

Programming with HTML, CSS, and JavaScript



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: 1.1.0

The Author

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 and Webucator at @webucator.

Class Files

Download the class files used in this manual at

<https://static.webucator.com/media/public/materials/classfiles/W8-20480-1.1.0.zip>.

Errata

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

Table of Contents

LESSON 1. A Quick Overview of Web Development.....	1
HTML is Part of a Team.....	1
Client-side Programming.....	4
Server-side Programming.....	6
LESSON 2. Introduction to HTML.....	9
📄 Exercise 1: A Simple HTML Document	10
HTML Elements, Attributes, and Comments.....	13
The HTML Skeleton.....	16
Viewing the Page Source.....	20
Special Characters.....	21
History of HTML.....	22
The lang Attribute.....	23
LESSON 3. Paragraphs, Headings, and Text.....	25
Paragraphs.....	25
Heading Levels.....	27
Breaks and Horizontal Rules.....	28
The div Element.....	29
📄 Exercise 2: Creating an HTML Page	31
Quoted Text.....	34
Preformatted Text.....	36
Inline Semantic Elements.....	37
📄 Exercise 3: Adding Inline Elements	42
LESSON 4. HTML Links.....	45
Text Links.....	45
Absolute vs. Relative Paths.....	46
Targeting New Tabs.....	49
Email Links.....	52
📄 Exercise 4: Adding Links	53
Lorem Ipsum.....	62
The title Attribute.....	63
Targeting a Specific Location on the Page.....	63
LESSON 5. HTML Images.....	67
Inserting Images.....	67
Image Links.....	70
📄 Exercise 5: Adding Images to the Page	72
Providing Alternative Images.....	76

LESSON 6. HTML Lists.....	79
Unordered Lists.....	79
Ordered Lists.....	82
Definition Lists.....	90
📄 Exercise 6: Creating Lists.....	93
LESSON 7. Sectioning a Web Page.....	101
Semantic Block-Level Elements.....	101
Articles vs. Sections.....	102
Sectioning the Home Page.....	104
Sectioning Content and Styling.....	110
Heading Levels and Sectioning Elements.....	111
📄 Exercise 7: Sectioning the Pages.....	112
LESSON 8. Crash Course in CSS.....	113
Benefits of Cascading Style Sheets.....	114
CSS Rules.....	114
Selectors.....	115
Combinators.....	119
Precedence of Selectors.....	121
How Browsers Style Pages.....	122
CSS Resets.....	125
CSS Normalizers.....	127
External Stylesheets, Embedded Stylesheets, and Inline Styles.....	128
📄 Exercise 8: Creating an External Stylesheet.....	133
📄 Exercise 9: Creating an Embedded Stylesheet.....	141
📄 Exercise 10: Adding Inline Styles.....	144
<div> and	146
📄 Exercise 11: Styling div and span.....	150
Media Types.....	152
Units of Measurement.....	154
Inheritance.....	159
LESSON 9. CSS Fonts.....	165
font-family.....	165
@font-face.....	169
font-size.....	176
font-style.....	181
font-variant.....	181
font-weight.....	182
line-height.....	185
font.....	190
📄 Exercise 12: Styling Fonts.....	194

LESSON 10. Color and Opacity.....	197
About Color and Opacity.....	197
Color and Opacity Values.....	197
color.....	200
opacity.....	203
📄 Exercise 13: Adding Color and Opacity to Text.....	206
LESSON 11. CSS Text.....	211
letter-spacing.....	211
text-align.....	213
text-decoration.....	215
text-indent.....	217
text-shadow.....	219
text-transform.....	221
white-space.....	223
word-break.....	226
word-spacing.....	228
📄 Exercise 14: Text Properties.....	231
LESSON 12. JavaScript Basics.....	237
JavaScript vs. EcmaScript.....	237
The HTML DOM.....	238
JavaScript Syntax.....	239
Accessing Elements.....	240
Where Is JavaScript Code Written?.....	241
JavaScript Objects, Methods and Properties.....	244
📄 Exercise 15: Alerts, Writing, and Changing Background Color.....	247

LESSON 13. Variables, Arrays, and Operators.....	251
JavaScript Variables.....	251
A Loosely Typed Language.....	252
Google Chrome DevTools.....	253
Storing User-Entered Data.....	257
📄 Exercise 16: Using Variables.....	261
Constants.....	262
Arrays.....	263
📄 Exercise 17: Working with Arrays.....	267
Associative Arrays.....	270
Playing with Array Methods.....	273
JavaScript Operators.....	274
The Modulus Operator.....	277
Playing with Operators.....	277
The Default Operator.....	280
📄 Exercise 18: Working with Operators.....	283
LESSON 14. JavaScript Functions.....	291
Global Objects and Functions.....	291
📄 Exercise 19: Working with Global Functions.....	294
User-defined Functions.....	299
📄 Exercise 20: Writing a JavaScript Function.....	303
Returning Values from Functions.....	307
LESSON 15. Built-In JavaScript Objects.....	309
String.....	309
Math.....	314
Date.....	317
Helper Functions.....	322
📄 Exercise 21: Returning the Day of the Week as a String.....	323

LESSON 16. Conditionals and Loops.....	327
Conditionals.....	327
Short-circuiting	331
Switch / Case.....	335
Ternary Operator.....	341
Truthy and Falsy.....	342
📄 Exercise 22: Conditional Processing.....	343
Loops.....	345
while and do...while Loops.....	346
for Loops.....	348
break and continue.....	350
📄 Exercise 23: Working with Loops.....	352
Array: forEach().....	355
LESSON 17. Event Handlers and Listeners.....	357
On-event Handlers.....	357
📄 Exercise 24: Using On-event Handlers.....	360
The addEventListener() Method.....	363
Anonymous Functions.....	370
Capturing Key Events.....	372
📄 Exercise 25: Adding Event Listeners.....	374
Benefits of Event Listeners.....	377
Timers.....	379
📄 Exercise 26: Typing Test.....	383

