document.body.style.backgroundColor = color;
11.  }
12. </script>
13. <title>Color Changer</title>
14. </head>
15. <body onkeypress="changeBg('white')">
16.   <main>
17.     <button onclick="changeBg('red')">
18.       Click to turn the page red.
19.     </button>
20.     <button ondblclick="changeBg('green')">
21.       Double-click to turn the page green.
22.     </button>
23.     <button onmousedown="changeBg('orange')"
24.       onmouseup="changeBg('white')">
25.       Click and hold to turn the page orange.
26.     </button>
27.     <a href="#" onmouseover="changeBg('pink')"
28.       onmouseout="changeBg('white')">Hover over to turn page pink.</a>
29.   </main>
30. </body>
31. </html>
```

EVALUATION COPY: Not to be used in class.



6.2. The addEventListener() Method

You have learned how to add *event handlers* using the on-event HTML attributes (e.g., onload, onclick, etc). Now, you will learn how to add *event listeners* using an EventTarget's addEventListener() method.

An EventListener represents an object that does something when an event occurs. Think of a swimmer on a block, waiting for the starting gun to go off. When the gun goes off, the swimmer dives. Here is some pseudo-code to set that up in JavaScript:

```
diver.addEventListener('shotFire', dive);
```

In the pseudo-code above, `diver` is the EventTarget, `shotFire` is the event type, and `dive` is the function that will be called when the event occurs. Functions that are called in response to an event are known as *callback functions*.


An EventTarget is any object on which an event can occur, including window, document, and any HTML element. The basic syntax is as follows:

```
object.addEventListener(eventType, callbackFunction);
```

We have already seen the different types of events: click, dblclick, load, mouseover, mouseout, etc. HTML attributes used to call these events all begin with “on”, but when referencing the event type directly, you do not include the “on” prefix. For example, the following code shows how to call the `init()` function when the load event of the window object occurs:

Demo 6.2: EventHandlers/Demos/window-load.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      function init(e) {
10.         alert('Hello, world!');
11.     }
12.     window.addEventListener('load', init);
13. </script>
14. <title>window load</title>
15. </head>
16. <body>
17. <main>
18.     <p>Nothing to show here.</p>
19. </main>
20. </body>
21. </html>
```



Run this in the browser and you will see the “Hello, world!” alert as soon as the page is finished loading.

Notice in the code above that `init` is passed to `addEventListener()` without the usual trailing parentheses associated with functions. It is `window.addEventListener('load', init);` and not `window.addEventListener('load', init());` The reason is that we are not *calling* the function at this point in the code. Rather, we are indicating that we want the function to be called when the relevant event occurs. If you make the mistake of including the parentheses, the function will be called immediately and the value returned from the function will be used as the callback function, probably resulting in an error.

The table below lists common event types with descriptions. These correspond to the on-event handlers we saw earlier.

Event Types

Event Type	Description
blur	The element lost the focus.
change	The element value was changed.
click	A pointer button was clicked.
dblclick	A pointer button was double-clicked.
focus	The element received the focus.
keydown	A key was pressed down.
keyup	A key was released.
load	The document has been loaded.
mousedown	A pointer button was pressed down.
mousemove	A pointer was moved within the element.
mouseout	A pointer was moved off of the element.
mouseover	A pointer was moved onto the element.
mouseup	A pointer button was released over the element.
reset	The form was reset.
select	Some text was selected.
submit	The form was submitted.

The Callback Function

In the example above, the callback function is `init(e)`. You may have noticed that it takes a single parameter, which we have called `e`, but the variable name is arbitrary. Common names are `e` and `evt`. This parameter will hold the *event* that caused the callback function to be called. Examine the following:

Demo 6.3: EventHandlers/Demos/window-load-e.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      function init(e) {
10.         alert(e);
11.         alert(e.currentTarget);
12.         alert(e.type);
13.     }
14.     window.addEventListener('load', init);
15. </script>
16. <title>window load</title>
17. </head>
18. <body>
19. <main>
20.     <p>Nothing to show here.</p>
21. </main>
22. </body>
23. </html>
```

This time, instead of alerting “Hello, world!”, the code alerts [object Event]:



and then alerts the `currentTarget` property of the event, which is the object that caused the event to occur: [object Window]:



Finally, it alerts the type of event: load:



Now let's take a look at how we use this passing of the event to make a function's response dependent on the event that spawned it:

Demo 6.4: EventHandlers/Demos/current-target.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      function changeBg(e) {
10.         const color = e.currentTarget.id;
11.         document.body.style.backgroundColor = color;
12.     }
13.
14.     function init(e) {
15.         const aqua = document.getElementById('aqua');
16.         const lime = document.getElementById('lime');
17.         const pink = document.getElementById('pink');
18.         aqua.addEventListener('click', changeBg);
19.         lime.addEventListener('click', changeBg);
20.         pink.addEventListener('click', changeBg);
21.     }
22.     window.addEventListener('load', init);
23. </script>
24. <title>window load</title>
25. </head>
26. <body>
27. <main>
28.     <button id="aqua">Aqua</button>
29.     <button id="lime">Lime</button>
30.     <button id="pink">Pink</button>
31. </main>
32. </body>
33. </html>
```

Run this page in your browser to see how it works.

1. When the page is loaded the `init()` function is called. It adds event listeners to each of the buttons, all with the same callback function: `changeBg`. Note that we have to add these event listeners **after** the document loads to be sure that the buttons exist. That is why we do it in the callback function of window's load event.

2. The callback function, `changeBg()`, sets the `color` variable to the value of the `id` of the event's `currentTarget` – the button that was clicked. It then changes the background color to `color`.

EVALUATION COPY: Not to be used in class.



6.3. Anonymous Functions

The `init()` function in the previous example is meant to be called once and only once – when the page finishes loading. As such, there is no reason for it to remain available after it is run. Such functions are often created as *anonymous functions* at the point in the code that they are needed. The syntax is as follows:

```
object.addEventListener(eventType, function(e) {  
  // function code here  
});
```

Notice the function has no name: `function init(e)` is replaced with `function(e)`. It doesn't need a name, because it will only be referenced this one time in the code.

Here is the last page rewritten to use an anonymous function:

Demo 6.5: EventHandlers/Demos/anonymous-function.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      function changeBg(e) {
10.         const color = e.currentTarget.id;
11.         document.body.style.backgroundColor = color;
12.     }
13.
14.     window.addEventListener('load', function(e) {
15.         const aqua = document.getElementById('aqua');
16.         const lime = document.getElementById('lime');
17.         const pink = document.getElementById('pink');
18.         aqua.addEventListener('click', changeBg);
19.         lime.addEventListener('click', changeBg);
20.         pink.addEventListener('click', changeBg);
21.     });
22. </script>
23. <title>Anonymous Function</title>
24. </head>
25. <body>
26. <main>
27.     <button id="aqua">Aqua</button>
28.     <button id="lime">Lime</button>
29.     <button id="pink">Pink</button>
30. </main>
31. </body>
32. </html>
```

Run this page in your browser and you'll see that it works the same as it did with a named function.

Note that we could make `changeBg()` an anonymous function as well, but because it is called three times, we would have to change it each place it is called. If we ever wanted to make modifications in the future, we would have to make those modifications in all three places. So, as it is reused, it makes more sense to give that one a name.



6.4. Capturing Key Events

The two types of keyboard events are:

1. `keydown` – fires when a key is pressed down.
2. `keyup` – fires when a key is released.

keypress

You may also see the `keypress` event, which fires when a key is pressed and then released. However, this event has been deprecated⁸ and is no longer recommended.

The target of keyboard events can be the document or any element on the page.

When capturing a keyboard event, it is common to want to know what key is pressed. This is available via the event's `key` property.

8. https://developer.mozilla.org/en-US/docs/Web/API/Element/keypress_event.

Demo 6.6: EventHandlers/Demos/keys.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     document.addEventListener('keyup', function(e) {
10.         document.getElementById('keyholder').innerHTML = e.key;
11.     });
12. </script>
13. <title>Key Press</title>
14. </head>
15. <body>
16. <main id="keyholder"></main>
17. </body>
18. </html>
```

Run this page in your browser and press any key to see how it works. Notice that when you press the **Enter** key, the word “Enter” is output. You could use the following code to capture this on an input field:

```
const myInput = document.getElementById('myInput');
myInput.addEventListener('keyup', function(e) {
    if (e.key === 'Enter') {
        doSomething();
    }
});
```

innerHTML

This demo uses the `innerHTML` property, which you can use to read and modify the HTML content of an element.



Exercise 11: Adding Event Listeners

⌚ 15 to 25 minutes

You will start with the following code:

Exercise Code 11.1: EventHandlers/Exercises/add-event-listener.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     // write changeBg function here
10.
11.     function changeBgWhite(e) {
12.         document.body.style.backgroundColor = 'white';
13.     }
14.
15.     // add your event listener here
16. </script>
17. <title>Color Changer</title>
18. </head>
19. <body>
20. <main>
21.     <button id="red">
22.         Click to turn the page red.
23.     </button>
24.     <button id="green">
25.         Double-click to turn the page green.
26.     </button>
27.     <button id="orange">
28.         Click and hold to turn the page orange.
29.     </button>
30.     <a href="#" id="pink">Hover over to turn page pink.</a>
31. </main>
32. </body>
33. </html>
```

1. Open EventHandlers/Exercises/add-event-listener.html in your editor.

2. Add an event listener to capture the load event of the window object. The callback function should be anonymous and should do the following:
 - A. Create variables holding the buttons and link.
 - B. Add a click event to the red button that calls changeBg.
 - C. Add a dblclick event to the green button that calls changeBg.
 - D. Add a mousedown event to the orange button that calls changeBg.
 - E. Add a mouseup event to the orange button that calls changeBgWhite.
 - F. Add a mouseover event to the link that calls changeBg.
 - G. Add a mouseout event to the link that calls changeBgWhite.
 - H. Add a keyup event to the document object that calls changeBgWhite.
3. Write the changeBg() function.

Challenge

1. Change the changeBgWhite() function as follows:

```
function changeBgWhite(e) {  
    changeBg('white');  
}
```

2. Change the changeBg() function to allow for a color value as a string as well as an event. If an event is passed in, it should get the color from the id of the currentTarget of the event as it does now. But if a string is passed in, it should use that string as the color value.

Solution: EventHandlers/Solutions/add-event-listener.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.    function changeBg(e) {
10.      const color = e.currentTarget.id;
11.      document.body.style.backgroundColor = color;
12.    }
13.
14.    function changeBgWhite(e) {
15.      document.body.style.backgroundColor = 'white';
16.    }
17.
18.    window.addEventListener('load', function() {
19.      const btnRed = document.getElementById('red');
20.      const btnGreen = document.getElementById('green');
21.      const btnOrange = document.getElementById('orange');
22.      const lnkPink = document.getElementById('pink');
23.
24.      btnRed.addEventListener('click', changeBg);
25.      btnGreen.addEventListener('dblclick', changeBg);
26.      btnOrange.addEventListener('mousedown', changeBg);
27.      btnOrange.addEventListener('mouseup', changeBgWhite);
28.      lnkPink.addEventListener('mouseover', changeBg);
29.      lnkPink.addEventListener('mouseout', changeBgWhite);
30.
31.      document.addEventListener('keyup', changeBgWhite);
32.    });
33. </script>
-----Lines 34 through 50 Omitted-----
```

Code Explanation

We need a `changeBgWhite()` function because we cannot key off the `id` value to change the background color to white for two reasons:

1. We have added two event handlers to the `btnOrange` button: `mousedown` and `mouseup`. For `mouseDown`, we call `changeBg()`, which keys off `btnOrange`'s `id` attribute ("orange") to change the background color to orange. For `mouseup` though, we want to change the background color to white, so we cannot call `changeBg()` again as that sets the color to the button's `id` value. That's why we need `changeBgWhite()`. The same logic applies to the `lnkPink` link.

2. The document object doesn't have an id value, so for keyup events, if we call `changeBg()`, the `e.currentTarget.id` value would be null. That's why we call `changeBgWhite()` instead.

Challenge Solution:

EventHandlers/Solutions/add-event-listener-challenge.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.      function changeBg(colorOrEvent) {
10.         let color = 'white'; // default
11.         if ( typeof colorOrEvent === 'string' ) {
12.             color = colorOrEvent;
13.         } else {
14.             color = colorOrEvent.currentTarget.id;
15.         }
16.         document.body.style.backgroundColor = color;
17.     }
18.
19.     function changeBgWhite(e) {
20.         changeBg('white');
21.     }
-----Lines 22 through 55 Omitted-----
```

EVALUATION COPY: Not to be used in class.



6.5. Benefits of Event Listeners

Using on-event handlers such as `onclick` and `onmouseover` is simple and straightforward, while using event listeners requires more JavaScript to set things up, so why use event listeners?

There are at least two major benefits to using event listeners:

1. You can add multiple event listeners to the same element.
2. Your HTML and JavaScript code are decoupled, which provides for easier maintenance and debugging.

To illustrate, take a look at the following JavaScript file:

Demo 6.7: EventHandlers/Demos/benefits.js

```
1.  function color() {
2.    document.body.style.backgroundColor = 'red';
3.  }
4.
5.  function reset() {
6.    document.body.style.backgroundColor = 'white';
7.  }
8.
9.  function log(e) {
10.   const t = e.currentTarget;
11.   console.log(t.id + ' clicked');
12. }
13.
14. window.addEventListener('load', function() {
15.   const btnColor = document.getElementById('btn-color');
16.   btnColor.addEventListener('click', color);
17.   btnColor.addEventListener('click', log);
18.
19.   const btnReset = document.getElementById('btn-reset');
20.   btnReset.addEventListener('click', reset);
21.   btnReset.addEventListener('click', log);
22. });
```

Notice that you don't need to see the HTML to understand how this code will work and when it will run.

1. The `color()` and `reset()` functions just change the background color of the page.
2. The `log(e)` function logs the button click. Here we just log it to the console, but in practice, we could log it to a permanent location using Ajax, which we do not cover in this course.
3. Each button gets two event listeners: one to change the color and the other to log the event. We couldn't do this with an `onclick` tag without rewriting our JavaScript to combine the logging with the color-changing functions.

To see how it works, open `EventHandlers/Demos/event-listeners-benefits.html` in Google Chrome with the console open and click the buttons several times.



6.6. Timers

Timers are started and stopped with the following four methods of the window object:

1. `setTimeout(function, waitTime)` – `waitTime` is in milliseconds.
2. `clearTimeout(timer)`
3. `setInterval(function, intervalTime)` – `intervalTime` is in milliseconds.
4. `clearInterval(interval)`

Let's take a look at how `setTimeout()` and `clearTimeout()` work first:

Demo 6.8: EventHandlers/Demos/timer.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.    // Create global timer variable
10.   let timer;
11.
12.   function changeBg(e) {
13.     const color = e.currentTarget.id;
14.     timer = setTimeout(function() {
15.       document.body.style.backgroundColor=color;
16.     }, 1000);
17.   }
18.
19.   function stopTimer() {
20.     clearTimeout(timer);
21.     alert('Timer cleared!');
22.   }
23.
24.   window.addEventListener('load', function() {
25.     btnRed = document.getElementById('red');
26.     btnWhite = document.getElementById('white');
27.     btnStop = document.getElementById('stop');
28.
29.     btnRed.addEventListener('click', changeBg);
30.     btnWhite.addEventListener('click', changeBg);
31.     btnStop.addEventListener('click', stopTimer);
32.   });
33. </script>
34. <title>Timer</title>
35. </head>
36. <body>
37. <main>
38.   <button id="red">Change Background to Red</button>
39.   <button id="white">Change Background to White</button>
40.   <button id="stop">Wait! Don't do it!</button>
41. </main>
42. </body>
43. </html>
```

Things to notice:

1. We make timer a global variable so that we can access the timer object from within multiple functions.

2. In the `changeBg()` function, we create the timer using `setTimeout()`. The first argument of `setTimeout()` is the function to execute and the second argument is the number of milliseconds to wait before executing it.
3. The `stopTimer()` function simply clears the timer using `clearTimeout()`.

The `setInterval()` and `clearInterval()` methods work the same way. The only difference is that the code gets executed repeatedly until the interval is cleared.

Demo 6.9: EventHandlers/Demos/interval.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.    // Create global interval and color variables
10.    let interval;
11.    let color = 'white';
12.
13.    function startTogglingBg() {
14.        interval = setInterval(function() {
15.            if (color === 'white') {
16.                color = 'red';
17.            } else {
18.                color = 'white';
19.            }
20.            document.body.style.backgroundColor=color;
21.        }, 500);
22.    }
23.
24.    function stopTogglingBg() {
25.        clearInterval(interval);
26.    }
27.
28.    window.addEventListener('load', function() {
29.        btnStart = document.getElementById('start');
30.        btnStop = document.getElementById('stop');
31.
32.        btnStart.addEventListener('click', startTogglingBg);
33.        btnStop.addEventListener('click', stopTogglingBg);
34.    });
35. </script>
36. <title>Timer</title>
37. </head>
38. <body>
39.   <main>
40.     <button id="start">Start</button>
41.     <button id="stop">Stop</button>
42.   </main>
43. </body>
44. </html>
```

Open EventHandlers/Demos/interval.html in your browser to see how it works. Click the **Start** button. The background should change back and forth from red to white. Click the **Stop** button to stop the changes.



Exercise 12: Typing Test

⌚ 10 to 20 minutes

In this exercise, you will create a simple typing test.

innerHTML

This exercise uses the `innerHTML` property, which you can use to read and modify the HTML content of an element.

Here is the starting code:

Exercise Code 12.1: EventHandlers/Exercises/typing-test.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.    // Global variable containing time passed
10.    let timePassed = 0;
11.
12.    function checkSentence(sentence, entry) {
13.      const msg = document.getElementById('message');
14.      if (sentence === entry) {
15.        msg.innerHTML = 'You finished in ' + timePassed + ' seconds';
16.        return true;
17.      }
18.      timePassed += .1;
19.      timePassed = parseFloat(timePassed.toFixed(1));
20.      msg.innerHTML = timePassed + ' seconds';
21.      return false;
22.    }
23.
24.    window.addEventListener('load', function() {
25.      const sentence = document.getElementById('sentence').innerHTML;
26.      const entryField = document.getElementById('entry');
27.
28.      // Write your code here.
29.    });
30.  </script>
31.  <title>Typing Test</title>
32.  </head>
33.  <body id="typing-test">
34.    <main>
35.      <div id="container">
36.        <p id="sentence">The quick brown fox jumps over the lazy dog.</p>
37.        <input id="entry" placeholder="Click to start timer.">
38.        <p id="message">0 seconds</p>
39.      </div>
40.    </main>
41.  </body>
42. </html>
```

1. Open EventHandlers/Exercises/typing-test.html in your editor.
2. Beneath the line where entryField is declared, add an event listener to entryField, so that when the user focuses on the field, an interval is created. The interval's function should run every 100 milliseconds and should do the following:

- A. Call `checkSentence()`, passing in the sentence and the value of `entryField` and assigning the result to a variable.
 - B. If `checkSentence()` returns `true`, clear the interval.
3. Test your solution in a browser.