LESSON 18. The HTML Document Object Model.....	389
CSS Selectors.....	390
The innerHTML Property.....	394
Nodes, NodeLists, and HTMLCollections.....	395
Accessing Element Nodes.....	395
📄 Exercise 27: Accessing Elements.....	406
Dot Notation and Square Bracket Notation.....	408
Accessing Elements Hierarchically.....	412
📄 Exercise 28: Working with Hierarchical Elements.....	416
Accessing Attributes.....	420
Creating New Nodes.....	422
Focusing on a Field.....	423
Shopping List Application.....	424
📄 Exercise 29: Logging.....	426
📄 Exercise 30: Adding EventListeners.....	429
📄 Exercise 31: Adding Items to the List.....	433
📄 Exercise 32: Dynamically Adding Remove Buttons to the List Items.....	435
📄 Exercise 33: Removing List Items.....	437
📄 Exercise 34: Preventing Duplicates and Zero-length Product Names.....	439
Manipulating Tables.....	440
LESSON 19. HTML Forms.....	449
How HTML Forms Work.....	449
The form Element.....	450
Form Elements.....	451
Buttons.....	461
📄 Exercise 35: Creating a Registration Form.....	464
Checkboxes.....	467
Radio Buttons.....	468
📄 Exercise 36: Adding Checkboxes and Radio Buttons.....	470
Fieldsets.....	472
Select Menus.....	474
Textareas.....	476
📄 Exercise 37: Adding a Select Menu and a Textarea.....	478
HTML Forms and CSS.....	480

LESSON 20. JavaScript Form Validation.....	483
Server-side Form Validation.....	483
HTML Form Validation.....	484
Accessing Form Data.....	489
Form Validation with JavaScript.....	491
📄 Exercise 38: Checking the Validity of the Email and URL Fields.....	493
Checking Validity on Input and Submit Events.....	494
Adding Error Messages.....	500
Validating Textareas.....	506
Validating Checkboxes.....	508
Validating Radio Buttons.....	510
Validating Select Menus.....	513
📄 Exercise 39: Validating the Ice Cream Order Form.....	516
Giving the User a Chance.....	523
LESSON 21. Regular Expressions.....	529
Getting Started.....	529
Regular Expression Syntax.....	530
Backreferences.....	541
Form Validation with Regular Expressions.....	543
Cleaning Up Form Entries.....	544
📄 Exercise 40: Cleaning Up Form Entries.....	549
A Slightly More Complex Example.....	551

LESSON 1

A Quick Overview of Web Development

Topics Covered

- ☒ Client-side web development languages.
- ☒ Server-side web development languages.

Introduction

Learning HTML is the first step to becoming a web developer. But it is just one of several languages you will need to know to create websites and web applications. In this lesson, you will learn where HTML fits in the ecosystem of web development.



1.1. HTML is Part of a Team

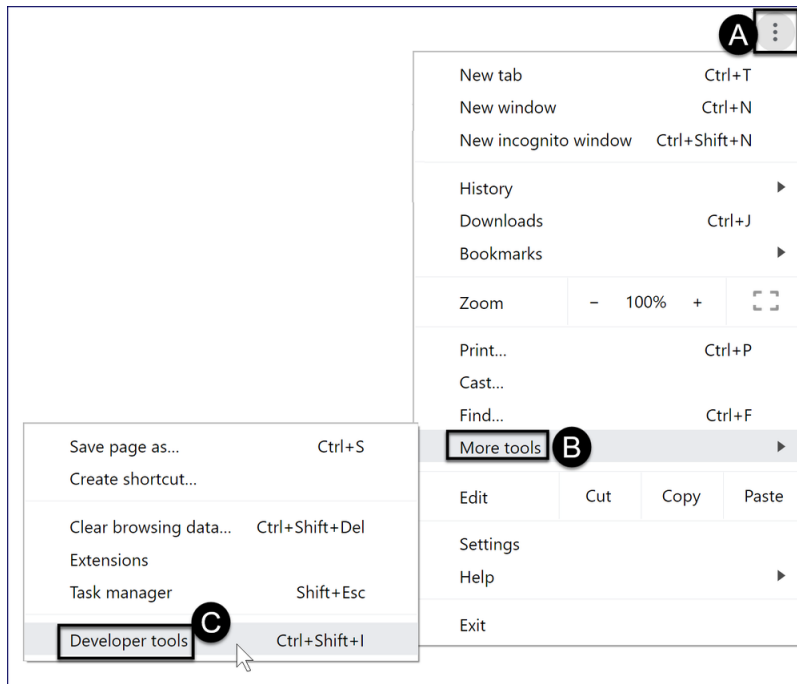
Before you get started writing HTML code, it's important to understand where HTML fits in the ecosystem of web development. Let's first consider what happens when you visit a website. When you type in a URL in the location bar of your browser (e.g., `https://www.runners-home.com`), the browser makes a request from the web server for a web page. If you don't specify the name of the file you want (e.g., `contact.html` or `about.html`), the web server will send a default page, which is most likely called `index.html`, `index.php`, `index.cfm`, or something similar. The web server returns that web page to the browser for display. The web page may include references to other files:

- Images to display on the page.
- Style sheets to add formatting to the page.
- Scripts to add interactivity to the page.

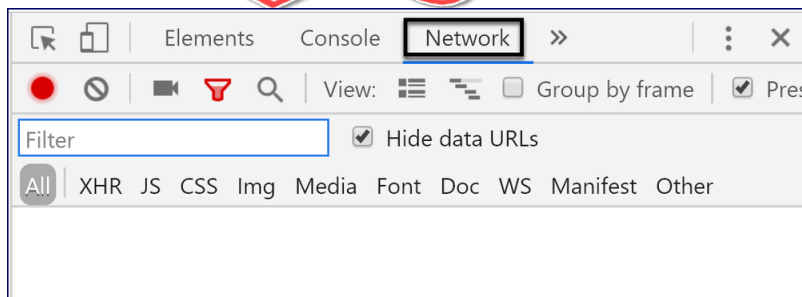
The browser will download these referenced files as well. To get a better feel for this, do the following in Google Chrome:

1. As illustrated below...
 - A. Click the three-vertical-dot icon in the upper right of Google Chrome:
 - B. Then select **More tools**.

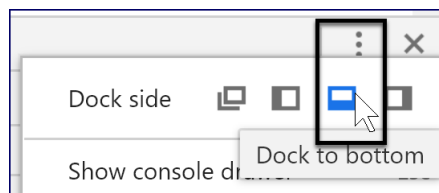
C. Then select **Developer tools**. This will open **Chrome's Developer tools**.



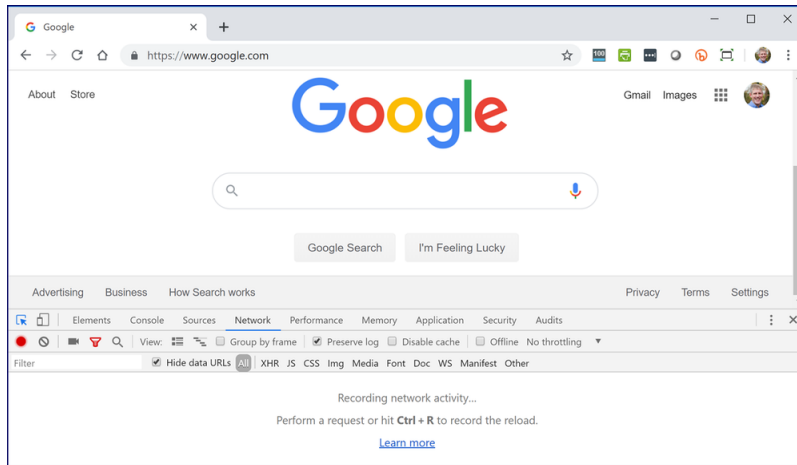
2. In **Developer tools**, select the **Network** tab:



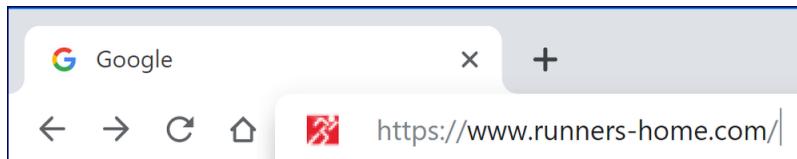
3. If **Developer tools** isn't docked on the bottom of the browser, move it to the bottom to make it easier to see the **Network** tab information:



Developer tools should now be at the bottom of the browser:



- Now, in the **location bar** of your browser, enter `https://www.runners-home.com` and press **Enter**:



- After the page loads, take a look at the **Network** tab. You should see something like this:

 A screenshot of the Chrome DevTools Network tab. The 'Network' tab is selected, and the 'All' filter is active. The table below shows a list of network requests. The first request, 'www.runners-home.com', is highlighted. The table has columns for Name, Status, Type, and Initiator.

Name	Status	Type	Initiator
www.runners-home.com	200	document	Other
toggle.js	200	script	(index)
normalize.css	200	stylesheet	(index)
all.css	200	stylesheet	(index)
styles.css	200	stylesheet	(index)
home-styles.css	200	stylesheet	(index)
runners-home.png	200	png	(index)
runners.jpg	200	jpeg	(index)
tips-running.png	200	png	(index)
tips-health.png	200	png	(index)
fa-brands-400.woff2	200	font	(index)
fa-solid-900.woff2	200	font	(index)

- A. `www.runners-home.com` – Although it doesn't specify the page, this represents the main page you requested: the HTML page (usually called `index`).
- B. `toggle.js` – A JavaScript page used for adding interactivity to the HTML page. Notice in the fourth column that the initiator for this page is (`index`). That means that the HTML code instructed the browser to download this page.
- C. Style sheets ending with `.css` – CSS pages used for adding style and formatting to the HTML page. Like the JavaScript page, these pages were also requested by the HTML code.
- D. Images ending with `.png` and `.jpg` – Images to display on the page. The images were also requested by the HTML code.
- E. Font files ending with `.woff2` – Web fonts for adding custom fonts to your web page.
- F. Though not shown in the screenshot above, you may also see `favicon.ico` – An icon used to identify the website on the browser tab:



Note that `favicon.ico` only gets delivered when pages are sent from a web server. When you open files in a browser directly from your file system, the favicon will not show up.

As you can see, HTML, while essential, is just a piece of the puzzle. Web development involves a combination of client-side-programming and server-side-programming languages. We will now introduce the most common languages, but don't worry if you don't fully understand the role of each one. At this point, the most important takeaway is that HTML is just one of many languages used in web development.



1.2. Client-side Programming

Client-side programming involves writing code that is interpreted by a browser, such as Google Chrome or Safari, whether it be on your desktop or mobile device. The most common languages and technologies used in client-side programming are HTML, Cascading Style Sheets (CSS), and JavaScript.

❖ 1.2.1. HTML

Hypertext Markup Language (HTML) is the language behind most web pages. The language is made up of elements that describe the structure of the content on a web page.

❖ 1.2.2. Cascading Style Sheets

Cascading Style Sheets (CSS) are used in HTML pages to format and lay out the content. CSS rules defining color, size, positioning, and other display aspects of elements are mixed within the HTML code or in linked external style sheets.

❖ 1.2.3. JavaScript¹

JavaScript is used to make HTML pages more dynamic and interactive. It can be used to validate forms, pop up new windows, create audio and video controls, and create dynamic effects such as drop-down menus and modal dialogs.

❖ 1.2.4. Ajax

The term *Ajax* was originally a pseudo-acronym for “Asynchronous JavaScript and XML,” but is now used much more broadly to cover all methods of communicating with a server using JavaScript.

The main purpose of Ajax is to provide a simple and standard means for a web page to communicate with the server without a complete page refresh.

❖ 1.2.5. JavaScript Frameworks

JavaScript frameworks are frameworks written in JavaScript that create a different approach to web application design. Popular frameworks include Angular (<https://angular.io>), React (<https://reactjs.org>), Vue.js (<https://vuejs.org>), and jQuery (<https://jquery.com>). You should learn JavaScript before beginning to work with a JavaScript framework.

❖ 1.2.6. CSS Frameworks

CSS frameworks are frameworks that allow you to quickly design HTML pages with a predefined set of CSS classes. The most popular CSS framework is Bootstrap (<https://getbootstrap.com/>), which comes with a library of stylish components that you can easily incorporate into your website.

1. The word “JavaScript” is a trademark of Oracle. Microsoft’s version of this language is called JScript.

Tailwind CSS (<https://tailwindcss.com/>) is a newer CSS framework that allows for more customization, but requires a better understanding of CSS.



1.3. Server-side Programming

Server-side programming involves writing code that connects web pages with databases, XML pages, email servers, file systems, and other systems and software accessible from the web server. The most common server-side languages and programming frameworks are PHP, Java Enterprise Edition, ASP.NET, ColdFusion, Node.js, and Python.

❖ 1.3.1. PHP

PHP (<https://www.php.net>) is open source. It is the language behind WordPress and has long been a popular alternative to proprietary languages such as ColdFusion and ASP.NET. PHP is lightweight and relatively simple to learn.

❖ 1.3.2. Java EE

Java EE (<https://docs.oracle.com/javaee>) is used in large web projects. With its power and robustness comes a steep learning curve.

❖ 1.3.3. ASP.NET

Microsoft's ASP.NET (<https://docs.microsoft.com/aspnet>) is not a language, but a framework for writing websites and software. ASP.NET pages can be written in many languages, but the most popular are C# (pronounced C-sharp) and Visual Basic .NET (VB.NET).

❖ 1.3.4. ColdFusion

ColdFusion (<https://coldfusion.adobe.com>), created by Allaire (now owned by Adobe), is arguably the simplest of all server-side languages. It is tag-based, which makes it look a lot like HTML and easier for client-side programmers to understand than some of the other choices.

❖ 1.3.5. Node.js

Node.js (<https://nodejs.org>) is a JavaScript runtime that can run on the server, allowing developers to use JavaScript for server-side scripting as well as client-side scripting.