Solution: EventHandlers/Solutions/typing-test.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.    // Global variable containing time passed
10.    let timePassed = 0;
11.
12.    function checkSentence(sentence, entry) {
13.        const msg = document.getElementById('message');
14.        if (sentence === entry) {
15.            msg.innerHTML = 'You finished in ' + timePassed + ' seconds';
16.            return true;
17.        }
18.        timePassed += .1;
19.        timePassed = parseFloat(timePassed.toFixed(1));
20.        msg.innerHTML = timePassed + ' seconds';
21.        return false;
22.    }
23.
24.    window.addEventListener('load', function() {
25.        const sentence = document.getElementById('sentence').innerHTML;
26.        const entryField = document.getElementById('entry');
27.
28.        entryField.addEventListener('focus', function() {
29.            const interval = setInterval(function() {
30.                const result = checkSentence(sentence, entryField.value);
31.                if (result) {
32.                    clearInterval(interval);
33.                }
34.            }, 100);
35.        });
36.    });
37. </script>
-----Lines 38 through 49 Omitted-----
```

Conclusion

In this lesson, you have learned:

- How to use on-event handlers to respond to user events.
- How to listen for events with the `addEventListener()` method and to understand the benefits of this approach.
- How to write anonymous functions.

- How to create timers and intervals.

Evaluation
Copy

LESSON 7

The HTML Document Object Model

EVALUATION COPY: Not to be used in class.

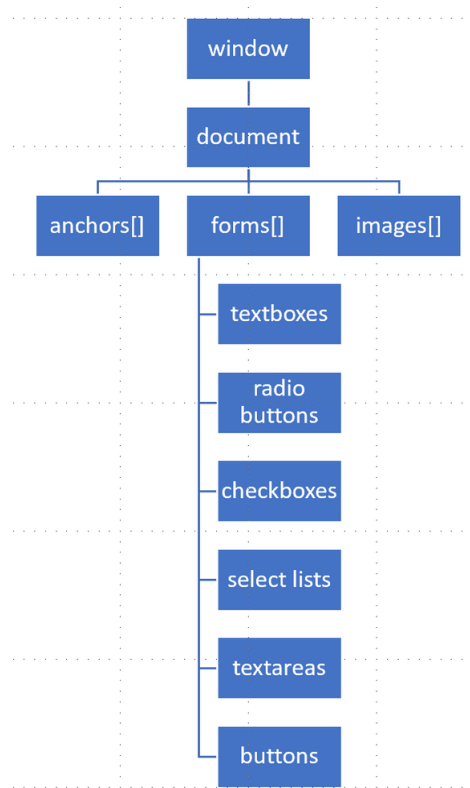
Topics Covered

- ☒ The HTML DOM.
- ☒ Accessing specific nodes.
- ☒ Accessing nodes by tag name, class name, and CSS selector.
- ☒ Accessing nodes hierarchically.
- ☒ Creating and removing nodes.
- ☒ Dynamically creating an HTML page.

Introduction

The HTML Document Object Model (DOM) is a W3C standard that defines a set of HTML objects and their methods and properties. JavaScript can be used to access, to create, and to destroy these objects, to invoke their methods, and to manipulate their properties.

A subset of the object hierarchy is shown below:



This lesson is concerned with the different ways of identifying and manipulating document nodes. While we have looked at some of these features in previous lessons, we present them here together for completeness.

EVALUATION COPY: Not to be used in class.



7.1. CSS Selectors

We will start with an introduction/review of CSS selectors as we can make use of them to access elements with JavaScript. There are several different types of selectors, including:

- Type
- Descendant
- Child

- Class
- ID
- Attribute
- Universal

Selectors identify the element(s) affected by a CSS rule.

❖ 7.1.1. Type Selectors

Type selectors specify elements by tag name and affect every instance of that element type. The rule below specifies that the text of every p element should be darkgreen and use a 10-point Verdana font:

```
p {  
  color: darkgreen;  
  font-family: Verdana;  
  font-size: 10pt;  
}
```

❖ 7.1.2. Descendant Selectors

Descendant selectors specify elements by ancestry. Each “generation” is separated by a space. For example, the following rule states that strong elements within p elements should have red text:

```
p strong {  
  color: red;  
}
```

With descendant selectors generations can be skipped. In other words, the code above does not require that the strong element is a direct child of the p element.

❖ 7.1.3. Child Selectors

Child selectors specify a direct parent-child relationship and are indicated by placing a > sign between the two tag names:

```
p > strong {  
  color: red;  
}
```

In this case, only strong elements that are direct children of p elements are affected.

❖ 7.1.4. Class Selectors

In HTML, almost all elements can take the class attribute, which assigns a class name to an element. The names given to classes are arbitrary, but should be descriptive of the purpose of the class. In CSS, class selectors begin with a dot. For example, the following rule specifies that any elements with the class “warning” should be bold and red:

```
.warning {  
  font-weight: bold;  
  color: #f00;  
}
```

Following are a couple of examples of elements of the “warning” class:

```
<h1 class="warning">WARNING</h1>  
<p class="warning">Don't go there!</p>
```

If the class selector is preceded by an element name, then that selector only applies to the specified type of element. To illustrate, the following two rules indicate that h1 elements of the class “warning” will be underlined, while p elements of the class “warning” should be bold, but will not be underlined:

```
h1.warning {  
  color: #f00;  
  text-decoration: underline;  
}  
  
p.warning {  
  color: #f00;  
  font-weight: bold;  
}
```

Because both rules indicate that the color should be red (#f00), this could be rewritten as follows:

```
.warning {
  color: #f00;
}

h1.warning {
  text-decoration: underline;
}

p.warning {
  font-weight: bold;
}
```

Note that you can assign an element any number of classes simply by separating the class names with spaces like this:

```
<div class="class1 class2 class3">...
```

❖ 7.1.5. ID Selectors

As with the `class` attribute, in HTML, almost all elements can take the `id` attribute, which is used to uniquely identify an element on the page. In CSS, id selectors begin with a pound sign (`#`) and have arbitrary names. The following rule will indent the element with the “`main-text`” id 20 pixels from the left and right:

```
#main-text {
  margin-left: 20px;
  margin-right: 20px;
}

<div id="main-text">
  This is the main text of the page...
</div>
```

❖ 7.1.6. Attribute Selectors

Attribute selectors specify elements that contain a specific attribute. They can also specify the value of that attribute.

The following selector affects all links with a `target` attribute:

```
a[target] {  
  color: red;  
}
```

The following selector would only affect links whose `target` attribute is “_blank”:

```
a[target='_blank'] {  
  color: red;  
}
```

Now, with that bit of CSS review out of the way, let’s move on to the HTML DOM.

EVALUATION COPY: Not to be used in class.



7.2. The innerHTML Property

Most HTML elements have an `innerHTML` property, which can be used to access and modify the HTML within an element.

innerHTML Illustration

Given the code:

```
<p>I <strong>love</strong> JavaScript.</p>
```

the `innerHTML` property of the `p` element would be: `I love JavaScript.`

Tip

You can use the `innerHTML` property to either **get** the element’s `innerHTML` value (as shown above) or to **set** the element’s `innerHTML` value. More on this later in the lesson.



7.3. Nodes, NodeLists, and HTMLCollections

In JavaScript, you will see the words `Node` and `NodeList` used often. For the most part, you can think of a `Node` as one of the following:

1. The document object.
2. An element.
3. A snippet of text within an element.

A `NodeList` is a list of `Node` elements and is similar to an array.

An `HTMLCollection` is very similar to a `NodeList` except that:

1. `HTMLCollections` are *live*, meaning that they take into account page changes. `NodeLists` are static.
2. `HTMLCollections` can only contain element nodes; whereas `NodeLists` can contain any type of `Node`; however, most of the time `NodeLists` will be lists of elements.

Don't Worry

If the difference between `Nodes` and `Elements` and between `NodeLists` and `HTMLCollections` seems fuzzy to you, don't worry too much about it. For the most part, you can think of `Nodes` and `Elements` as interchangeable and `NodeLists` and `HTMLCollections` as arrays containing elements. It's not until you get to pretty advanced JavaScript that you have to be able to differentiate between these different types.

- For a full technical definition of `Node`, see <https://developer.mozilla.org/en-US/docs/Web/API/Node>.
- For a full technical definition of `HTMLCollection`, see <https://developer.mozilla.org/en-US/docs/Web/API/HTMLCollection>.



7.4. Accessing Element Nodes

JavaScript provides several different ways to access elements on the page. We will look at the following methods:

- `getElementById(id)` – returns a single `Element` Node with the passed-in `id` or `null` if no such element exists.
- `getElementsByName(className)` – returns an `HTMLCollection` of `Element` Nodes with the passed-in `className`.
- `getElementsByTagName(tagName)` – returns an `HTMLCollection` of `Element` Nodes with the passed-in `tagName`.
- `querySelectorAll(selector)` – returns a `NodeList` of `Element` Nodes matching the passed-in `selector`.
- `querySelector(selector)` – returns the first `Element` Node matching the passed-in `selector`.

❖ 7.4.1. `getElementById()`

We have already seen the `document.getElementById(id)` method, which returns the first element with the given `id` (there shouldn't be more than one on the page!) or `null` if none is found. The following example illustrates how `getElementById()` works:

Demo 7.1: HTMLDOM/Demos/get-element-by-id.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.  window.addEventListener('load', function() {
10.    const elem = document.getElementById('beatles-list');
11.    alert(elem.innerHTML);
12.  });
13. </script>
14. <title>getElementById()</title>
15. </head>
16. <body>
17. <main>
18. <h1>Rockbands</h1>
19. <h2>Beatles</h2>
20.   <ol id="beatles-list">
21.     <li>Paul</li>
22.     <li>John</li>
23.     <li>George</li>
24.     <li>Ringo</li>
25.   </ol>
26.   <h2>Rolling Stones</h2>
27.   <ol id="stones-list">
28.     <li>Mick</li>
29.     <li>Keith</li>
30.     <li>Charlie</li>
31.     <li>Bill</li>
32.   </ol>
33. </main>
34. </body>
35. </html>
```

When this page loads, the following alert box will pop up:



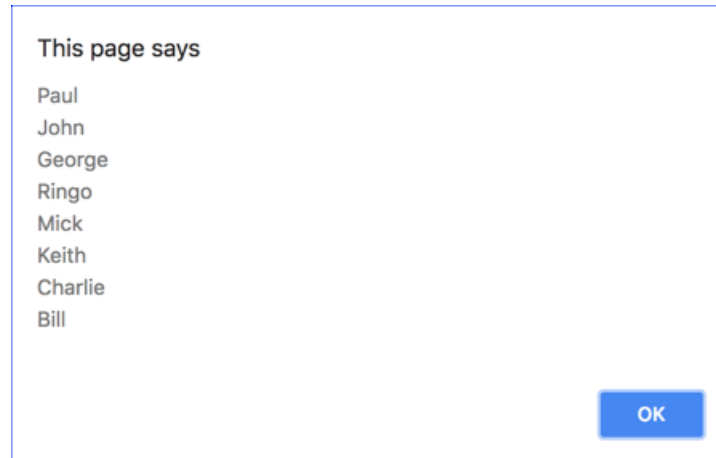
❖ 7.4.2. `getElementsByName()`

The `getElementsByName()` method of an element node retrieves all descendant (children, grandchildren, etc.) elements that have the specified tag name and stores them in a `NodeList`, which can be treated like an array of elements. The following example illustrates how `getElementsByName()` works:

Demo 7.2: `HTMLDOM/Demos/get-elements-by-tag-name.html`

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.  window.addEventListener('load', function() {
10.    const elems = document.getElementsByName('li');
11.    let msg = "";
12.    for (let elem of elems) {
13.      msg += elem.innerHTML + "\n";
14.    }
15.    alert(msg);
16.  });
17. </script>
18. <title>getElementsByName()</title>
19. </head>
20. <body>
21. <main>
22.   <h1>Rockbands</h1>
23.   <h2>Beatles</h2>
24.   <ol>
25.     <li>Paul</li>
26.     <li>John</li>
27.     <li>George</li>
28.     <li>Ringo</li>
29.   </ol>
30.   <h2>Rolling Stones</h2>
31.   <ol>
32.     <li>Mick</li>
33.     <li>Keith</li>
34.     <li>Charlie</li>
35.     <li>Bill</li>
36.   </ol>
37. </main>
38. </body>
39. </html>
```

When this page loads, the following alert box will pop up:



❖ 7.4.3. `getElementsByClassName()`

The `getElementsByClassName()` method is applicable to all elements that can have descendant elements. It is used to retrieve all the descendant (children, grandchildren, etc.) elements that have a specific class name. For example, the following code would return a `NodeList` containing all elements of the “warning” class:

```
const warnings = document.getElementsByClassName('warning');
```

❖ 7.4.4. `querySelectorAll()` and `querySelector()`

We can exploit the various CSS selectors (reviewed above) by using `querySelectorAll()` and `querySelector()`. Unlike the `getElementById()`, `getElementsByTagName()`, and `getElementsByClassName()` methods, which find elements by one specific value (id, tag name, and class name, respectively), `document.querySelector()` provides a way to find an element using many different properties of the element, and `querySelectorAll()` provides a way to find all such elements. For example, the following code would return a node list containing all a elements that are direct children of td elements:

```
const linksInTds = document.querySelectorAll('td>a');
```

The `document.querySelector()` method is the same as `document.querySelectorAll()` but rather than returning a list, it returns only the first element found. The following two lines of code would both return the first link element found in an td element:

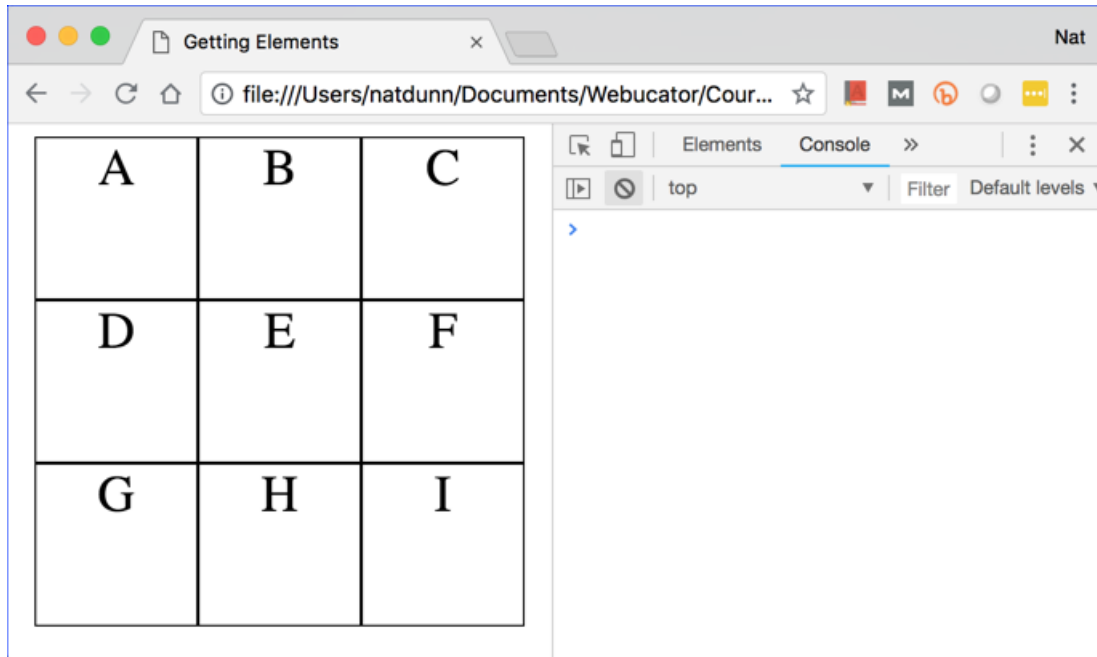
```
const firstLinkInTd = document.querySelectorAll('td>a')[0];  
const firstLinkInTd = document.querySelector('td>a');
```

Now you have a chance to play with these methods using Chrome DevTools Console. You will start with the following file:

Demo 7.3: HTMLODOM/Demos/getting-elements.html

```
1.  <!DOCTYPE html>  
2.  <html lang="en">  
3.  <head>  
4.  <meta charset="UTF-8">  
5.  <meta name="viewport" content="width=device-width,initial-scale=1">  
6.  <link rel="stylesheet" href="../normalize.css">  
7.  <link rel="stylesheet" href="../styles.css">  
8.  <title>Getting Elements</title>  
9.  </head>  
10. <body>  
11. <main>  
12.   <div id="board">  
13.     <div class="row">  
14.       <div class="col">A</div>  
15.       <div class="col">B</div>  
16.       <div class="col">C</div>  
17.     </div>  
18.     <div class="row">  
19.       <div class="col">D</div>  
20.       <div class="col">E</div>  
21.       <div class="col">F</div>  
22.     </div>  
23.     <div class="row">  
24.       <div class="col">G</div>  
25.       <div class="col">H</div>  
26.       <div class="col">I</div>  
27.     </div>  
28.   </div>  
29. </main>  
30. </body>  
31. </html>
```

1. Open HTMLODOM/Demos/getting-elements.html in Google Chrome and open the console:



2. Using the console, write code to do the following:
 - A. Turn the background of the whole board to pink:

A	B	C
D	E	F
G	H	I

- B. Turn the second row to lime:

A	B	C
D	E	F
G	H	I

C. Turn the middle cell to white:

A	B	C
D	E	F
G	H	I

D. Refresh the page and clear the console to start with the original board. Turn the first column pink. There are several ways to do this. Can you figure out more than one?

A	B	C
D	E	F
G	H	I

- E. Refresh the page and clear the console to start with the original board. Change the content of the squares from A-I to 1-9:

1	2	3
4	5	6
7	8	9

3. Here are possible solutions:

- A. Turn the background of the whole board to pink:

A	B	C
D	E	F
G	H	I

```

> const board = document.getElementById('board');
  board.style.backgroundColor = 'pink';
< "pink"
> |

```

B. Turn the second row to lime:

A	B	C
D	E	F
G	H	I

```
> const board = document.getElementById('board');
board.style.backgroundColor = 'pink';
< "pink"
> const rows = document.getElementsByClassName('row');
rows;
< ▶ HTMLCollection(3) [div.row, div.row, div.row]
> rows[1].style.backgroundColor = 'lime';
< "lime"
> |
```

C. Turn the middle cell to white:

A	B	C
D	E	F
G	H	I

```
> const r2cols = rows[1].getElementsByClassName('col');
r2cols;
< ▶ HTMLCollection(3) [div.col, div.col, div.col]
> r2cols[1].style.backgroundColor = 'white';
< "white"
> |
```

D. Refresh the page and clear the console to start with the original board. Turn the first column pink:

A	B	C
D	E	F
G	H	I

Three possible solutions:

```
> const cols = document.querySelectorAll('.row>.col:first-child');
cols;
< ▶ NodeList(3) [div.col, div.col, div.col]
> for (var col of cols) {
  col.style.backgroundColor = 'pink';
}
< "pink"
```

```
> const rows = document.getElementsByClassName('row');
  for (let row of rows) {
    row.querySelector('.col').style.backgroundColor = 'pink';
  }
< "pink"
```

querySelector () gets the first element that matches, so it only gets the first column in the row.


```
> const rows = document.getElementsByClassName('row');
  for (let row of rows) {
    const col = row.getElementsByClassName('col')[0];
    col.style.backgroundColor = 'pink';
  }
< "pink"
```

- E. Refresh the page and clear the console to start with the original board. Change the content of the squares from A-I to 1-9:

1	2	3
4	5	6
7	8	9

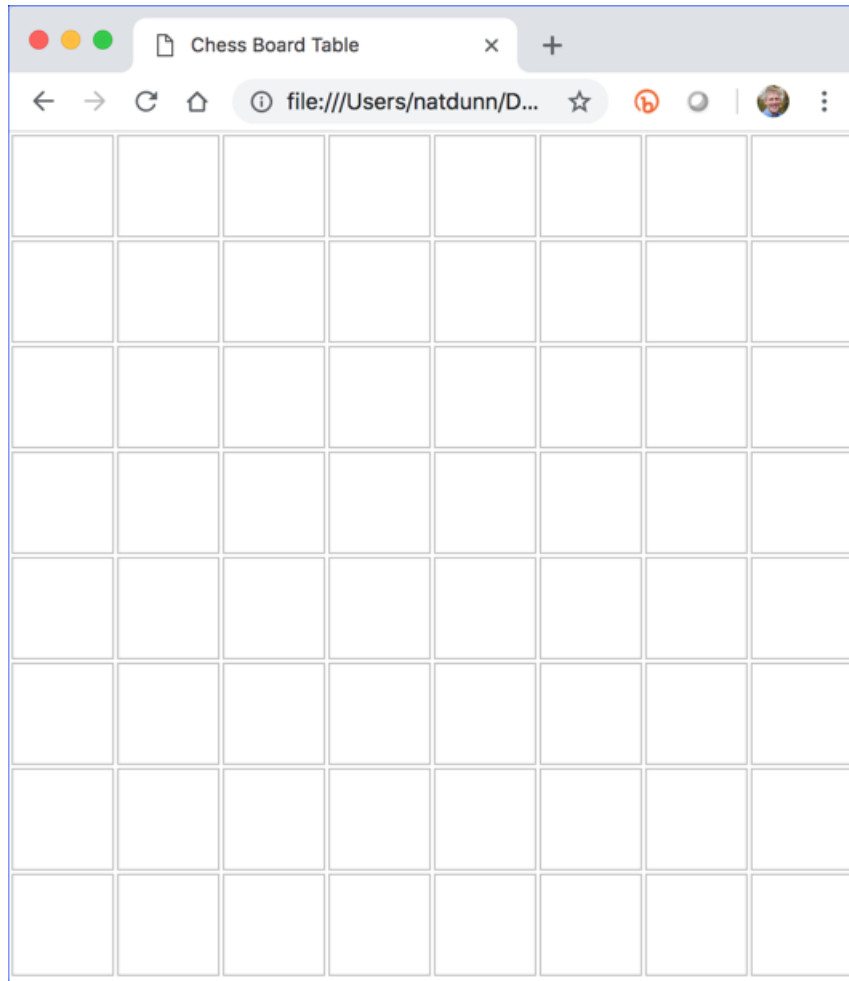
```
> const squares = document.getElementsByClassName('col');
  for (let i=0; i < squares.length; i++) {
    squares[i].innerHTML = i+1;
  }
< 9
> |
```

Exercise 13: Accessing Elements

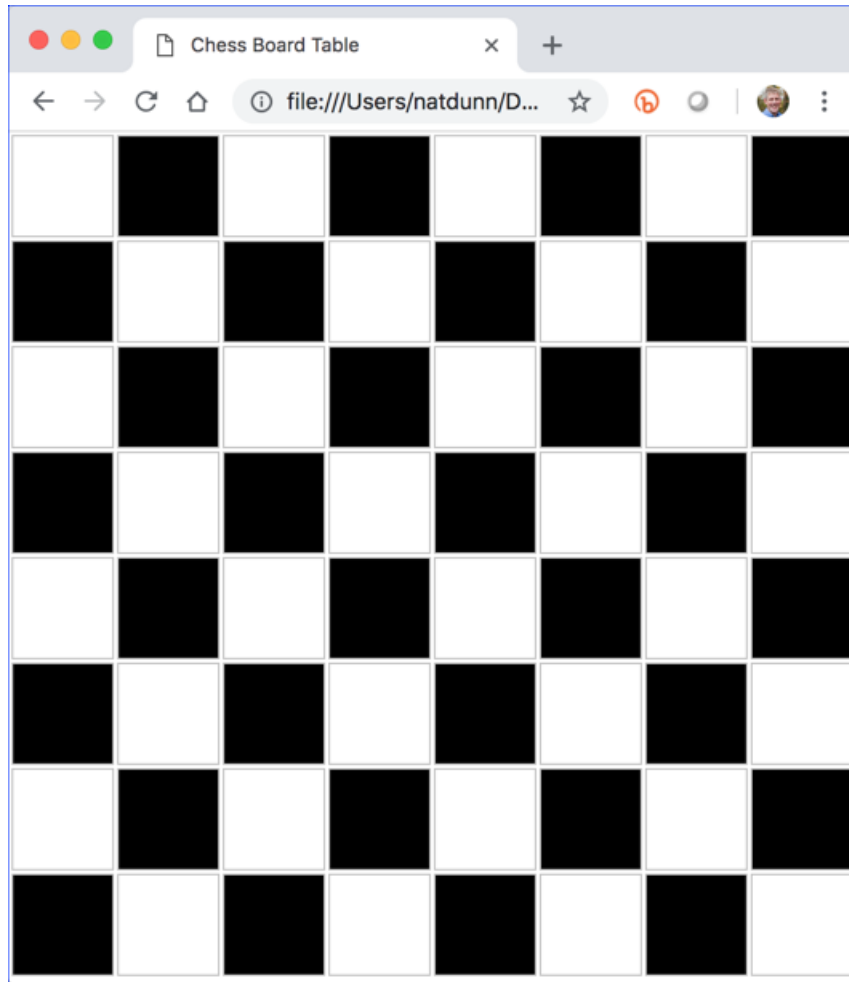
 10 to 15 minutes

In this exercise, you will practice accessing elements in JavaScript.

1. Open `HTMLDOM/Exercises/chessboard-table.html` in your browser. It contains an 8 x 8 table:



2. Open `HTMLDOM/Exercises/chessboard-table.html` for editing.
3. Add JavaScript so that when the page loads, it checks the table to look like this:



Solution: HTMLDOM/Solutions/chessboard-table.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.    window.addEventListener('load', function(e) {
10.      const oddrows = document.querySelectorAll('tr.odd');
11.      const evenrows = document.querySelectorAll('tr.even');
12.      for (row of oddrows) {
13.        const evencols = row.querySelectorAll('.even');
14.        for (col of evencols) {
15.          col.style.backgroundColor = 'black';
16.        }
17.      }
18.      for (row of evenrows) {
19.        const oddcols = row.querySelectorAll('.odd');
20.        for (col of oddcols) {
21.          col.style.backgroundColor = 'black';
22.        }
23.      }
24.    });
25.  </script>
-----Lines 26 through 112 Omitted-----
```

Evaluation
Copy

Code Explanation

The solution shown here is just one of many ways to do this.

EVALUATION COPY: Not to be used in class.



7.5. Dot Notation and Square Bracket Notation

In the first lesson of this course, we took a look at two ways to access elements in JavaScript: *dot notation* and *square bracket notation*. Let's review these concepts again.

Dot notation lets us refer to hierarchical DOM elements starting with the top-most element (window) then a set of dot-separated names, referencing elements by their name. For instance, to get an input

element with the name `fname` inside a form with the name `loginform`, we might use the following (as long as there are no hyphens in the names):

```
window.document.loginform.fname
```

❖ 7.5.1. Collections of Elements

A document can have multiple form elements as children. We call this the document's forms collection. We can reference the specific form by its order on the page. Like arrays, collections in JavaScript start with index `0`, so the first form on the page would be `forms[0]`.

```
window.document.forms[0].fname
```

❖ 7.5.2. window is Implicit

As `window` is the implicit top-level object, we don't have to refer to it explicitly. The preceding code samples could be written as:

```
document.loginform.fname  
document.forms[0].fname
```

Similarly, we can reference objects with *square bracket notation*, where the key is the name of the element:

```
document['loginform']['fname']
```

This is equivalent to the dot-notation references we showed earlier and can be used interchangeably.

Let's play with this a little in the Chrome DevTools Console using the following file:

Demo 7.4: HTMLODOM/Demos/forms.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <title>Forms</title>
9. </head>
10. <body>
11. <main>
12.   <form name="form-a">
13.     <input name="fname">
14.   </form>
15.   <form name="form-b">
16.     <input name="fname">
17.   </form>
18.   <form name="form-c">
19.     <input name="fname">
20.   </form>
21.   <form name="form-d">
22.     <input name="fname">
23.   </form>
24. </main>
25. </body>
26. </html>
```



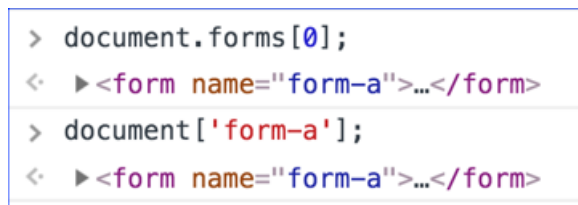
Notice the file has four form elements named “form-a”, “form-b”, “form-c”, and “form-d”. Each of those forms has an input element named “fname”.

1. Open HTMLODOM/Demos/forms.html in Google Chrome.
2. In the console, type `document.forms`; and press **Enter**. Then click the triangle (circled below) to expand the collection:

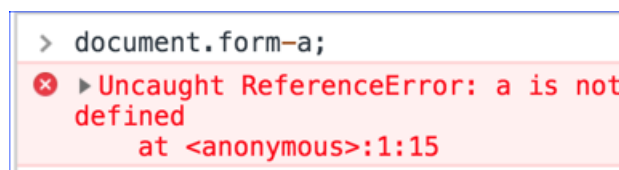


Notice that you see both ways of referencing the forms, by index (0, 1, 2, and 3) and by name (form-a, form-b, form-c, and form-d).

3. Now type `document["forms"]`; and press **Enter** and notice that you get the same result, demonstrating that you can use dot and square-bracket notation interchangeably.
4. Now run each of the following and notice that they both deliver the first form:
 - A. `document.forms[0]`;
 - B. `document["form-a"]`;



5. However, if you try to access the same form using dot notation you will get an error:



This is because of the hyphen in the name. It reads this as `document.form` minus `a` and errors because `a` is undefined. So, when using hyphens in names, you should use square-bracket notation or use another technique for getting the objects.

6. You can use either dot notation or square-bracket notation to access the “fname” input elements, because the name doesn’t contain a hyphen:

```
> const form = document['form-a'];  
< undefined  
> form.fname;  
< <input name="fname">  
> form['fname'];  
< <input name="fname">
```

7. Enter your name in the first form’s textbox and type `document.forms['form-a']['fname'].value` (or one of the other variations) at the console:

<input type="text" value="Nat"/> <input type="text"/> <input type="text"/>	<pre>> document['form-a'].fname.value; < "Nat" ></pre>
--	---

8. Now use JavaScript to set the value of fname in form-b:

<input type="text"/> <input type="text" value="Nathaniel"/>	<pre>> document['form-b'].fname.value = 'Nathaniel'; < "Nathaniel"</pre>
--	--

EVALUATION COPY: Not to be used in class.



7.6. Accessing Elements Hierarchically

JavaScript provides a variety of methods and properties for accessing elements based on their hierarchical relationship. The most common are shown in the table below:

Properties for Accessing Element Nodes

Property	Description
children	A collection of the element's child elements.
firstElementChild	A reference to an element's first child element. The equivalent of <code>children[0]</code> .
lastElementChild	A reference to an element's last child element. The equivalent of <code>children[children.length-1]</code> .
previousElementSibling	A reference to the previous element at the same level in the document tree.
nextElementSibling	A reference to the next element at the same level in the document tree.
parentNode	A reference to an element's parent node. ⁹

The `children` property returns a collection of element nodes. The other properties return a single element node.

These properties provide a flexible way to get elements on the page, relative to their parents, siblings, or children. We can do anything with the returned elements that we did previously when retrieving the elements with `getElementById()`, `querySelector()` and the other methods – set the background color, change the font style, etc.

Let's take a look at how we might use these properties:

9. A node is an object in the document tree. Elements, attributes, and text snippets are all examples of nodes. While there are some obscure exceptions, you can generally expect the `parentNode` of an element to be an element.

Demo 7.5: HTMLDOM/Demos/elem-hierarchy.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.    function modify() {
10.      const list = document.getElementById('list');
11.      const liFirst = list.firstChild;
12.      liFirst.style.backgroundColor = 'pink';
13.      const liLast = list.lastElementChild;
14.      liLast.style.backgroundColor = 'aqua';
15.      const siblingPrev = liLast.previousElementSibling;
16.      siblingPrev.style.backgroundColor = 'lime';
17.
18.      for (item of list.children) {
19.        item.innerHTML += ' - check';
20.      }
21.    }
22.
23.    window.addEventListener('load', function() {
24.      const goBtn = document.getElementById('btn-go');
25.      goBtn.addEventListener('click', modify);
26.    });
27.  </script>
28.  <title>Element Hierarchy</title>
29.  </head>
30.  <body>
31.    <main>
32.      <button id="btn-go">Go</button>
33.      <ul id="list">
34.        <li>Item 1</li>
35.        <li>Item 2</li>
36.        <li>Item 3</li>
37.        <li>Item 4</li>
38.        <li>Item 5</li>
39.      </ul>
40.    </main>
41.  </body>
42. </html>
```



Our simple page displays a button and five unordered list items, with text “Item 1”, “Item 2”, etc.

Clicking the button calls the function `modify()`, which does the following:

- Gets the first child of the list using `firstElementChild`, and sets its background to pink.
- Gets the last child of the list using `lastElementChild`, and sets its background to aqua.

- Gets the next-to-last child of the list using `previousElementSibling` (relative to the already-gotten `liLast`), and sets its background to lime.
- Loops through all the list items (children of the list) adding “ - check” to the `innerHTML`.

We'll ask you to try out these properties in the next exercise.



Exercise 14: Working with Hierarchical Elements

Evaluation
Copy

⌚ 10 to 15 minutes

You will start with the code shown below:

Exercise Code 14.1: HTMLDOM/Exercises/elem-hierarchy.html

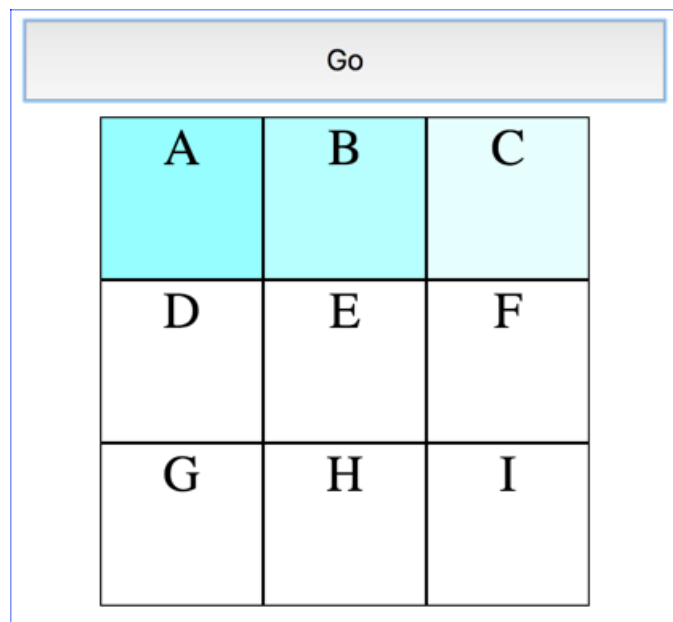
```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.      function create() {
10.         const board = document.getElementById('board');
11.         const topRow = board.firstChild;
12.         const trLeftCol = topRow.firstChild;
13.         trLeftCol.style.backgroundColor='rgba(0, 255, 255, .5)';
14.         const trCenterCol = trLeftCol.nextElementSibling;
15.         trCenterCol.style.backgroundColor='rgba(102, 255, 255, .5)';
16.         const trRightCol = topRow.lastElementChild;
17.         trRightCol.style.backgroundColor='rgba(204, 255, 255, .5)';
18.     }
19.
20.     window.addEventListener('load', function() {
21.         const goBtn = document.getElementById('btn-go');
22.         goBtn.addEventListener('click', create);
23.     });
-----Lines 24 through 28 Omitted-----
29.     <button id="btn-go">Go</button>
30.     <div id="board">
31.         <div class="row">
32.             <div class="col">A</div>
33.             <div class="col">B</div>
34.             <div class="col">C</div>
35.         </div>
36.         <div class="row">
37.             <div class="col">D</div>
38.             <div class="col">E</div>
39.             <div class="col">F</div>
40.         </div>
41.         <div class="row">
42.             <div class="col">G</div>
43.             <div class="col">H</div>
44.             <div class="col">I</div>
45.         </div>
46.     </div>
-----Lines 47 through 49 Omitted-----
```

rgba(R, G, B, A) Functional Notation

We are using `rgba(R, G, B, A)` functional notation in this exercise. **R**, **G**, and **B** indicate the amount of **Red**, **Green**, and **Blue** in the color. **A** indicates the opacity level: 0 (fully transparent) to 1 (full opacity).