❖ 1.3.6. Python

Python (<https://www.python.org>) has been a popular open-source programming language for a long time. There are many web frameworks based on Python, the most popular of which is Django (<https://www.djangoproject.com>).

Conclusion

This lesson has provided a general overview of the different languages and frameworks commonly used in web development. Again, don't worry if you don't remember all the different technologies and their specific roles. Take it one step at a time. The first step is to learn HTML.

LESSON 2

Introduction to HTML

Topics Covered

- ☑ Creating a simple HTML page.
- ☑ Elements and attributes.
- ☑ The skeleton of an HTML document.
- ☑ Whitespace.
- ☑ Special characters.
- ☑ History of HTML.


Evaluation
Copy

Introduction

You're likely learning HTML because you have dreams of creating a website; perhaps like one you have seen or perhaps like no other that's yet been created. You have the dream, now it's time to start building the foundation.

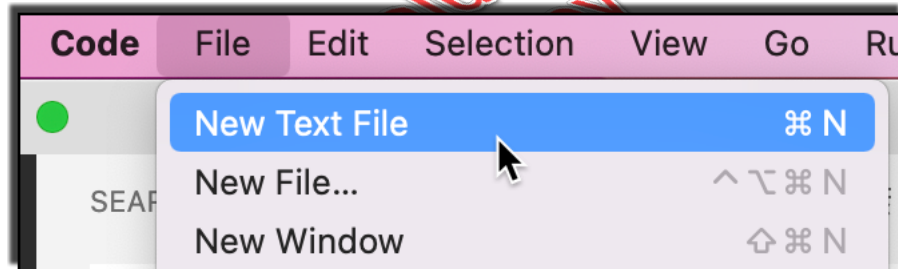
We will begin with a simple exercise.

Exercise 1: A Simple HTML Document

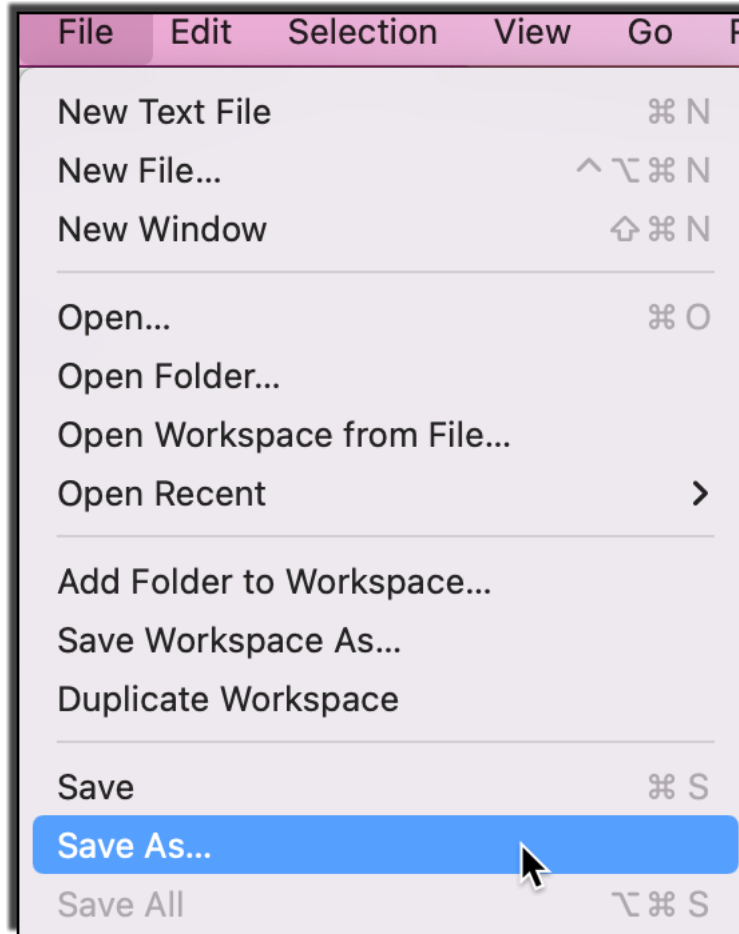
 5 to 15 minutes

In this exercise, you will create your first HTML document by simply copying some code. The purpose is to give you some sense of the structure of an HTML document.

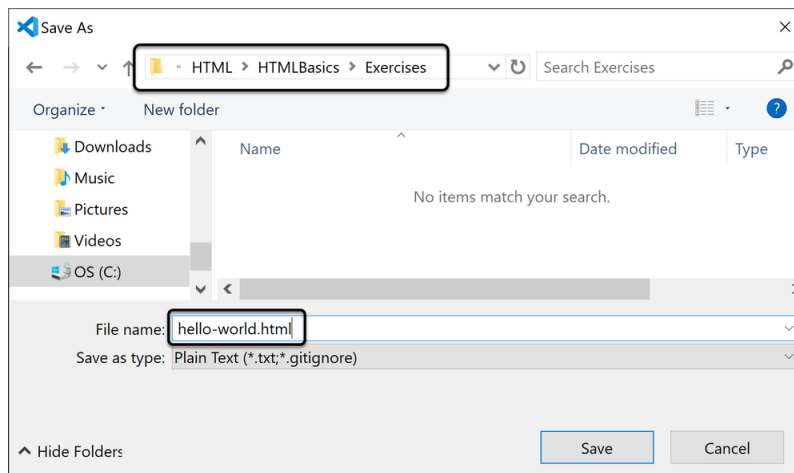
1. Create a new file in Visual Studio Code by selecting **New Text File** from the **File** menu:



2. To save the file, select **File > Save As...**



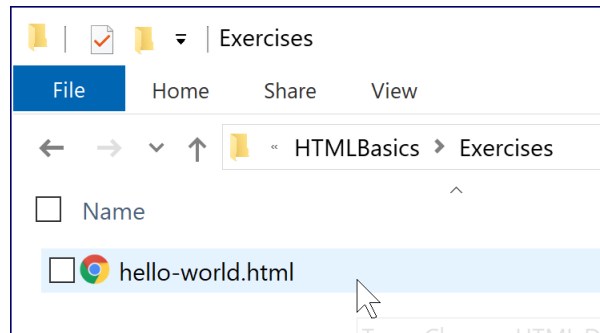
3. Save the file as `hello-world.html` in the `HTMLBasics/Exercises` folder:



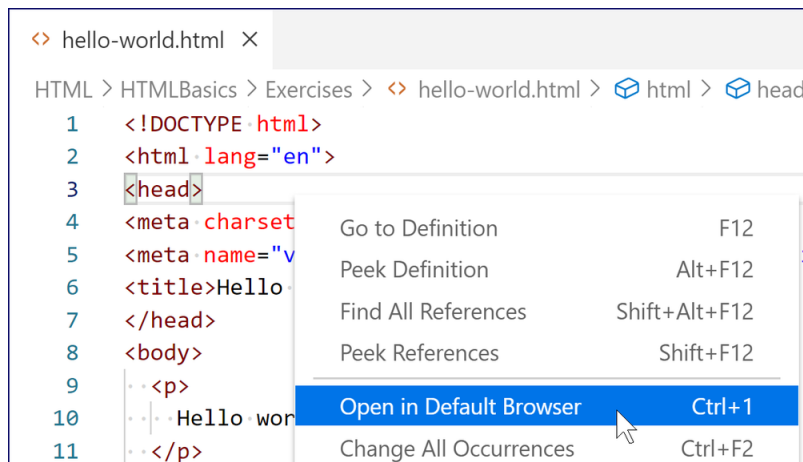
4. Type the following exactly as shown:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Hello world!</title>
</head>
<body>
  <p>Hello world!</p>
</body>
</html>
```

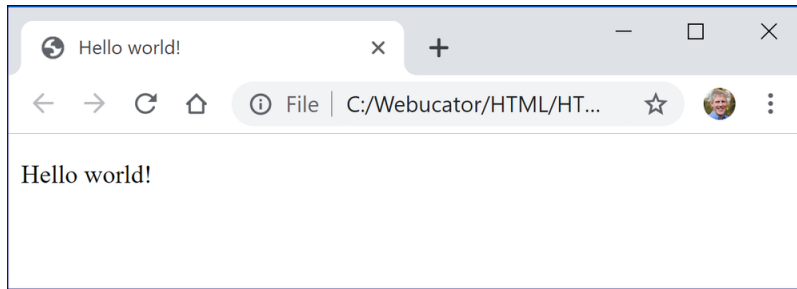
5. Save the file again and then open it in your browser either by navigating to the file in your folder system and double-clicking it:



Or by right-clicking the file in Visual Studio Code and selecting **Open in Default Browser**, which you should have added when setting up Visual Studio Code:



The page should appear as follows:



Solution: HTMLBasics/Solutions/hello-world.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. <title>Hello world!</title>
7. </head>
8. <body>
9.   <p>Hello world!</p>
10. </body>
11. </html>
```

Evaluation
Copy

2.1. HTML Elements, Attributes, and Comments

HTML *elements* describe the structure and content of a web page. *Tags* are used to indicate the beginning and end of elements. The syntax is as follows:

```
<tagname>Element content</tagname>
```

❖ 2.1.1. Attributes

Tags often have *attributes* for further defining the element. Attributes usually come in name-value pairs.

Note that attributes only appear in the opening tag, like so:

```
<tagname att1="value" att2="value">Element content</tagname>
```

There are some attributes that do not need to take a value. You can think of them as being “on” when the attribute is present and “off” when it is not. For example:

```
<tagname att>Element content</tagname>
```

The order of attributes is not important.

❖ 2.1.2. Empty vs. Container Tags

The tags shown above are called *container* tags because they have both an opening and closing tag with content contained between them. Tags that do not contain content are called *empty* tags. The syntax is as follows:

```
<tagname>
```

or

```
<tagname att1="value" att2="value">
```

Shortcut Close

Empty tags may also be written as follows:

```
<tagname />
```

or

```
<tagname att1="value" att2="value" />
```

The forward slash (/) at the end, just before the close angle bracket (>), explicitly indicates that this tag is closed. In general, it is not necessary to use this shortcut close, but it also doesn’t cause any harm. Our only recommendation is that if you use it, use it consistently.

❖ 2.1.3. Blocks and Inline Elements

Block-level Elements

Block-level elements are elements that separate a block of content. For example, a paragraph (`<p>`) element is a block-level element. Other block-level elements include:

1. Lists (`` and ``)
2. Tables (`<table>`)
3. Forms (`<form>`)
4. Divs (`<div>`)

Inline Elements

Inline elements are elements that affect only snippets of content and do not block off a section of a page. Examples of inline elements include:

1. Links (`<a>`)
2. Images (``)
3. Form elements (`<input>`, `<button>`, `<select>`, `<textarea>`, etc.)
4. Phrase elements (``, ``, `<code>`, etc.)
5. Spans (``) – wraps text without giving it any special meaning. Meaning and style can be applied through its attributes.

You will learn what most of these elements do in upcoming lessons.

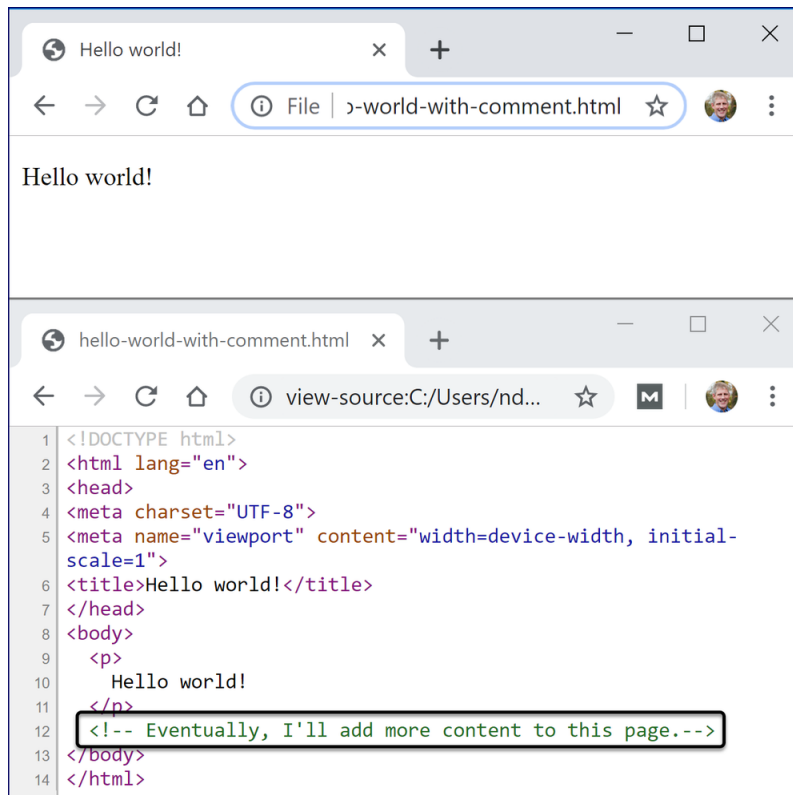
❖ 2.1.4. Comments

HTML comments are enclosed in `<!--` and `-->`.