In this exercise, you will practice working with JavaScript's hierarchical elements.

1. Open `HTMLDOM/Exercises/elem-hierarchy.html` in the browser, click the **Go** button, and notice how the background colors of the first row's cells change:



2. Note that a click handler has been added to the button so that the function `create()` is called when the user clicks the button.
3. Finish the `create()` function so that each cell has a different color. You can use your own colors or the ones listed below:
 - A. `rgba(0, 255, 255, .5)`
 - B. `rgba(102, 255, 255, .5)`
 - C. `rgba(204, 255, 255, .5)`
 - D. `rgba(255, 0, 255, .5)`
 - E. `rgba(255, 102, 255, .5)`

F. `rgba(255, 204, 255, .5)`

G. `rgba(255, 255, 0, .5)`

H. `rgba(255, 255, 102, .5)`

I. `rgba(255, 255, 204, .5)`

Evaluation
Copy

Solution: HTMLDOM/Solutions/elem-hierarchy.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.    function create() {
10.      const board = document.getElementById('board');
11.      const topRow = board.firstElementChild;
12.      const trLeftCol = topRow.firstElementChild;
13.      trLeftCol.style.backgroundColor='rgba(0, 255, 255, .5)';
14.      const trCenterCol = trLeftCol.nextElementSibling;
15.      trCenterCol.style.backgroundColor='rgba(102, 255, 255, .5)';
16.      const trRightCol = topRow.lastElementChild;
17.      trRightCol.style.backgroundColor='rgba(204, 255, 255, .5)';
18.
19.      const middleRow = topRow.nextElementSibling;
20.      const mrLeftCol = middleRow.firstElementChild;
21.      mrLeftCol.style.backgroundColor='rgba(255, 0, 255, .5)';
22.      const mrCenterCol = mrLeftCol.nextElementSibling;
23.      mrCenterCol.style.backgroundColor='rgba(255, 102, 255, .5)';
24.      const mrRightCol = middleRow.lastElementChild;
25.      mrRightCol.style.backgroundColor='rgba(255, 204, 255, .5)';
26.
27.      const bottomRow = board.lastElementChild;
28.      const brLeftCol = bottomRow.firstElementChild;
29.      brLeftCol.style.backgroundColor='rgba(255, 255, 0, .5)';
30.      const brCenterCol = brLeftCol.nextElementSibling;
31.      brCenterCol.style.backgroundColor='rgba(255, 255, 102, .5)';
32.      const brRightCol = bottomRow.lastElementChild;
33.      brRightCol.style.backgroundColor='rgba(255, 255, 204, .5)';
34.    }
35.
36.    window.addEventListener('load', function() {
37.      const goBtn = document.getElementById('btn-go');
38.      goBtn.addEventListener('click', create);
39.    });
40.  </script>
-----Lines 41 through 65 Omitted-----
```

EVALUATION COPY: Not to be used in class.



7.7. Accessing Attributes

Essentially, all standard attributes of HTML elements can be accessed as properties of the element. For example, given the following link:

```
<a href="https://www.google.com"
  id="google" target="_blank">Google</a>
```

We can access the value of the `target` attribute like this:

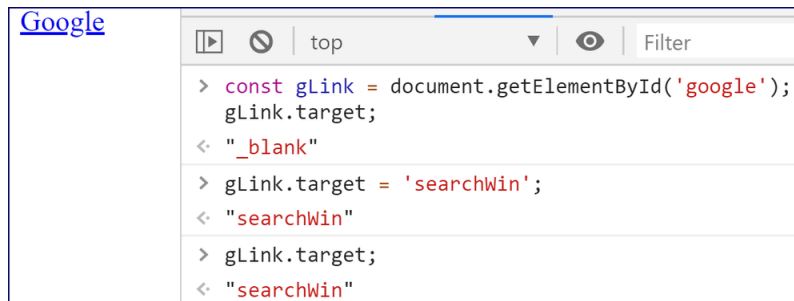
```
const gLink = document.getElementById('google');
console.log(gLink.target);
```

Likewise, we can set the value of the `target` attribute using the `target` property:

```
gLink.target = "searchWin";
```

To test this:

1. Open `HTMLDOM/Demos/attributes.html` in Google Chrome.
2. Click the Google link and notice that it opens in a new window or tab.
3. Run the code above at the console:



Notice that before you set `gLink.target`, its value is “_blank” and after you set it, its value is “searchWin”.

You can also access and modify attribute values using the following methods and properties:

Methods and Properties for Working with Attributes

Method/Property	Description
<code>hasAttribute(AttributeName)</code>	Returns a Boolean (true/false) value indicating whether or not the element to which the method is applied includes the given attribute.
<code>getAttribute(AttributeName)</code>	Returns the attribute value or null if the attribute doesn't exist.
<code>setAttribute(AttributeName, attributeValue)</code>	Adds an attribute with a value or, if the attribute already exists, changes the value of the attribute.
<code>removeAttribute(AttributeName)</code>	Removes the attribute (if it exists) from an element.
<code>attributes</code>	Property referencing the collection of an element's attributes.

EVALUATION COPY: Not to be used in class.

7.8. Creating New Nodes

The document node has separate methods for creating element nodes and creating text nodes: `createElement()` and `createTextNode()`. These methods each create a node in memory that then has to be placed somewhere in the object hierarchy. A new node can be inserted as a child to an existing node with that node's `appendChild()` and `insertBefore()` methods.

Moving Nodes

You can also use the `appendChild()` and `insertBefore()` methods to move an existing node – the node will be removed from its current location and placed at the new location (since the same node cannot exist twice in the same document).

These methods and some others are described in the table below:

Methods for Inserting and Removing Nodes

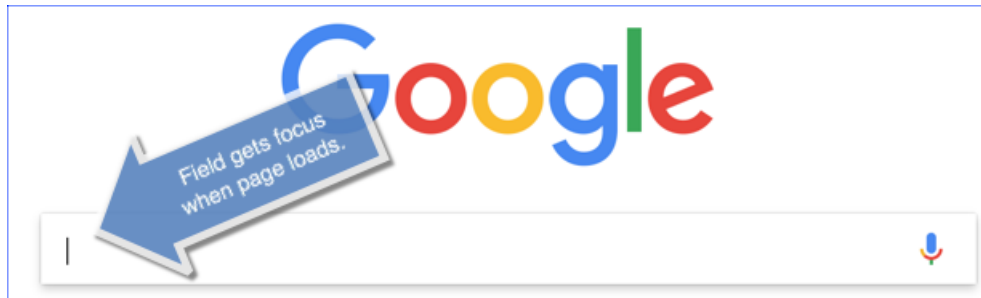
Method	Description
<code>appendChild()</code>	Takes a single parameter: the node to insert, and inserts that node after the last child node.
<code>insertBefore()</code>	Takes two parameters: the node to insert and the child node that it should precede. The new child node is inserted before the referenced child node.
<code>replaceChild()</code>	Takes two parameters: the new node and the node to be replaced. It replaces the old node with the new node and returns the old node.
<code>remove()</code>	Removes an element from the Document Object Model. It does not destroy the element, it just removes it from its parent.

EVALUATION COPY: Not to be used in class.



7.9. Focusing on a Field

When you visit `https://www.google.com`, you will notice that the search input field gets immediate focus, so that you can start typing your search right away:



This is accomplished using the `focus()` method of the `input` element, like this:

```
const searchInput = document.getElementById('search');
searchInput.focus();
```

It is often tied to the window's load event, like this:

Demo 7.6: HTMLDOM/Demos/focus.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.     window.addEventListener("load", function() {
10.         const searchInput = document.getElementById('search');
11.         searchInput.focus();
12.     });
13. </script>
14. <title>Focus</title>
15. </head>
16. <body>
17. <main>
18.     <form>
19.         <input id="search" name="search">
20.         <button>Search</button>
21.     </form>
22. </main>
23. </body>
24. </html>
```



Open HTMLDOM/Demos/focus.html in your browser to see how it works.

EVALUATION COPY: Not to be used in class.



7.10. Shopping List Application

Using what we have learned in this lesson, we will build the one-page shopping list application shown below:

Shopping List App

New Item:

Active List

Lettuce	<input data-bbox="634 478 667 520" type="button" value="-"/>
Bread	<input data-bbox="634 531 667 573" type="button" value="-"/>

Common Items

Milk	<input data-bbox="971 478 1003 520" type="button" value="+"/>
Eggs	<input data-bbox="971 531 1003 573" type="button" value="+"/>
Bread	<input data-bbox="971 583 1003 625" type="button" value="+"/>
Chicken	<input data-bbox="971 636 1003 678" type="button" value="+"/>
Tomatoes	<input data-bbox="971 688 1003 730" type="button" value="+"/>

Log

1. 3:56:32 PM: Page Loaded
2. 3:56:59 PM: Milk added.
3. 3:57:05 PM: Lettuce added.
4. 3:57:07 PM: Bread added.
5. 3:57:14 PM: Milk removed.

Open `HTMLDOM/Solutions/shopping-list.html` in your browser to see how the finished application works:

1. Notice that “Page Loaded” is logged and the **New Item** field gets focus.
2. Add Milk by clicking the **+** button next to Milk under **Common Items**.
3. Add Lettuce by typing “Lettuce” in the **New Item** field and pressing the **+** button. Notice the **New Item** field gets focus, making it easy to enter another value.
4. Add Bread by typing “Bread” in the **New Item** field and pressing the **Enter** key.
5. Try adding Bread again both by clicking the **+** button and using the **New Item** field. Both attempts should fail silently.
6. Try pressing the **+** button next to an empty **New Item** field. It should fail silently.
7. Try entering just spaces in the **New Item** field and pressing the **+** button. It should fail silently.
8. Remove Milk by clicking the **-** button next to Milk under **Active List**.

The HTML (`HTMLDOM/Exercises/shopping-list.html`) and CSS (`HTMLDOM/Exercises/shopping-list.css`) have already been completed. You will build the JavaScript (`HTMLDOM/Exercises/shopping-list.js`) piece by piece.



Exercise 15: Logging

🕒 15 to 25 minutes

In this exercise, you will complete the `log(msg)` function.

1. Open `HTMLODOM/Exercises/shopping-list.html` in your editor. Examine the section of the code shown below. The ordered list will contain the log. You will need to access that ordered list and add list items to it with JavaScript.

```
<section id="log">
  <h2>Log</h2>
  <ol></ol>
</section>
```

2. Open `HTMLODOM/Exercises/shopping-list.js` in your editor.
3. In the `log(msg)` function, write code to:
 - A. Access the ordered list shown above and save it in a constant.
 - B. Create a new list item element and save it in a constant.
 - C. Get the current date and save it in a constant.
 - D. Set the `innerHTML` of the new list item to the current local time using the `toLocaleTimeString()` method, followed by a colon, followed by the `msg` passed to `log(msg)`. For example, “5:53:12 PM: Page Loaded”.
 - E. Append the new list item to the ordered list.
4. Test your code in the browser. When the page loads, it should log “Page Loaded”. If it isn’t working, use the console to help you debug.

Solution: HTMLDOM/Solutions/shopping-list.1.js

```
1.  /* Log Messages */
2.  function log(msg) {
3.      // Access the ordered list and save it in a variable
4.      const log = document.querySelector('section#log>ol');
5.      // Create a new list item element and save it in a variable
6.      const newItem = document.createElement('li');
7.      // Get the current date and save it in a variable
8.      const now = new Date();
9.      // Set the innerHTML of the new list item
10.     newItem.innerHTML = now.toLocaleTimeString() +
11.         ': <em>' + msg + '</em>';
12.     // Append the new list item to the ordered list
13.     log.appendChild(newItem);
14. }

-----Lines 15 through 30 Omitted-----
```



Exercise 16: Adding EventListeners

🕒 25 to 40 minutes

In this exercise, you will add EventListeners in the `init()` function so that you can log when a new item is added. You will not yet write the code to actually add the items. You will do that in the next exercise.

1. Open `HTMLDOM/Exercises/shopping-list.html` in your editor. You will have to listen for the following events:
 - A. Clicks on any button element with the class `"btn-add"`.
 - B. Clicks on the button element with the id `"add-new-item"`.
 - C. Keyup events on the input element with the id `"new-item"`.
2. Open `HTMLDOM/Exercises/shopping-list.js` in your editor if it isn't already open.
3. Beneath the `log('Page Loaded');` line, declare the following three constants:
 - A. `btnListAdd` – A collection of button elements with the class `"btn-add"`.
 - B. `btnAddNewItem` – The button element with the id `"add-new-item"`.
 - C. `newItem` – The input element with the id `"new-item"`.
4. Add a line of code to place focus on the `newItem` input, so the user can just start typing in a new item.
5. Each button in the `btnListAdd` collection is coded as follows:

```
<button class="btn-add" name="Milk">+</button>
```

When the user clicks one of these buttons, your code should pass the name of that button as the argument for `product` to the `addToList(product)` function. To do this, you will need to loop through these buttons, adding click EventListeners to each. You will need to know which of the buttons is clicked (`e.currentTarget`) so that you get the value of its `name` attribute.

6. The `add-new-item` button is coded as follows:

```
<button id="add-new-item">+</button>
```

And the associated text field is:

```
<input id="new-item">
```

When the user clicks the “add-new-item” button, your code should:

- A. Pass the value of the text field as the argument for `product` to the `addToList(product)` function.
 - B. Clear the text field.
 - C. Place focus on the text field.
7. Finally, you need to add an `EventListener` for the `keyup` event on the “new-item” text field. The callback function should check if the key pressed was the **Enter** key. If it was, it should:
 - A. Pass the value of the text field as the argument for `product` to the `addToList(product)` function.
 - B. Clear the text field.
 - C. Place focus on the text field.
8. Test your code in the browser. At this point, the shopping lists won’t change, but logging should work when you add new items. If it isn’t working, use the console to help you debug.

Solution: HTMLDOM/Solutions/shopping-list.2.js

```
-----Lines 1 through 20 Omitted-----
21. function init() {
22.     log('Page Loaded');
23.     const btnListAdd = document.getElementsByClassName('btn-add');
24.     const btnAddNewItem = document.getElementById('add-new-item');
25.     const newItem = document.getElementById('new-item');
26.     newItem.focus();
27.
28.     /* Add event listeners to all common list Add buttons */
29.     for (btn of btnListAdd) {
30.         btn.addEventListener('click', function(e) {
31.             const button = e.currentTarget;
32.             const product = button.name;
33.             addToList(product);
34.             newItem.focus();
35.         });
36.     }
37.
38.     /* Add event listener to New Item Add button */
39.     btnAddNewItem.addEventListener('click', function() {
40.         addToList(newItem.value);
41.         newItem.value='';
42.         newItem.focus();
43.     });
44.
45.     /*
46.      Add event listener capturing Enter press while
47.      focus is on New Item field
48.     */
49.     newItem.addEventListener('keyup', function(e) {
50.         if (e.key === 'Enter') {
51.             addToList(newItem.value);
52.             newItem.value='';
53.             newItem.focus();
54.         }
55.     });
56. }
57.
58. window.addEventListener("load", init);
```

Exercise 17: Adding Items to the List

🕒 15 to 25 minutes

In this exercise, you will write the `addToList()` function.

1. Open `HTMLDOM/Exercises/shopping-list.js` in your editor if it isn't already open.
2. Currently, the `addToList()` function should look like this:

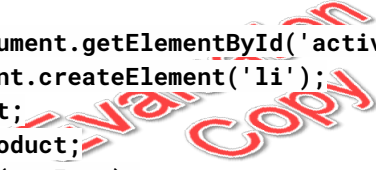
```
function addToList(product) {  
  log(product + ' added.')
```

```
}  
  
You will write your code above the log(product + ' added.') line that does the following:
```

- A. Removes leading and trailing whitespace from the passed-in product, so that if the user enters “ Milk ”, we store it as “Milk”.
 - B. Access the “active-items-list” unordered list and save it in a constant.
 - C. Create a new list item element and save it in a constant.
 - D. Set the title of the new list item element to the product name.
 - E. Set the `innerHTML` of the new list item element to the product name.
 - F. Append the new list item to the “active-items-list” unordered list.
3. Test your code in the browser. You should now be able to add items to list. If it isn't working, use the console to help you debug.

Solution: HTMLDOM/Solutions/shopping-list.3.js

```
-----Lines 1 through 16 Omitted-----  
17. function addToList(product) {  
18.     product = product.trim();  
19.  
20.     const activeList = document.getElementById('active-items-list');  
21.     const newItem = document.createElement('li');  
22.     newItem.title = product;  
23.     newItem.innerHTML = product;  
24.     activeList.appendChild(newItem);  
25.     log(product + ' added.');
```



```
26. }  
-----Lines 27 through 65 Omitted-----
```


Exercise 18: Dynamically Adding Remove Buttons to the List Items

🕒 15 to 25 minutes

In this exercise, you will continue to work in the `addToList()` function. You will add remove buttons to the list items you created in the last exercise.

1. Open `HTMLDOM/Exercises/shopping-list.js` in your editor if it isn't already open.
2. Currently, the `addToList()` function should look something like this:

```
function addToList(product) {  
  product = product.trim();  
  
  const activeList = document.getElementById('active-items-list');  
  const newItem = document.createElement('li');  
  newItem.title = product;  
  newItem.innerHTML = product;  
  activeList.appendChild(newItem);  
  log(product + ' added.')
```



```
}
```

You will write your code below the `log(product + ' added.')` line that does the following:


- A. Create a button element with a minus sign that calls `removeFromList()` when clicked and append it to the new list item.
- B. Add a space between the product name and the new button.
- C. Check if the list item being added is in the common list items. If it is, disable the “add” button for that list item by setting its `disabled` property to `true`. **Hint:** Look at the name attributes of the buttons in the “common-items-list” list. Can you use `querySelector()` to find a button with the same name as the new list item you’re adding?

Note that these directions are intentionally less specific than in the previous exercises.

3. Test your code in the browser. The list items in the ‘active-items-list’ ordered list should now have remove buttons. They won’t actually remove the items, but they should log “Item removed” when clicked. Also, any item in the “common-items-list” list that is also in the “active-items-list” should have its “add” button disabled (red and unclickable). If your code isn’t working, use the console to help you debug.

Solution: HTMLDOM/Solutions/shopping-list.4.js

```
-----Lines 1 through 16 Omitted-----
17. function addToList(product) {
18.     product = product.trim();
19.
20.     const activeList = document.getElementById('active-items-list');
21.     const newItem = document.createElement('li');
22.     newItem.title = product;
23.     newItem.innerHTML = product + ' '; // space before button
24.     activeList.appendChild(newItem);
25.     log(product + ' added.');
```



```
26.
27.     const btnRemove = document.createElement('button');
28.     btnRemove.innerHTML = '-';
29.     btnRemove.addEventListener('click', removeFromList);
30.     newItem.appendChild(btnRemove);
31.
32.     // Check if list item being added is in common list items
33.     // If it is, we need to disable its button there.
34.     const selector = '#common-items-list>li>button[name="' + product + '"]';
35.     const btnMatch = document.querySelector(selector);
36.     if (btnMatch) {
37.         btnMatch.disabled = true;
38.     }
39. }
```

```
-----Lines 40 through 78 Omitted-----
```



Exercise 19: Removing List Items

⌚ 15 to 25 minutes

In this exercise, you will write the `removeFromList()` function to remove elements from the 'active-items-list' ordered list.

1. Open `HTMLDOM/Exercises/shopping-list.js` in your editor if it isn't already open.
2. Currently, the `removeFromList()` function should look like this:

```
function removeFromList(e) {  
  log('Item Removed');  
}
```

- A. Using the passed-in event (`e`), access the list item that contains the button that was clicked to call this function and assign that list item to a constant.
 - B. Remove that item from the list.
 - C. Change `log('Item Removed')` to log the name of the product removed.
 - D. Check if the list item being removed is in the common list items. If it is, re-enable the "add" button for that list item by setting its `disabled` property to `false`.
3. Test your code in the browser. When a remove button is clicked, the associated list item should now get removed and the log should tell you which item was removed. In addition, if there is an associated list item in the "common-items-list" list, its "add" button should be re-enabled. If your code isn't working, use the console to help you debug.

Solution: HTMLDOM/Solutions/shopping-list.5.js

```
-----Lines 1 through 10 Omitted-----  
11.  /* Remove item from list */  
12.  function removeFromList(e) {  
13.      const item = e.currentTarget.parentNode;  
14.      item.remove();  
15.      log(item.title + ' removed.');
```

16.

```
17.      // Check if list item being removed is in common list items  
18.      // If it is, we need to enable its button there.  
19.      const selector = '#common-items-list>li>button[name="' +  
20.          item.title + '"]';  
21.      const btnMatch = document.querySelector(selector);  
22.      if (btnMatch) {  
23.          btnMatch.disabled = false;  
24.      }  
25.  }
```

-----Lines 26 through 89 Omitted-----



Exercise 20: Preventing Duplicates and Zero-length Product Names

⌚ 15 to 25 minutes

In this exercise, you will finalize the shopping list by preventing duplicate values and empty strings from being added to the “active-items-list” list.

1. There are a couple of issues still. Open `HTMLDOM/Exercises/shopping-list.html` in your browser.
2. Add Milk via the **Common Items** list and then try adding it again using the **New Item** form field. Milk will be listed twice in your **Active List**. We'll fix that.
3. Press the **+** button next to the empty **New Item** form field. It will add an empty item to your **Active List**. We'll fix that too.
4. Open `HTMLDOM/Exercises/shopping-list.js` in your editor if it isn't already open.
5. Below the line in which you trim the product name, add code that checks if that product is already listed in the “active-items-list” list. If it is or if the trimmed product name is an empty string, return `false` so that the rest of the code in the function doesn't run.
6. Test your code in the browser.
 - A. Add Milk via the **Common Items** list and then try adding it again using the **New Item** form field. It should fail silently.
 - B. Press the **+** button next to the empty **New Item** form field. It should fail silently.
7. If your code isn't working, use the console to help you debug.

Solution: HTMLDOM/Solutions/shopping-list.js

```
-----Lines 1 through 24 Omitted-----
25.  /* Add product to list */
26.  function addToList(product) {
27.      product = product.trim();
28.
29.      // Check if list item is already in active list
30.      // or if product is empty string.
31.      let selector = '#active-items-list>li[title="' + product + '"]';
32.      const liMatch = document.querySelector(selector);
33.      if (liMatch || !product.length) {
34.          return false;
35.      }
36.      const activeList = document.getElementById('active-items-list');
37.      const newItem = document.createElement('li');
38.      newItem.title = product;
39.      newItem.innerHTML = product + ' ';
40.      activeList.appendChild(newItem);
41.      log(product + ' added.');
```

Evaluation Copy

```
42.
43.      const btnRemove = document.createElement('button');
44.      btnRemove.innerHTML = '-';
45.      btnRemove.addEventListener('click', removeFromList);
46.      newItem.appendChild(btnRemove);
47.
48.      // Check if list item being added is in common list items
49.      // If it is, we need to disable its button there.
50.      selector = '#common-items-list>li>button[name="' + product + '"]';
51.      const btnMatch = document.querySelector(selector);
52.      if (btnMatch) {
53.          btnMatch.disabled = true;
54.      }
55.  }
```

```
-----Lines 56 through 94 Omitted-----
```

EVALUATION COPY: Not to be used in class.

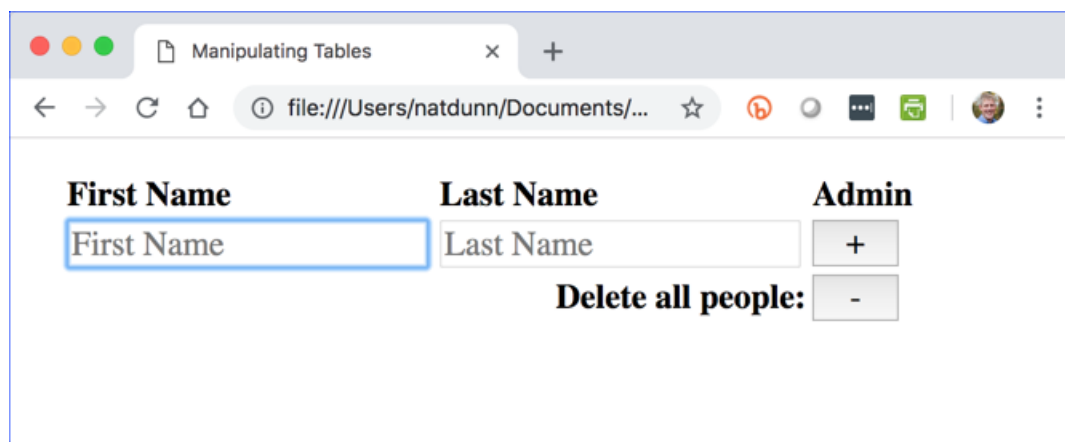


7.11. Manipulating Tables

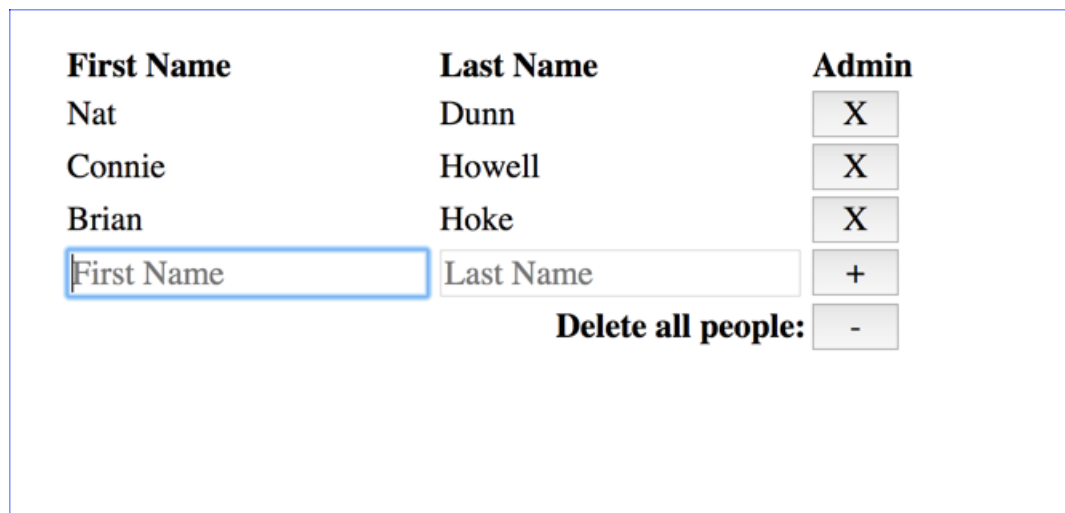
HTML tables can be created and manipulated dynamically with JavaScript. Each `table`, `tbody`, `thead`, and `tfoot` element contains a `rows` array and methods for inserting and deleting rows: `insertRow()` and `deleteRow()`. Each `tr` element contains a `cells` array and methods for inserting and deleting cells: `insertCell()` and `deleteCell()`. The following example shows how these objects can be used to dynamically create HTML tables.

First let's take a look at how the page works in the browser. Open `HTMLDOM/Demos/table.html` in your browser to follow along.

1. When it first loads, you see a screen like this:



2. Fill in the form and press the + sign several times:



3. Press the **X** next to one of the rows to delete that row:

First Name	Last Name	Admin
Connie	Howell	<input type="button" value="X"/>
Brian	Hoke	<input type="button" value="X"/>
<input type="text" value="First Name"/>	<input type="text" value="Last Name"/>	<input type="button" value="+"/>
Delete all people:		<input type="button" value="-"/>

4. Press the **-** next to **Delete all people** to remove all rows and get back to where we started:

First Name	Last Name	Admin
<input type="text" value="First Name"/>	<input type="text" value="Last Name"/>	<input type="button" value="+"/>
Delete all people:		<input type="button" value="-"/>

Now let's look at the code:

Demo 7.7: HTMLDOM/Demos/table.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.  function addRow(tbodyId, cells) {
10.    // Get the tbody and insert a new row
11.    const tbody = document.getElementById(tbodyId);
12.    const newRow = tbody.insertRow();
13.
14.    // Insert cells based on passed-in cells array
15.    for (const cellText of cells) {
16.      cell = newRow.insertCell();
17.      cell.innerHTML = cellText;
18.    }
19.
20.    // Insert a final cell with a Delete button
21.    newCell = newRow.insertCell();
22.    const btnDelete = document.createElement('button');
23.    btnDelete.innerHTML = 'X';
24.    btnDelete.addEventListener('click', function(e) {
25.      btnDelete.parentNode.parentNode.remove();
26.    });
27.    newCell.appendChild(btnDelete);
28.  }
29.
30.  function deleteAllRows(tbodyId) {
31.    const tbody = document.getElementById(tbodyId);
32.    while (tbody.rows.length > 0) {
33.      tbody.deleteRow(0);
34.    }
35.  }
36.
37.  function prepareCells(fName, lName) {
38.    //Create a cells array to pass to the
39.    const cells = [fName.value, lName.value];
40.    addRow('people', cells);
41.    fName.value = '';
42.    lName.value = '';
43.    fName.focus();
44.  }
```

```

45.
46. window.addEventListener('load', function() {
47.     const btnAdd = document.getElementById("btn-add");
48.     const btnDeleteAll = document.getElementById("btn-delete-all");
49.     const fName = document.getElementById('firstname');
50.     const lName = document.getElementById('lastname');
51.
52.     btnAdd.addEventListener('click', function() {
53.         prepareCells(fName, lName);
54.     });
55.
56.     lName.addEventListener('keyup', function(e) {
57.         if (e.key === 'Enter') {
58.             prepareCells(fName, lName);
59.         }
60.     });
61.
62.     btnDeleteAll.addEventListener('click', function() {
63.         deleteAllRows('people');
64.     });
65.
66.     fName.focus();
67. });
68. </script>
69. <title>Manipulating Tables</title>
70. </head>
71. <body id="table-demo">
72. <main>
73. <table>
74.     <thead>
75.         <tr>
76.             <th>First Name</th>
77.             <th>Last Name</th>
78.             <th>Admin</th>
79.         </tr>
80.     </thead>
81.     <tbody id="people"></tbody>
82.     <tbody>
83.         <tr>
84.             <td><input id="firstname" placeholder="First Name"></td>
85.             <td><input id="lastname" placeholder="Last Name"></td>
86.             <td><button type="button" id="btn-add">+</button></td>
87.         </tr>
88.     </tbody>
89. </tfoot>

```

```
90.     <tr>
91.         <th colspan="2">Delete all people:</th>
92.         <td><button type="button" id="btn-delete-all"></button></td>
93.     </tr>
94. </tfoot>
95. </table>
96. </main>
97. </body>
98. </html>
```

The body of the page contains a table with a `thead` that contains a single row of headers:

```
<thead>
  <tr>
    <th>First Name</th>
    <th>Last Name</th>
    <th>Admin</th>
  </tr>
</thead>
```

Below the `thead` are two `tbody` elements.

1. The first is empty and has an `id` of “people”. We will add and remove people from this `tbody`.
2. The second contains form elements for adding new rows:

```
<tr>
  <td><input id="firstname" placeholder="First Name"></td>
  <td><input id="lastname" placeholder="Last Name"></td>
  <td><button type="button" id="btn-add">+</button></td>
</tr>
```

Below the `tbody` elements is a `tfoot` element with a button for deleting all rows.

The JavaScript contains two generic functions: `addRow()` and `deleteAllRows()`. By “generic”, we mean that these functions are not tied to this application. They could be used with any table.

The `addRow()` function takes two parameters: the `id` of the `tbody` element to which to add the row and an array of strings to populate the new row’s cells:

```
function addRow(tbodyId, cells) {
  // Get the tbody and insert a new row
  const tbody = document.getElementById(tbodyId);
  const newRow = tbody.insertRow();

  // Insert cells based on passed-in cells array
  for (const cellText of cells) {
    cell = newRow.insertCell();
    cell.innerHTML = cellText;
  }

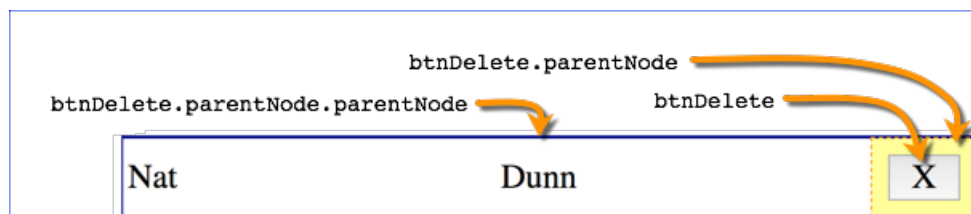
  // Insert a final cell with a Delete button
  newCell = newRow.insertCell();
  const btnDelete = document.createElement('button');
  btnDelete.innerHTML = 'X';
  btnDelete.addEventListener('click', function(e) {
    btnDelete.parentNode.parentNode.remove();
  });
  newCell.appendChild(btnDelete);
}
```

Note this line of code:

Evaluation Copy

```
btnDelete.parentNode.parentNode.remove();
```

The first `parentNode` is the cell that contains `btnDelete`. The second `parentNode` is the row that contains that cell. That is the row that we are removing. We've added some styling below to make this easier to see:



The `deleteAllRows()` function takes one parameter: the id of the `tbody` element containing the rows to be deleted. It then uses a `while` loop to delete the first row over and over until there are no rows left:

```
function deleteAllRows(tbodyId) {  
  const tbody = document.getElementById(tbodyId);  
  while (tbody.rows.length > 0) {  
    tbody.deleteRow(0);  
  }  
}
```

The other JavaScript wires up the `eventListeners` and prepares the cells for passing data to `addRow()`.

You may wish to practice inserting removing rows and cells using Chrome DevTools Console. Just open the `HTMLODOM/Demos/table.html` file in Google Chrome, add some rows through the form, then open the console and see if you can add and remove individual rows and cells with JavaScript.

Conclusion

In this lesson, you have learned to work with the HTML DOM to create and modify HTML page elements dynamically with JavaScript.

LESSON 8

CSS Object Model

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☑ Changing values of CSS properties dynamically.
- ☑ Hiding and showing elements.

Introduction

We can use JavaScript to both retrieve information about an element's CSS styles and to set those styles programmatically.

EVALUATION COPY: Not to be used in class.



8.1. Changing CSS with JavaScript

Throughout this course we have used JavaScript to change background colors using the following syntax:

```
element.style.backgroundColor = value;
```

But we can do a lot more than change the background color. We can both get and set any styles for most any element.

Each CSS property has a corresponding property of the JavaScript style object:

- If the CSS property is a simple word (e.g., color) then the JavaScript property is the same (e.g., style.color).

- If the CSS property has a dash in it (e.g., background-color) then the JavaScript property uses lower camel case (e.g., style.backgroundColor).

The `style` object is a collection of an element's styles that are either defined within that HTML element's `style` attribute or directly in JavaScript. Styles defined in the `<style>` tag or in an external stylesheet are not part of the `style` object.

The W3C specifies a method for getting at the current (or actual) style of an object: the window object's `getComputedStyle()` method.

```
window.getComputedStyle(element)
```

Note that the reference to `window` can be excluded as `window` is the implicit object. For example:

```
const contactForm = document.getElementById("contact-form");  
const computedStyle = getComputedStyle(contactForm);
```

Using this method - with `getComputedStyle()`, as opposed to `element.style` - we can get the styles set with inline CSS (i.e., within `style` attributes), with embedded CSS (i.e., within `<style>` tags), or with external (i.e., linked) stylesheets. Furthermore, as the name of the method `getComputedStyle()` suggests, these are computed (calculated) styles: whereas `element.style` just gives us style info as set in CSS, `getComputedStyle()` gives us the real-time calculated CSS.

Let's take a look at a simple example to make this more clear.

Demo 8.1: CSSObjectModel/Demos/board.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <style>
9.  div#board {
10.     margin:auto;
11.     width:306px;
12.  }
13.
14.  div.row {
15.     height:100px;
16.  }
17.
18.  div.col {
19.     border:1px solid black;
20.     float:left;
21.     font-size:xx-large;
22.     height:100px;
23.     text-align:center;
24.     width:100px;
25.  }
26. </style>
27. <title>Board</title>
28. </head>
29. <body>
30. <main>
31.   <div id="board" style="font-style:italic;">
32.     <div class="row">
33.       <div class="col">A</div>
34.       <div class="col">B</div>
35.       <div class="col">C</div>
36.     </div>
37.     <div class="row">
38.       <div class="col">D</div>
39.       <div class="col">E</div>
40.       <div class="col">F</div>
41.     </div>
42.     <div class="row">
43.       <div class="col">G</div>
44.       <div class="col">H</div>
```

Evaluation
Copy

```
45.     <div class="col">I</div>
46.     </div>
47. </div>
48. </main>
49. </body>
50. </html>
```

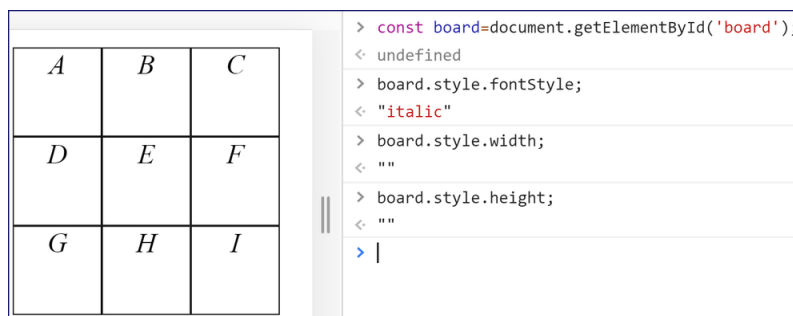
Things to notice:

1. We have explicitly set the font-style of div#board to “italic” using its style attribute.
2. We have explicitly set the width of div#board to “306px” using an embedded stylesheet.
3. We have **not** set the height of div#board.

With that file open, do the following at Chrome DevTools Console:

1. Type `const board = document.getElementById('board');` and press **Enter**.
2. Type `board.style.fontStyle` and press **Enter**. Notice that it outputs “italic”. That’s because we set the font-style using the style attribute. An element’s style property in JavaScript only has access to properties set using the style attribute.
3. Type `board.style.width` and press **Enter**. Notice that it returns an empty string. That’s because we set the width outside of the style attribute.
4. Type `board.style.height` and press **Enter**. Notice that it again returns an empty string. That’s because we haven’t set the height in the style attribute, or anywhere else for that matter.

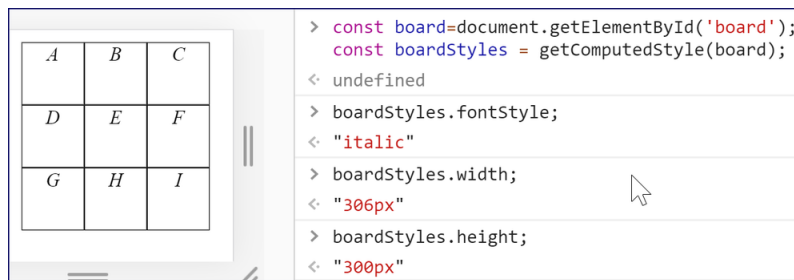
Here is a screenshot showing those results:



Now let’s use `getComputedStyle()` instead.

1. Type `const board = document.getElementById('board');` and press **Enter**.
2. Type `const boardStyles = getComputedStyle(board);` and press **Enter**.
3. Type the following and notice it outputs the computed value each time:
 - A. `boardStyles.fontStyle` and press **Enter**.
 - B. `boardStyles.width` and press **Enter**.
 - C. `boardStyles.height` and press **Enter**.

Here is a screenshot showing those results:



❖ 8.1.1. The style Property vs. getComputedStyle()

The takeaway from all this is:

- Use `getComputedStyle(elem)` to **get** the style of an element.
- Use `elem.style` to **set** the style of an element.

Dot Notation vs. Square Bracket Notation

It is common to use dot notation whenever possible, and in the examples above, we have done so. However, it is worth noting that you can also use square bracket notation. For example, the two statements below are equivalent:

```
getComputedStyle(board).fontStyle;
getComputedStyle(board)['fontStyle'];
```

As we will see later, when the name of the style property is stored in a variable, you can only use square bracket notation. For example:

```
const styleProp = 'fontStyle';
// The Correct Way
getComputedStyle(board)[styleProp];

// Incorrect as styleProp is undefined in getComputedStyle(board)
getComputedStyle(board).styleProp;
```

EVALUATION COPY: Not to be used in class.



8.2. Hiding and Showing Elements

Elements can be hidden and shown by changing their `visibility` or `display` values.

The `visibility` property can be set to `"visible"` or `"hidden"` and the `display` property can be set to `"block"`, `"table-row"`, `"list-item"`, `"none"`, and many other values.

When an element's `visibility` is set to `"hidden"`, it disappears, but it continues to occupy its space. In the following table, the second row's `visibility` is set to `"hidden"`:

tr2 visibility made hidden	
Row 1	
Row 3	
Row 4	

When an element's `display` is set to `"none"`, it disappears, and it no longer occupies any space on the page. In the following table, the second row's `display` is set to `"none"`:

tr2 display made none	
Row 1	
Row 3	
Row 4	

Take a look at the code in the file below, which shows two different ways of showing and hiding table rows:

Evaluation
Copy

Demo 8.2: CSSObjectModel/Demos/visibility.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.  function toggleVisibility(e) {
10.   const button = e.currentTarget;
11.   const elem = document.getElementById(button.title);
12.   if (getComputedStyle(elem).visibility === "visible") {
13.     elem.style.visibility = "hidden";
14.   } else {
15.     elem.style.visibility = "visible";
16.   }
17.   const computedStyle = getComputedStyle(elem);
18.   msg(button.title, 'visibility', computedStyle.visibility);
19. }
20.
21. function toggleDisplay(e) {
22.   const button = e.currentTarget;
23.   const elem = document.getElementById(button.title);
24.   if (elem.style.display === "none") {
25.     elem.style.display = "table-row";
26.   } else {
27.     elem.style.display = "none";
28.   }
29.   const computedStyle = getComputedStyle(elem);
30.   msg(button.title, 'display', computedStyle.display);
31. }
32.
33. function msg(elemId, styleProp, styleValue) {
34.   document.getElementById("msg").style.display = "block";
35.   document.getElementById("elemId").innerHTML = elemId;
36.   document.getElementById("styleProp").innerHTML = styleProp;
37.   document.getElementById("styleValue").innerHTML = styleValue;
38. }
39.
40. window.addEventListener('load', function() {
41.   const btnsVisibility = document.getElementsByClassName('visibility');
42.   for (button of btnsVisibility) {
43.     button.addEventListener('click', toggleVisibility);
44.   }
```

```

45.
46.     const btnsDisplay = document.getElementsByClassName('display');
47.     for (button of btnsDisplay) {
48.         button.addEventListener('click', toggleDisplay);
49.     }
50. })
51. </script>
52. <title>Showing and Hiding Elements with JavaScript</title>
53. </head>
54. <body>
55. <main>
56.     <h1>Hiding and Showing Elements</h1>
57.     <table>
58.         <caption id="msg">
59.             <span id="elemId"></span> <em id="styleProp"></em>
60.             made <span id="styleValue"></span>
61.         </caption>
62.         <tr id="tr1"><td>Row 1</td></tr>
63.         <tr id="tr2"><td>Row 2</td></tr>
64.         <tr id="tr3"><td>Row 3</td></tr>
65.         <tr id="tr4"><td>Row 4</td></tr>
66.     </table>
67.
68.     <h2>visibility</h2>
69.     <button title="tr1" class="visibility">Row 1</button>
70.     <button title="tr2" class="visibility">Row 2</button>
71.     <button title="tr3" class="visibility">Row 3</button>
72.     <button title="tr4" class="visibility">Row 4</button>
73.
74.     <h2>display</h2>
75.     <button title="tr1" class="display">Row 1</button>
76.     <button title="tr2" class="display">Row 2</button>
77.     <button title="tr3" class="display">Row 3</button>
78.     <button title="tr4" class="display">Row 4</button>
79. </main>
80. </body>
81. </html>

```

This page has two functions for changing whether a table row appears: `toggleVisibility()` and `toggleDisplay()`.

1. The `toggleVisibility()` function checks the computed value of the `visibility` property of a table row. If it is "visible", it sets it to "hidden". Otherwise, it sets it to "visible".

2. The `toggleDisplay()` function checks the computed value of the `display` property of a table row. If it is `"none"`, it sets it to `"table-row"`. Otherwise, it sets it to `"none"`.

EVALUATION COPY: Not to be used in class.



8.3. Checking and Changing Other Style Properties

You can check and change other style properties in the same way we have done with `display` and `visibility`. For many properties, you can change styles using this function:

```
function changeStyle(elem, styleProp, styleValue) {  
    elem.style[styleProp] = styleValue; // set style  
    return getComputedStyle(elem)[styleProp]; // return new style  
}
```

There may be no need to return the new style, but it might be interesting in case you want to log it from the calling code.

Here are some sample calls to `changeStyle()` to change the `h1` element of a page. Open `CSSObjectModel/Demos/visibility.html` in Google Chrome and try this yourself:

<div>Hiding and Showing Elements</div>	<pre>> function changeStyle(elem, styleProp, styleValue) { elem.style[styleProp] = styleValue; return getComputedStyle(elem)[styleProp]; } < undefined</pre>
	<pre>> const h1 = document.querySelector('h1'); changeStyle(h1, 'width', '120px'); < "120px"</pre>
	<pre>> changeStyle(h1, 'margin', '85px 15px'); < "85px 15px"</pre>
	<pre>> changeStyle(h1, 'float', 'right'); < "right"</pre>
	<pre>> changeStyle(h1, 'textAlign', 'center'); < "center"</pre>
	<pre>> changeStyle(h1, 'padding', '10px'); < "10px"</pre>
	<pre>> changeStyle(h1, 'fontSize', 'x-large'); < "24px"</pre>
	<pre>> changeStyle(h1, 'backgroundColor', 'yellow'); < "rgb(255, 255, 0)"</pre>
	<pre>> changeStyle(h1, 'border', '1px dashed #f66'); < "1px dashed rgb(255, 102, 102)"</pre>

We have created a page to allow you to practice calling the `changeStyle()` function.

Demo 8.3: CSSObjectModel/Demos/changing-styles-simple.html

```

1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.    function changeStyle(elem, styleProp, styleValue) {
10.      elem.style[styleProp] = styleValue; // set style
11.      return getComputedStyle(elem)[styleProp]; // return new style
12.    }
13.  </script>
14.  <title>Changing Styles</title>
15.  </head>
16.  <body id="changing-styles">
17.  <main>
18.    <p id="hello-world">Hello, World!</p>
19.  </main>
20.  </body>
21.  </html>

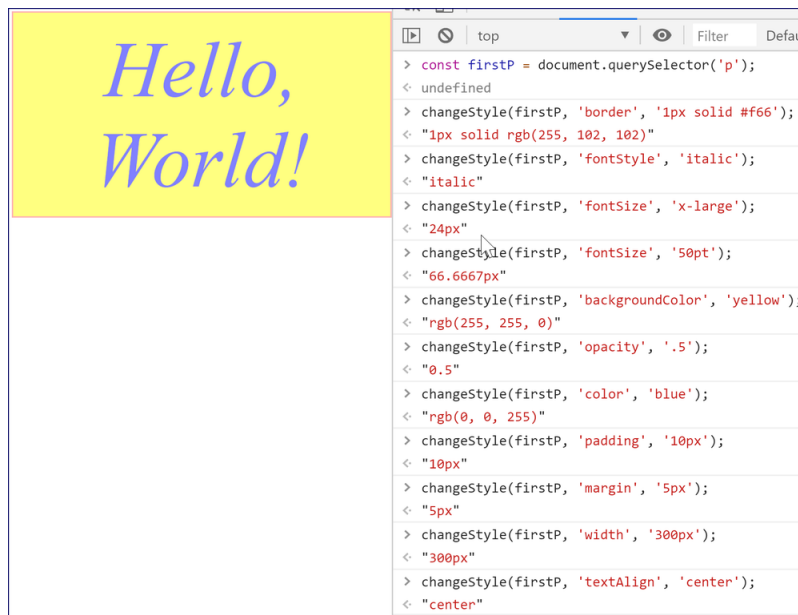
```

1. Open `CSSObjectModel/Demos/changing-styles-simple.html` in your browser and open the console.
2. Enter the following to assign the first paragraph to a variable:

```
const firstP = document.querySelector('p');
```

3. See if you can use the `changeStyle()` function to change some of the first paragraph's styles.

Here's how we did it:



EVALUATION COPY: Not to be used in class.



8.4. Increasing and Decreasing Measurements

1. Open `CSSObjectModel/Demos/changing-styles-simple.html` in your browser if it's not already open.

2. Enter the following at the console:

```
const firstP = document.querySelector('p');  
getComputedStyle(firstP).width;
```

3. Notice that the value returned ends in “px” (e.g., “300px”).

Values for `margin`, `padding`, `height`, `width`, `fontSize` and other properties are strings and often include the unit (e.g., “px”). If you want to increment or decrement these values, you first have to extract the number from the style value and then, after doing the math, add back the unit. For example, let’s say you want to increase the font size of a paragraph by two pixels. The steps to do that would be:

1. Get the current font size. Let’s say it is 20 pixels. That will be returned as “20px”.
2. Use `parseInt()` to extract the number from that and turn it into an integer.
3. Add 2 to the result.
4. Use `String()` to change the number back to a string and append “px” to the end.

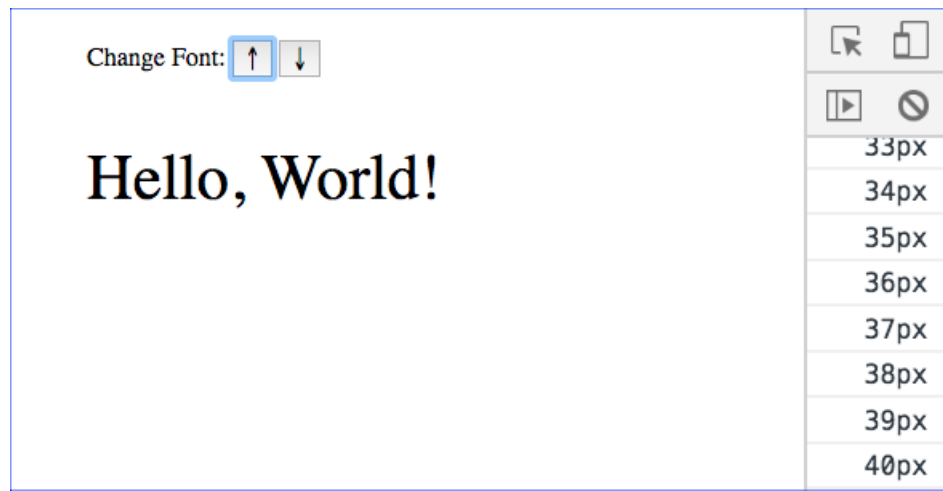
The following example shows how to do this with the `fontSize` property.

Demo 8.4: CSSObjectModel/Demos/changing-font-size.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      function changeFontSize(elem, change) {
10.         // Use parseInt to remove the unit and return an integer
11.         const curFontSize = parseInt(getComputedStyle(elem).fontSize);
12.
13.         // Add change to curFontSize to get newFontSize
14.         const newFontSize = curFontSize + change;
15.
16.         // Set the fontSize by converting newFontSize
17.         // to a string and appending unit
18.         elem.style.fontSize = String(newFontSize) + 'px';
19.         return getComputedStyle(elem).fontSize;
20.     }
21.
22.     window.addEventListener('load', function() {
23.         const p = document.getElementById('hello-world');
24.         const btnIncrease = document.getElementById('increase');
25.         const btnDecrease = document.getElementById('decrease');
26.
27.         btnIncrease.addEventListener('click', function() {
28.             // Increase font size by 1 unit
29.             const fontSize = changeFontSize(p, 1);
30.             console.log(fontSize);
31.         });
32.
33.         btnDecrease.addEventListener('click', function() {
34.             // Decrease font size by 1 unit
35.             const fontSize = changeFontSize(p, -1);
36.             console.log(fontSize);
37.         });
38.     })
39. </script>
40. <title>Changing Font Size</title>
41. </head>
42. <body id="changing-styles">
43. <main>
44.     <label>Change Font:</label>
```

```
45. <button id="increase">&uparrow;</button>
46. <button id="decrease">&downarrow;</button>
47. <p id="hello-world">Hello, World!</p>
48. </main>
49. </body>
50. </html>
```

After hitting the up arrow a bunch of times, the page will look like this:



❖ 8.4.1. Making changeFontSize() More Flexible

Currently, the `changeFontSize()` function only works for incrementing and decrementing the font size. The following code makes it more flexible, so that it will work with incrementing or decrementing any property value that is set in pixels:

```
function changeStyleWithPx(elem, styleProp, change) {
  const curStyleValue = parseInt(getComputedStyle(elem)[styleProp]);
  const newStyleValue = curStyleValue + change;
  elem.style[styleProp] = String(newStyleValue) + 'px';
  console.log(styleProp + ': ' + elem.style[styleProp]);
}
```

Two things to notice:

1. The function now includes a `styleProp` parameter to take the style property being changed.

2. We use square-bracket notation rather than dot notation for the style properties. For example, instead of `getComputedStyle(elem).styleProp`, we use `getComputedStyle(elem)[styleProp]`. That is because `styleProp` is not a property of `getComputedStyle(elem)`, but rather a variable holding a string. If we pass `'fontStyle'` as the `styleProp`, `getComputedStyle(elem)[styleProp]` gets interpreted as `getComputedStyle(elem)['fontStyle']`. The code `getComputedStyle(elem).styleProp` would return undefined because, again, `styleProp` is not a property of `getComputedStyle(elem)`.

EVALUATION COPY: Not to be used in class.



8.5. Custom data Attributes

HTML tags can take *custom data attributes* that take the form of *data-attribute-name*. These attributes are used to provide additional information about an element and are available through the special `dataset` property of element nodes. To illustrate, let's look at an example we saw earlier in the course:

Demo 8.5: CSSObjectModel/Demos/add-event-listener.html

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.  function changeBg(colorOrEvent) {
10.    let color = 'white'; // default
11.    if ( typeof colorOrEvent === 'string' ) {
12.      color = colorOrEvent;
13.    } else {
14.      color = colorOrEvent.currentTarget.id;
15.    }
16.    document.body.style.backgroundColor = color;
17.  }
18.
19.  function changeBgWhite(e) {
20.    changeBg('white');
21.  }
-----Lines 22 through 55 Omitted-----
```

Notice that we key off the `id` values to set the background color. That's not a great way to handle this as the `id` value should not be tied to any specific functionality.

Take a look at the following, which solves this problem using custom data attributes:

Demo 8.6: [CSSObjectModel/Demos/add-event-listener-improved.html](#)

```
-----Lines 1 through 8 Omitted-----
9.  function changeBg(e) {
10.    const eventType = e.type;
11.    const target = e.currentTarget;
12.    switch(eventType) {
13.      case 'click':
14.      case 'dblclick':
15.      case 'mousedown':
16.      case 'mouseover':
17.        color = target.dataset.activeColor;
18.        break;
19.      case 'mouseup':
20.      case 'mouseout':
21.        color = target.dataset.inactiveColor;
22.        break;
23.      default:
24.        color = 'white';
25.    }
26.    document.body.style.backgroundColor = color;
27.  }

-----Lines 28 through 47 Omitted-----
48.  <main>
49.    <button id="btn-red" data-active-color="red">
50.      Click to turn the page red.
51.    </button>
52.    <button id="btn-green" data-active-color="green">
53.      Double-click to turn the page green.
54.    </button>
55.    <button id="btn-orange" data-active-color="orange"
56.      data-inactive-color="white">
57.      Click and hold to turn the page orange.
58.    </button>
59.    <a href="#" id="link-pink" data-active-color="pink"
60.      data-inactive-color="white">Hover over to turn page pink.</a>
61.  </main>

-----Lines 62 through 63 Omitted-----
```

Things to notice:

1. We have given the `id` attributes new values.
2. We have added new `data-active-color` and `data-inactive-color` attributes to the controls.
3. The `changeBg()` function now gets the event type from the passed-in event and keys off of it to decide whether to use the `activeColor` value (`click`, `dblclick`, `mousedown`, and `mouseover`) or to use the `inactiveColor` value (`mouseout` and `mouseup`). For all other event types (e.g., `keyup`), it sets `color` to 'white.'
4. This switch-case statement might look a little funny to you as it appears that nothing happens for many of the cases (e.g., 'click', and 'dblclick'). Remember that cases continue to be evaluated until a `break` statement is reached, so `color = target.dataset.activeColor;` runs for all of the first four cases:

```
case 'click':  
case 'dblclick':  
case 'mousedown':  
case 'mouseover':  
    color = target.dataset.activeColor;  
    break;
```

Review the following code to see some of these concepts used in practice. Be sure to open `CSSObject Model/Demos/changing-styles.html` in the browser and play around.

Demo 8.7: CSSObjectModel/Demos/changing-styles.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script>
9.   function changeStyle(elem, styleProp, styleValue) {
10.     elem.style[styleProp] = styleValue;
11.     console.log(styleProp + ': ' + elem.style[styleProp]);
12.   }
13.
14.   function changeStyleWithPx(elem, styleProp, change) {
15.     const curStyleValue = parseInt(getComputedStyle(elem)[styleProp]);
16.     const newStyleValue = curStyleValue + change;
17.     elem.style[styleProp] = String(newStyleValue) + 'px';
18.     console.log(styleProp + ': ' + elem.style[styleProp]);
19.   }
20.
21.   window.addEventListener('load', function() {
22.     const p = document.getElementById('hello-world');
23.     let selector;
24.
25.     selector = 'button[data-font-style]';
26.     const bsFtStyle = document.querySelectorAll(selector);
27.     for (btn of bsFtStyle) {
28.       btn.addEventListener('click', function(e) {
29.         const fontStyle = e.currentTarget.dataset.fontStyle;
30.         changeStyle(p, 'fontStyle', fontStyle);
31.       });
32.     }
33.
34.     selector = 'button[data-font-size]';
35.     const bsFtSize = document.querySelectorAll(selector);
36.     for (btn of bsFtSize) {
37.       btn.addEventListener('click', function(e) {
38.         const fontSize = e.currentTarget.dataset.fontSize;
39.         changeStyle(p, 'fontSize', fontSize);
40.       });
41.     }
42.
43.     selector = 'button[data-border-style]';
44.     const bsBrdStyle = document.querySelectorAll(selector);
```

```

45.   for (btn of bsBrdStyle) {
46.       btn.addEventListener('click', function(e) {
47.           const borderStyle = e.currentTarget.dataset.borderStyle;
48.           changeStyle(p, 'borderStyle', borderStyle);
49.       });
50.   }
51.
52.   selector = 'button[data-padding]';
53.   const bsPadding = document.querySelectorAll(selector);
54.   for (btn of bsPadding) {
55.       btn.addEventListener('click', function(e) {
56.           const target = e.currentTarget;
57.           // Use ternary operator to assign change value
58.           const change = target.dataset.padding==='increase' ? 10 : -10;
59.           changeStyleWithPx(p, 'padding', change);
60.       });
61.   }
62.
63.   selector = 'button[data-margin]';
64.   const bsMargin = document.querySelectorAll(selector);
65.   for (btn of bsMargin) {
66.       btn.addEventListener('click', function(e) {
67.           const target = e.currentTarget;
68.           // Use ternary operator to assign change value
69.           const change = target.dataset.margin==='increase' ? 10 : -10;
70.           changeStyleWithPx(p, 'margin', change);
71.       });
72.   }
73. });
74. </script>
75. <title>Changing Styles</title>
76. </head>
77. <body id="changing-styles">
78. <main>
79.   <div>
80.     <label>fontStyle:</label>
81.     <button class="font-style italic" data-font-style="italic">Italic</button>
82.     <button class="font-style normal" data-font-style="normal">Normal</button>
83.   </div>
84.   <div>
85.     <label>fontSize:</label>
86.     <button class="font-size xx-small" data-font-size="xx-small">A</button>
87.     <button class="font-size x-small" data-font-size="x-small">A</button>
88.     <button class="font-size small" data-font-size="small">A</button>
89.     <button class="font-size medium" data-font-size="medium">A</button>

```

```
90.     <button class="font-size large" data-font-size="large">A</button>
91.     <button class="font-size x-large" data-font-size="x-large">A</button>
92.     <button class="font-size xx-large" data-font-size="xx-large">A</button>
93. </div>
94. <div>
95.     <label>borderStyle:</label>
96.     <button class="border-style none" data-border-style="none">none</button>
97.     <button class="border-style dotted" data-border-style="dotted">dotted</button>
98.     <button class="border-style dashed" data-border-style="dashed">dashed</button>
99.     <button class="border-style solid" data-border-style="solid">solid</button>
100. </div>
101. <div>
102.     <label>padding:</label>
103.     <button class="padding increase" data-padding="increase">Increase</button>
104.     <button class="padding decrease" data-padding="decrease">Decrease</button>
105. </div>
106. <div>
107.     <label>margin:</label>
108.     <button class="margin increase" data-margin="increase">Increase</button>
109.     <button class="margin decrease" data-margin="decrease">Decrease</button>
110. </div>
111. <p id="hello-world">Hello, World!</p>
112. </main>
113. </body>
114. </html>
```

EVALUATION COPY: Not to be used in class.

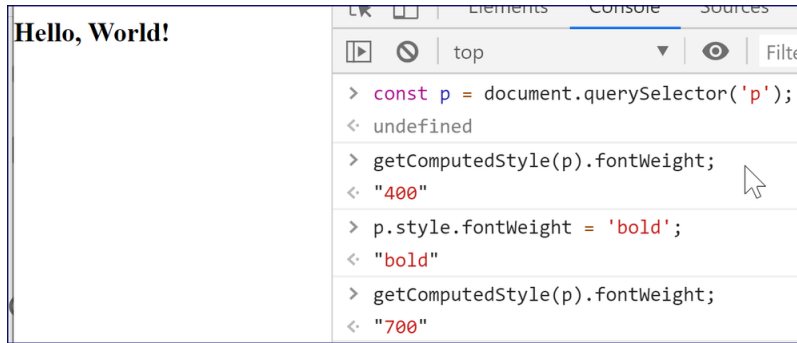


8.6. Gotcha with fontWeight

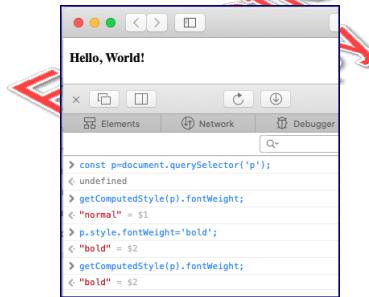
These days, the most popular browsers tend to be very much in sync with their support and implementation of CSS and JavaScript; however, Google Chrome and Safari return different values when reading the computed `fontWeight` style property.

- Chrome returns the weight as a number: “400” for “normal” and “700” for “bold”.
- Safari returns the weight as a keyword: “normal” and “bold”.

The following two screenshots illustrate this:



fontWeight in Google Chrome



fontWeight in Safari

As a result, when checking to see if an element is bold or not, you need to write code that checks both possibilities. The following demo shows a function for handling this:

Demo 8.8: CSSObjectModel/Demos/changing-font-weight.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      function toggleBold(elem) {
10.         const weight = getComputedStyle(elem).fontWeight;
11.         console.log("Old weight: " + weight);
12.         if (weight === 'bold' || weight > 400) {
13.             elem.style.fontWeight = "normal";
14.         } else {
15.             elem.style.fontWeight = "bold";
16.         }
17.         console.log("New weight: " + getComputedStyle(elem).fontWeight);
18.     }
19.
20.     window.addEventListener('load', function() {
21.         const p = document.getElementById('hello-world');
22.         const btn = document.getElementById('font-weight');
23.         btn.addEventListener('click', function() {
24.             toggleBold(p);
25.         });
26.     });
27. </script>
28. <title>Changing Font Weight</title>
29. </head>
30. <body id="changing-styles">
31. <main>
32.     <label>Change Font Weight:</label>
33.     <button id="font-weight">Toggle Font Weight</button>
34.     <p id="hello-world">Hello, World!</p>
35. </main>
36. </body>
37. </html>
```

EVALUATION COPY: Not to be used in class.



8.7. Font Awesome

Font Awesome¹⁰ provides a collection of free vector icons that you can use on your websites. You can get access to these icons through a free content delivery network (CDN). To do so, you will need get your own unique `<script>` tag with the latest version of Font Awesome:

1. Go to <https://fontawesome.com/start>.
2. Enter your email address in the form and submit. You will be sent an email asking you to confirm your email address and create an account. After creating an account, you will be provided with a `<script>` tag that looks something like this:

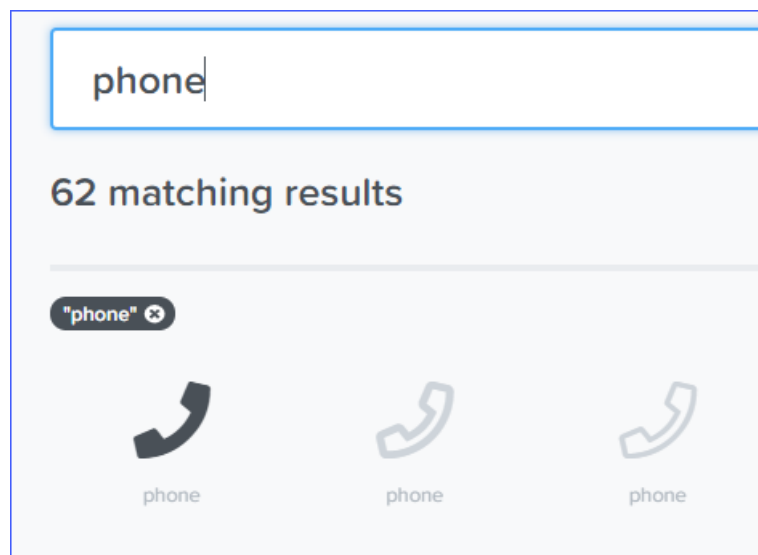
```
<script src="https://kit.fontawesome.com/yoursecretcode.js"
  crossorigin="anonymous"></script>
```

3. Paste this tag into the head of your HTML. At that point, that page will be able to use Font Awesome icons.

❖ 8.7.1. Finding and Using Icons

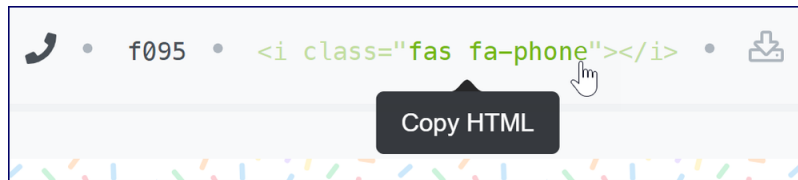
To find Font Awesome icons:

1. Use the search feature at <https://fontawesome.com/icons>:



10. <https://fontawesome.com>

2. Click the icon you want to use.
3. Find the HTML code snippet on the icon page and copy it by clicking the clipboard icon:



4. Then paste that code snippet wherever you want the icon to show up.

Below is a page containing several Font Awesome icons:

Demo 8.9: CSSObjectModel/Demos/font-awesome.html

```

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link rel="stylesheet" href="../normalize.css">
7. <link rel="stylesheet" href="../styles.css">
8. <script src="https://kit.fontawesome.com/43fbe82898.js"
9.   crossorigin="anonymous"></script>
10. <title>Font Awesome</title>
11. </head>
12. <body id="font-awesome">
13. <main>
14.   <i class="fas fa-phone"></i>
15.   <i class="fas fa-envelope-square"></i>
16.   <i class="fas fa-search"></i>
17.   <i class="fab fa-grunt"></i>
18.   <i class="fab fa-facebook"></i>
19.   <i class="fas fa-chevron-up"></i>
20.   <i class="fas fa-chevron-down"></i>
21. </main>
22. </body>
23. </html>

```

The icons on this page are shown below:



Open CSSObjectModel/Demos/font-awesome.html in your browser to see the web page.



8.8. `classList` Property

The `classList` property of an element returns a list of the classes the element contains. This list can be modified using the following methods:

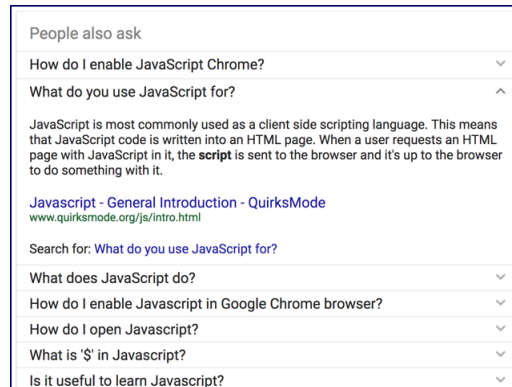
`classList` Methods

Method	Description
<code>add(className)</code>	Adds <code>className</code> class.
<code>remove(className)</code>	Removes <code>className</code> class. Note that this doesn't error if the element doesn't contain the <code>className</code> class.
<code>toggle(className)</code>	If element contains the <code>className</code> class, it removes it. If it doesn't contain the <code>className</code> class, it adds it.
<code>contains(className)</code>	Returns true if the element contains the <code>className</code> class. Otherwise, returns false.
<code>replace(oldClass, newClass)</code>	Replaces <code>oldClass</code> with <code>newClass</code> .

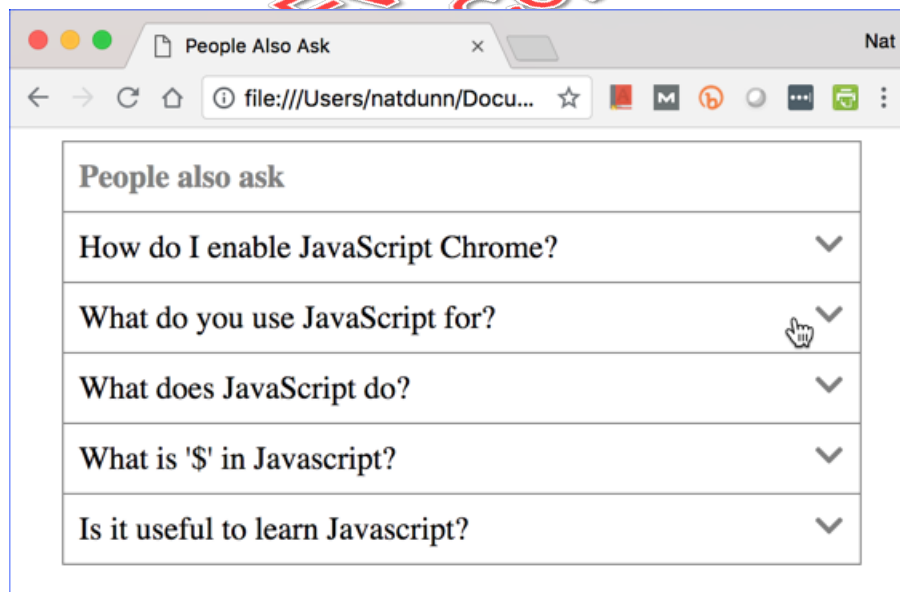
Exercise 21: Showing and Hiding Elements

 20 to 30 minutes

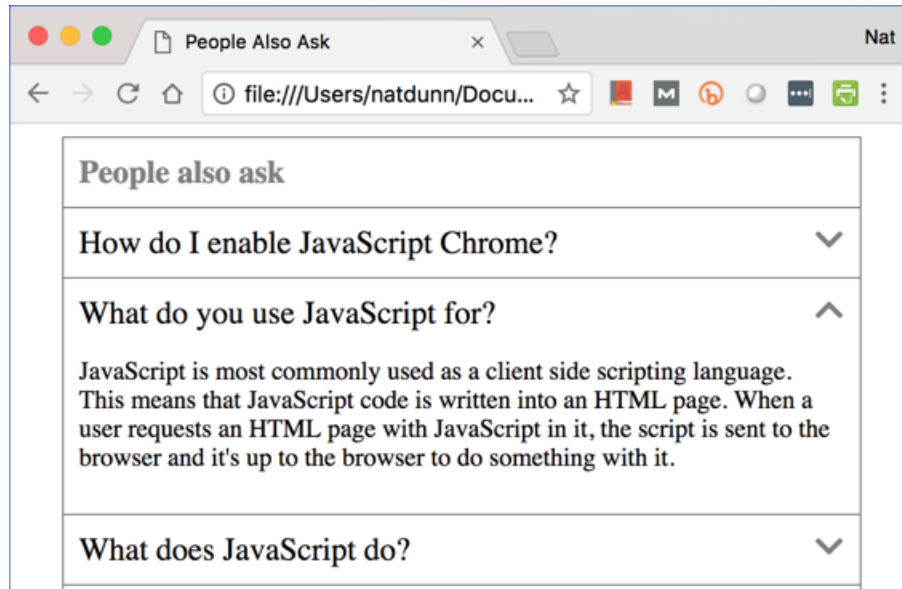
Google search has a “People also ask” feature that shows a list of questions similar to your search. When you click one of the questions the beginning of the answer shows up below it:



In this exercise, you will start with a list of questions with the answers hidden as shown below:



When the user clicks one of the questions, the answer will appear:



The starting code is shown below.

Exercise Code 21.1: CSSObjectModel/Exercises/people-also-ask.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link href="../styles.css" rel="stylesheet">
7. <link rel="stylesheet" href="https://use.fontawesome.com/releas  ↵↵
   es/v5.3.1/css/all.css"
8.   integrity="sha384-mzrmE5qonljUremF  ↵↵
   sqc01SB46JvROS7bZs3IO2EmfFsd15uHvIt+Y8vEf7N7fWAU"
9.   crossorigin="anonymous">
10. <script>
11.   function toggleDisplay(elem) {
12.     // Toggle the display of elem between "block" and "none".
13.   }
14.
15.   function toggleAnswer(e) {
16.     /*
17.      Get the question div that was clicked
18.      and use it to then get the answer div.
19.      Send the answer div to toggleDisplay().
20.      Toggle the class of the chevron i element
21.      between 'fa-chevron-down' and 'fa-chevron-up'.
22.     */
23.   }
24.
25.   window.addEventListener('load', function(e) {
26.     const questions = document.querySelectorAll('.question');
27.     for (question of questions) {
28.       question.addEventListener('click', toggleAnswer);
29.     }
30.   });
31. </script>
32. <title>People Also Ask</title>
33. </head>
34. <body id="people-also-ask">
35. <main>
36.   <h1>People also ask</h1>
37.   <ul id="questions">
38.     <li>
39.       <div class="question">
40.         How do I enable JavaScript Chrome?
41.         <i class="fas fa-chevron-down chevron"></i>
42.       </div>
```

```

43.     <div class="answer">
44.         <p><strong>If you'd like to turn JavaScript
45.     off or on for all sites:</strong></p>
46.         <ol>
47.             <li>Click the Chrome menu in the top right-hand
48.         corner of your browser.</li>
49.             <li>Select Settings.</li>
50.             <li>Click Show advanced settings.</li>
51.             <li>Under the "Privacy" section,
52.         click the Content settings button.</li>
53.         </ol>
54.     </div>
55. </li>
-----Lines 56 through 104 Omitted-----
105. </ul>
106. </main>
107. </body>
108. </html>

```

1. Open CSSObjectModel/Exercises/people-also-ask.html for editing.
2. Write the code for the toggleDisplay() function so that it toggles the value of the display property of the passed-in element between "block" and "none".
3. Write the code for the toggleAnswer() function so that it:
 - A. Gets the question div that was clicked and uses it to then get the answer div. **Hint:** You may want to review Accessing Elements Hierarchically (see page 180).
 - B. Sends the answer div to toggleDisplay().
 - C. Toggles the class of the chevron i element between "fa-chevron-down" and "fa-chevron-up". You should use one or more of the methods for classList.
4. Test your solution in a browser.

Challenge

1. When a question is opened, make the question bold:

People also ask	
How do I enable JavaScript Chrome?	▼
What do you use JavaScript for?	▼
What does JavaScript do?	▲
Used in Web pages, JavaScript is a "client-side" programming language. This means JavaScript scripts are read, interpreted and executed in the client, which is your Web browser. By comparison, "server-side" programming languages run on a remote computer, such as a server hosting a website.	
What is '\$' in Javascript?	▼
Is it useful to learn Javascript?	▼

2. When a question is closed, meaning that's it already been opened and presumably the answer has been seen, change the opacity of the question to .5:

People also ask	
How do I enable JavaScript Chrome?	▼
What do you use JavaScript for?	▼
What does JavaScript do?	▼
What is '\$' in Javascript?	▼
Is it useful to learn Javascript?	▼

Solution: CSSObjectModel/Solutions/people-also-ask.html

```
-----Lines 1 through 8 Omitted-----
9.  <script>
10.    function toggleDisplay(elem) {
11.      if (getComputedStyle(elem).display === "none") {
12.        elem.style.display = "block";
13.      } else {
14.        elem.style.display = "none";
15.      }
16.    }
17.
18.    function toggleAnswer(e) {
19.      const question = e.currentTarget;
20.      const answer = question.nextElementSibling;
21.      toggleDisplay(answer);
22.      const chevron = question.querySelector('.chevron');
23.      chevron.classList.toggle('fa-chevron-down');
24.      chevron.classList.toggle('fa-chevron-up');
25.    }
26.
27.    window.addEventListener('load', function(e) {
28.      const questions = document.querySelectorAll('.question');
29.      for (question of questions) {
30.        question.addEventListener('click', toggleAnswer);
31.      }
32.    });
33.  </script>
-----Lines 34 through 110 Omitted-----
```

Challenge Solution:

CSSObjectModel/Solutions/people-also-ask-challenge.html

```
-----Lines 1 through 18 Omitted-----
19.     const weight = getComputedStyle(elem).fontWeight;
20.     if (weight === 'bold' || weight > 400) {
21.         elem.style.fontWeight = "normal";
22.         markRead(elem);
23.     } else {
24.         elem.style.fontWeight = "bold";
25.     }
26. }
27.
28. function markRead(elem) {
29.     elem.style.opacity = .5;
30. }
31.
32. function toggleAnswer(e) {
33.     const question = e.currentTarget;
34.     const answer = question.nextElementSibling;
35.     toggleDisplay(answer);
36.     toggleBold(question);
37.     const chevron = question.querySelector('.chevron');
38.     chevron.classList.toggle('fa-chevron-down');
39.     chevron.classList.toggle('fa-chevron-up');
40. }
-----Lines 41 through 125 Omitted-----
```

Conclusion

In this lesson, you have learned how to dynamically modify the content of an HTML page and to dynamically modify CSS styles of HTML elements.

LESSON 9

Errors and Exceptions

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☑ Using try/catch/finally to handle errors

Introduction

JavaScript provides several methods for catching and handling errors, the most useful of which is try/catch/finally.

EVALUATION COPY: Not to be used in class.



9.1. Runtime Errors

A runtime error is an error that occurs while a program is being executed. A runtime error can be the result of invalid user input, a browser change, or bad data sent from the server.

It is the programmer's job to anticipate, "catch," and "handle" potential runtime errors.

❖ 9.1.1. Completely Unhandled Errors

Look at this seemingly trivial code sample:

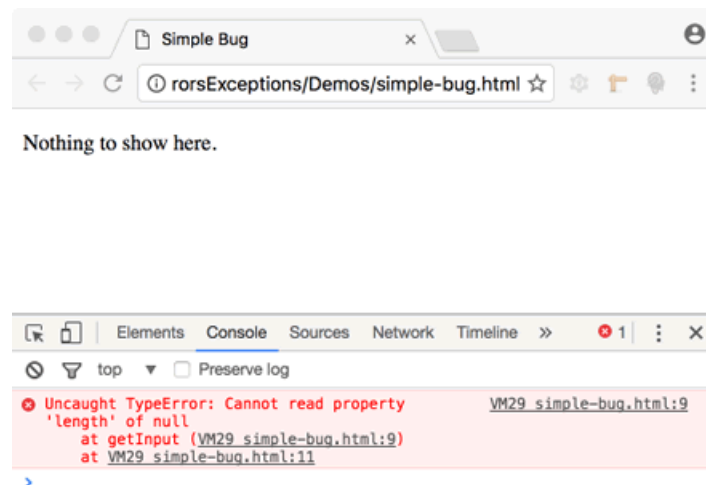
Demo 9.1: ErrorsExceptions/Demos/simple-bug.html

```
-----Lines 1 through 8 Omitted-----
9.  function getInput() {
10.    const name = prompt('Type your name', '');
11.    alert('Your name has ' + name.length + ' letters.');
```

```
12.  }
13.  getInput();
-----Lines 14 through 22 Omitted-----
```

It may not be obvious, but this code has a bug waiting to break free. If the user clicks **Cancel** or presses **Esc** the `prompt()` function will return `null`, which will cause the next line to fail with a null reference error.

If you as a programmer don't take any step to deal with this error, the user won't know what went wrong. The error message will most likely be hidden in the console:



EVALUATION COPY: Not to be used in class.



9.2. Globally Handled Errors

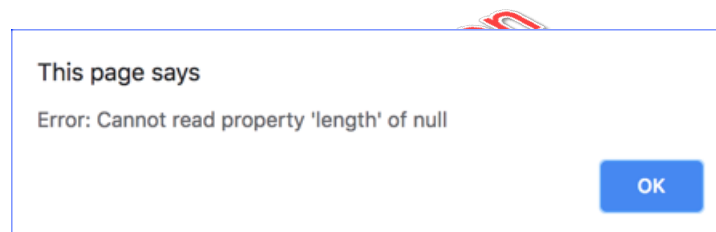
The window object has an event called `error` for which we can add an event handler, listening for global errors. The next demo shows an example of this:

Demo 9.2: ErrorsExceptions/Demos/simple-bug-onerror.html

```
-----Lines 1 through 8 Omitted-----
9.  window.addEventListener("error", function (e) {
10.    alert('Error: ' + e.error.message);
11.    return true;
12.  });
13.
14.  function getInput() {
15.    const name = prompt('Type your name', '');
16.    alert('Your name has ' + name.length + ' letters. ');
17.  }
18.  getInput();
-----Lines 19 through 27 Omitted-----
```

If the user presses **Esc** when the prompt asks for a name, our event handler fires.

Here's the alert that we show to the user:



This makes sure the user knows there was an error, but it doesn't help resolve the error. Also, it will let the user know about all errors that occur, even innocuous ones.

EVALUATION COPY: Not to be used in class.



9.3. Structured Error Handling

The best way to deal with errors is to detect them as close as possible to where they occur. This will increase the chance that we know what to do with the error. To that effect, JavaScript implements structured error handling, via the `try...catch...finally` block, also present in many other languages:

```

try {
    // statements;
} catch (error) {
    // statements;
} finally {
    // statements;
}

```

The idea is simple. If anything goes wrong in the statements that are inside the try block then the statements in the catch block will be executed. The finally block is optional and, if present, is always executed last, whether or not an error is caught.

The finally Block

In JavaScript, you're unlikely to need the finally block except for advanced code involving nested try / catch blocks.

Let's fix our example to catch that error:

Demo 9.3: ErrorsExceptions/Demos/simple-bug-try-catch.html

```

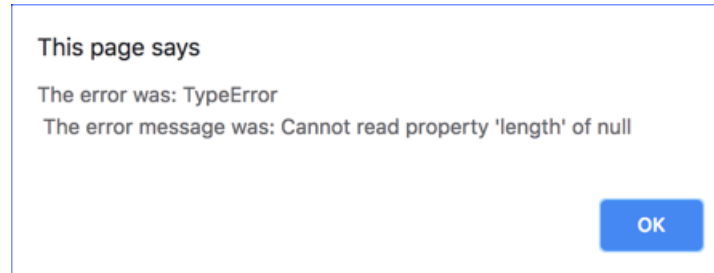
-----Lines 1 through 8 Omitted-----
9.   window.addEventListener("error", function (e) {
10.       alert('Error: ' + e.error.message);
11.       return true;
12.   });
13.
14.   function getInput() {
15.       try {
16.           const name = window.prompt('Type your name', '');
17.           alert('Your name has ' + name.length + ' letters. ');
18.       } catch (error) {
19.           alert('The error was: ' + error.name +
20.               '\n The error message was: ' + error.message);
21.       }
22.   }
23.   getInput();
-----Lines 24 through 32 Omitted-----

```

The error object in the catch block has two important properties: name and message.

- name - contains the type of error, which we could use to decide how we handle the error.
- message - contains the error message.

With that in place, if we reload the page and cancel out of the prompt, we will get the following alert:



It's a good programming practice to only handle the error on the spot if you are certain of what it is and if you actually have a way to take care of it (other than just suppressing it altogether.) To better target our error handling code, we will change it to only handle errors named "TypeError", which is the error name that we have identified for this bug.

Demo 9.4: ErrorsExceptions/Demos/simple-bug-try-catch-specific.html

```

-----Lines 1 through 8 Omitted-----
9.  window.addEventListener("error", function (e) {
10.      alert('Error: ' + e.error.message);
11.      return true;
12.  });
13.
14.  function getInput() {
15.      try {
16.          const name = window.prompt('Type your name', '');
17.          alert('Your name has ' + name.length + ' letters.');
```

```

18.      } catch (error) {
19.          if (error.name == 'TypeError') {
20.              alert('Please try again.');
```

```

21.              getInput();
22.          } else {
23.              throw error;
24.          }
25.      }
26.  }
27.  getInput();
-----Lines 28 through 36 Omitted-----
```

Now if a different type of error happens, that error will not be handled. The `throw` statement will forward the error as if we never had this `try...catch...finally` block. It is said that the error will *bubble up*.

❖ 9.3.1. Throwing Custom Errors

We can use the `throw` statement to throw our own errors. While there are several ways to do this, a simple way is to throw a new `Error` object:

```
throw new Error('The given color is not a valid color value.');
```



Exercise 22: Try/Catch

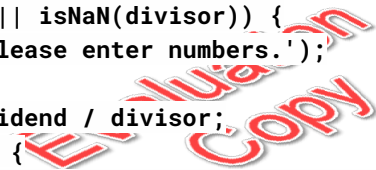
⌚ 10 to 15 minutes

In this exercise, you will handle potentially-problematic user input in a simple calculator, designed to return the quotient of two user-entered numbers.

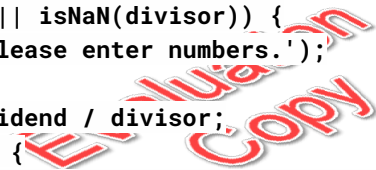
1. Open `ErrorException/Exercises/try-catch.html` for editing.
2. Within a `try` block:
 - A. Get the values of `dividendField` and `divisorField` and convert them to floats.
 - B. If the divisor is 0, throw a new `Error` with the message “Cannot divide by zero.”
 - C. If either the dividend or the divisor is not a number, throw a new `Error` with the message “Please enter numbers.”
 - D. Divide the dividend by the divisor and assign the result to a variable. If that result is not a number, throw a new `Error` with the message “Cannot solve this equation.”
 - E. Write the equation with the result out to the `innerHTML` of the `msgField` output. Note that this should only happen if none of the above checks resulted in errors.
3. Within the `catch` block, write “There was a problem: ” followed by the error message out to the `innerHTML` of the `msgField` output.
4. Test your solution in a browser.

Solution: ErrorsExceptions/Solutions/try-catch.html

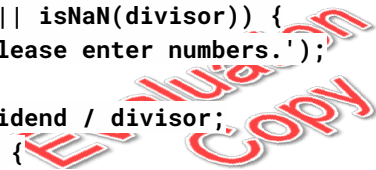
```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.  function displayAnswer() {
10.    const dividendField = document.getElementById('dividend');
11.    const divisorField = document.getElementById('divisor');
12.    const msgField = document.getElementById('msg');
13.
14.    try {
15.      const dividend = parseFloat(dividendField.value);
16.      const divisor = parseFloat(divisorField.value);
17.      if (divisor === 0) {
18.        throw new Error('Cannot divide by zero.');
```



```
19.      }
20.      if (isNaN(dividend) || isNaN(divisor)) {
21.        throw new Error('Please enter numbers.');
```



```
22.      }
23.      const quotient = dividend / divisor;
24.      if (isNaN(quotient)) {
25.        throw new Error('Cannot solve this equation.');
```



```
26.      }
27.
28.      msgField.innerHTML = String(dividend) + " / " +
29.        String(divisor) + " = " + String(quotient);
30.    }
31.    catch (e) {
32.      msgField.innerHTML = 'There was a problem: ' + e.message;
33.    }
34.  }
35.
36.  window.addEventListener('load', function() {
37.    const equals = document.getElementById('equals');
38.    equals.addEventListener('click', displayAnswer)
39.  });
40. </script>
41. <title>Try/Catch</title>
42. </head>
43. <body id="exercise">
44. <main>
```



```
45.     <input type="text" id="dividend" class="operator"> /
46.     <input type="text" id="divisor" class="operator">
47.     <button id="equals">=</button>
48.     <output id="msg"></output>
49. </main>
50. </body>
51. </html>
```

Evaluation
Copy

Conclusion

In this lesson, you have learned to use JavaScript's `try/catch/finally` to catch and handle errors.