For example:

```
<!-- This is an HTML comment -->
```

Commented content will not show up on the web page. It is meant for developers' eyes only. However, as illustrated in the following screenshot, users can see comments by viewing the source of the page.² So don't put anything in your comments that you wouldn't want your site visitors to read.



Comments are generally used for one of two purposes:

1. To write helpful notes about the code; for example, why something is written in a specific way.
2. To comment out some code that is not currently needed, but may be used sometime in the future.



2.2. The HTML Skeleton

At its simplest, an HTML page contains what can be thought of as a skeleton: the main structure of the page. It looks like this:

-
2. You'll learn how to view the source of a page shortly.

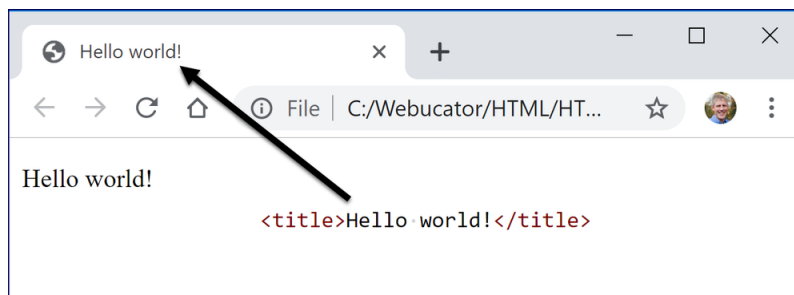
Demo 2.1: HTMLBasics/Demos/skeleton.html

```
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. <title></title>
7. </head>
8. <body>
9.   <!--
10.     Content that appears on the page goes in the body
11.     (but I won't show up because I'm in a comment).
12.   -->
13. </body>
14. </html>
```

❖ 2.2.1. The head Element

The head element (<head>) contains content that is not displayed on the page itself. Some of the elements commonly found in the head are:

1. Title of the page (<title>). Browsers typically show the title in the “title bar” at the top of the browser window:



2. Meta tags (<meta>), which contain descriptive information about the page.
3. Script blocks (<script>), which contain JavaScript code for adding functionality and interactivity to a page.
4. Style blocks (<style>), which contain Cascading Style Sheet rules for formatting a page.
5. References or links to external style sheets (<link>).

Here is an example head element:

```
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link href="styles.css" rel="stylesheet">
<script src="script.js"></script>
<title>Dummy title</title>
</head>
```

Don't worry about the `<link>` and `<script>` tags. You'll learn about those when you learn CSS and JavaScript. For now, it is enough to know that HTML pages can reference CSS and JavaScript pages.

❖ 2.2.2. The body Element

The body element (`<body>`) contains all of the content that appears on the page itself. Tags that can be placed within the `<body>` tag will be covered thoroughly throughout these lessons.

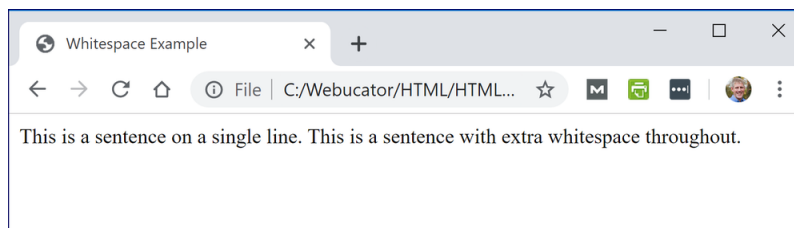
❖ 2.2.3. Whitespace

Extra whitespace is ignored in HTML. This means that all hard returns, tabs, and multiple spaces are condensed into a single space for display purposes. Review the following demo:

Demo 2.2: HTMLBasics/Demos/whitespace.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. <title>Whitespace Example</title>
7. </head>
8. <body>
9. This is a sentence on a single line.
10.
11.     This
12.     is
13.     a
14.         sentence with
15.         extra whitespace
16.     throughout.
17.
18. </body>
19. </html>
```

Open HTMLBasics/Demos/whitespace.html in your browser. You will see that the two sentences in the code above will be rendered in exactly the same way. Notice that all extra whitespace is ignored:



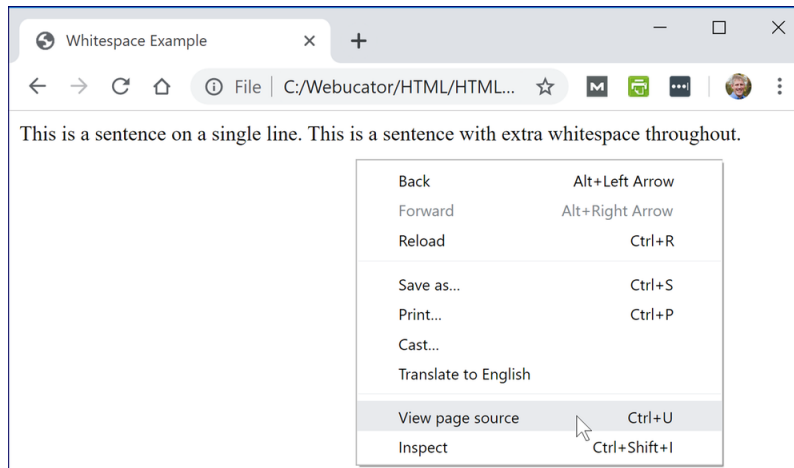
Why is extra whitespace ignored?

Browsers ignore extra whitespace so that web developers can use hard returns, spaces, and tabs to make their code readable. For example, we like to limit the length of one line of HTML code to 120 characters as this makes it easier to read the code. But we don't want text that we send to the browser to also be limited to 120-character lines.

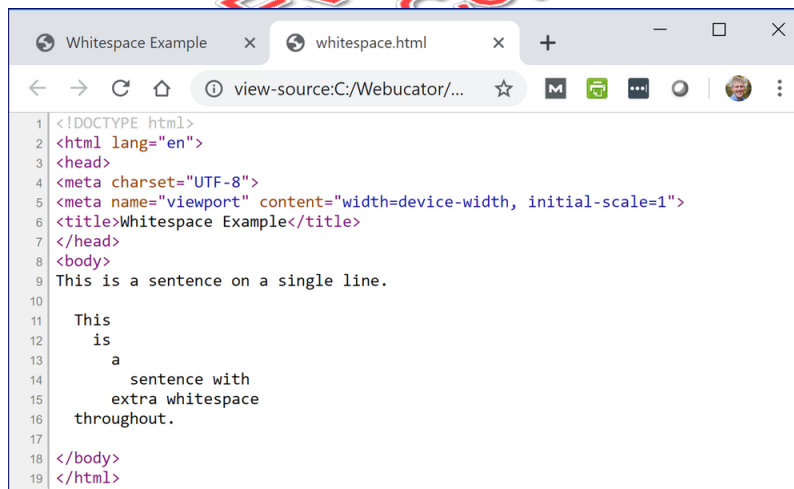


2.3. Viewing the Page Source

Most browsers will let you view the source of an HTML page. This is a useful way to see what the browser sees. In Google Chrome, you can do this by right-clicking the page and selecting **View Page Source**:



You will see the source of the page you created:



This demonstrates that Chrome does see all the whitespace in the page source. But it condenses it when it presents the web page.



2.4. Special Characters

Special characters (e.g., characters that do not show up on your keyboard) can be added to HTML pages using entity names and numbers. For example, a copyright symbol (©) can be added using `©` or `©`. The following table shows some of the more common character references:³

HTML Entities

DESCRIPTION	NUMBER	NAME	SYMBOL
Quotation mark	<code>&#34;</code>	<code>&quot;</code>	"
Single quotation mark (apostrophe)	<code>&#39;</code>	<code>&apos;</code>	'
Ampersand	<code>&#38;</code>	<code>&amp;</code>	&
Less than	<code>&#60;</code>	<code>&lt;</code>	<
Greater than	<code>&#62;</code>	<code>&gt;</code>	>
Non-breaking space	<code>&#160;</code>	<code>&nbsp;</code>	
Cent sign	<code>&#162;</code>	<code>&cent;</code>	¢
Pound sign	<code>&#163;</code>	<code>&pound;</code>	£
Yen sign	<code>&#165;</code>	<code>&yen;</code>	¥
Euro sign	<code>&#8364;</code>	<code>&euro;</code>	€
Copyright	<code>&#169;</code>	<code>&copy;</code>	©
Registered trademark	<code>&#174;</code>	<code>&reg;</code>	®
Trademark	<code>&#8482;</code>	<code>&trade;</code>	™
Inverted question mark	<code>&#191;</code>	<code>&iquest;</code>	¿
Inverted exclamation mark	<code>&#161;</code>	<code>&iexcl;</code>	¡
Fraction: one-fourth	<code>&#188;</code>	<code>&frac14;</code>	¼
Fraction: one-half	<code>&#189;</code>	<code>&frac12;</code>	½
Fraction: three-fourths	<code>&#190;</code>	<code>&frac34;</code>	¾
En dash	<code>&#8211;</code>	<code>&ndash;</code>	–
Em dash	<code>&#8212;</code>	<code>&mdash;</code>	—
Dagger	<code>&#8224;</code>	<code>&dagger;</code>	†
Horizontal ellipsis	<code>&#8230;</code>	<code>&hellip;</code>	...



3. See <https://html.spec.whatwg.org/multipage/named-characters.html#named-character-references> for the official list of HTML entities.

2.5. History of HTML

HTML has a long history and several versions:

1. HTML was invented in the early 1990s.
2. In 1996, the World Wide Web Consortium (W3C)⁴ began maintaining the HTML specification. At that point, HTML was already on version 2.0.
3. HTML 3.2 and HTML 4.0 were both released in 1997.
4. XHTML, a separate XML version of HTML, was released in 2000.
5. HTML5 was released in 2014 and updated to HTML 5.1 (now with a space before the 5) in 2016. As of this writing, it is in version 5.3.
6. For a while, two separate groups, the W3C and WHATWG⁵ managed separate HTML specifications, with at least a little tension between the two groups⁶. In 2019, the W3C gave full control of the HTML standard to WHATWG.

❖ 2.5.1. HTML5 / HTML 5

You may hear a lot about HTML5 or HTML 5. For a few years, the distinction between HTML 4 and HTML 5 was important. Today, you can simply think of everything as just HTML. The HTML you use will be determined more by what modern browsers support than by what the specifications specify.

❖ 2.5.2. What This Means for You

As a web developer, you don't need to be too concerned with this history. The question is: **what can you do today?** The best, most developer-friendly online reference is kept by Mozilla at <https://developer.mozilla.org/en-US/docs/Web/HTML/Reference>.

Because people have been writing HTML for a long time, web pages exist that use deprecated (phased out) tags and outdated techniques, such as using `color` and `bgcolor` attributes instead of CSS to add color to pages. Modern browsers tend to be backward compatible, but you should avoid using any deprecated tags and attributes.

4. <https://www.w3.org/standards/techs/html>

5. <https://html.spec.whatwg.org>

6. https://en.wikipedia.org/wiki/HTML5#W3C_and_WWHATWG_conflict

To indicate that you are using the latest version of HTML, you should use the following DOCTYPE at the beginning of every HTML document:

```
<!DOCTYPE html>
```

This DOCTYPE is completely backward compatible and will make all browsers work in “standards mode,” which is almost definitely what you want.

Although they are not required, you should generally use the following <meta> tags:

```
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
```

The first specifies the character set and the second makes web pages adjust for different screen sizes.

The opening of an HTML page should look like this (assuming your page is in English):

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Evaluation
Copy



2.6. The lang Attribute

The lang attribute is used to tell the browser and other user agents⁷ the language contained within an element. While it is not required, the W3C recommends that lang be included in the <html> tag of all HTML documents, like so:

```
<html lang="en">
```

According to the W3C,⁸ the lang attribute is helpful in:

7. A user agent is software that acts on behalf of the user. When used in web development, the term refers to a web browser or any other software used to request a web page from the server.
8. <https://www.w3.org/International/questions/qa-html-language-declarations>

1. Assisting search engines.
2. Assisting speech synthesizers.
3. Helping a user agent select glyph variants for high-quality typography.
4. Helping a user agent choose a set of quotation marks.
5. Helping a user agent make decisions about hyphenation, ligatures, and spacing.
6. Assisting spell checkers and grammar checkers.

If a portion of the page is written in a different language, you can wrap that portion in a tag that includes the `lang` attribute, like this:

```
<span lang="fr">Bonjour, mon ami!</span>
```

A smart screen reader⁹ could use that information to properly pronounce the French.

Conclusion

In this lesson, you have learned the basics of HTML. You should understand how an HTML page is structured and understand the basic syntax of HTML tags. In addition, you have learned some of the history of HTML.

9. Screen readers make it possible for visually impaired people to read web pages and other computer-based content. A widely used screen reader is JAWS from Freedom Scientific (<https://www.freedomscientific.com>).

LESSON 3

Paragraphs, Headings, and Text

Topics Covered

- ☒ Paragraphs.
- ☒ Headings.
- ☒ Breaks and horizontal rules.
- ☒ Quoted text.
- ☒ Preformatted text.
- ☒ Phrase elements.

Evaluation
Copy

Introduction

This lesson discusses how to properly mark up text. With just a few exceptions, it does not discuss how to change the formatting or display of these elements. That is a task for CSS.



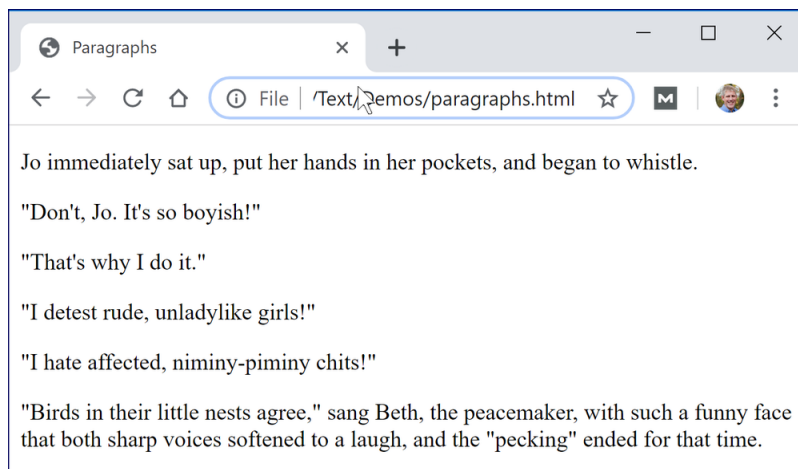
3.1. Paragraphs

Paragraph text should be contained in `<p>` tags as shown in the following example:

Demo 3.1: Text/Demos/paragraphs.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. <title>Paragraphs</title>
7. </head>
8. <body>
9.   <!-- From Little Women by Louisa May Alcott-->
10.   <p>Jo immediately sat up, put her hands in her pockets,
11.     and began to whistle.</p>
12.   <p>"Don't, Jo. It's so boyish!"</p>
13.   <p>"That's why I do it."</p>
14.   <p>"I detest rude, unladylike girls!"</p>
15.   <p>"I hate affected, niminy-piminy chits!"</p>
16.   <p>"Birds in their little nests agree," sang Beth, the peacemaker,
17.     with such a funny face that both sharp voices softened to a laugh,
18.     and the "pecking" ended for that time.</p>
19. </body>
20. </html>
```

This page will be rendered as follows:



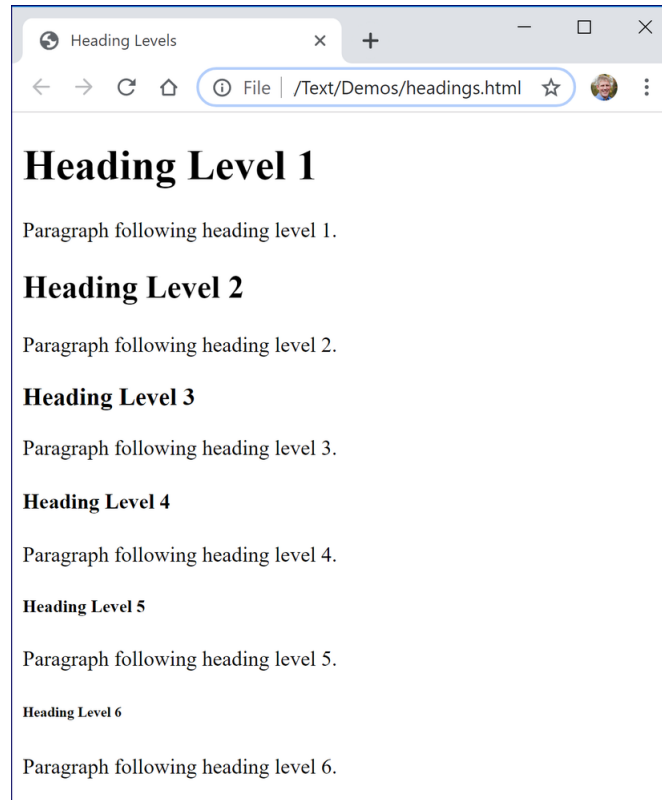
3.2. Heading Levels

HTML supports six levels of heading. The tags are `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, and `<h6>`, descending in importance from `<h1>` to `<h6>`. Headings are block-level elements. Examine the following code:

Demo 3.2: Text/Demos/headings.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.  <title>Heading Levels</title>
7.  </head>
8.  <body>
9.    <h1>Heading Level 1</h1>
10.   <p>Paragraph following heading level 1.</p>
11.   <h2>Heading Level 2</h2>
12.   <p>Paragraph following heading level 2.</p>
13.   <h3>Heading Level 3</h3>
14.   <p>Paragraph following heading level 3.</p>
15.   <h4>Heading Level 4</h4>
16.   <p>Paragraph following heading level 4.</p>
17.   <h5>Heading Level 5</h5>
18.   <p>Paragraph following heading level 5.</p>
19.   <h6>Heading Level 6</h6>
20.   <p>Paragraph following heading level 6.</p>
21. </body>
22. </html>
```

The following screenshot shows how they are formatted by default:



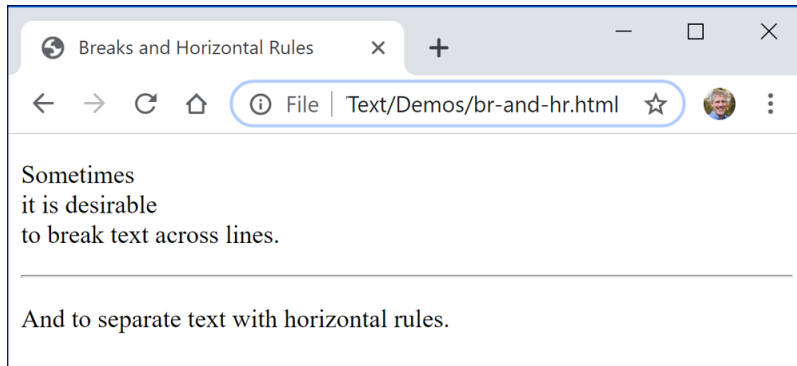
3.3. Breaks and Horizontal Rules

The `
` tag forces a line break. The `<hr>` tag creates a horizontal rule across the page. The following code shows how they are used:

Demo 3.3: Text/Demos/br-and-hr.html

```
-----Lines 1 through 7 Omitted-----
8.  <body>
9.    <p>Sometimes<br>it is desirable<br>to break text across lines.</p>
10.  <hr>
11.  <p>And to separate text with horizontal rules.</p>
12.  </body>
-----Line 13 Omitted-----
```

The following screenshot shows how they appear by default:



Notice that the `
` and `<hr>` tags are both *empty* tags, meaning that they do not contain any content, and therefore, do not have a corresponding closing tag.



3.4. The div Element

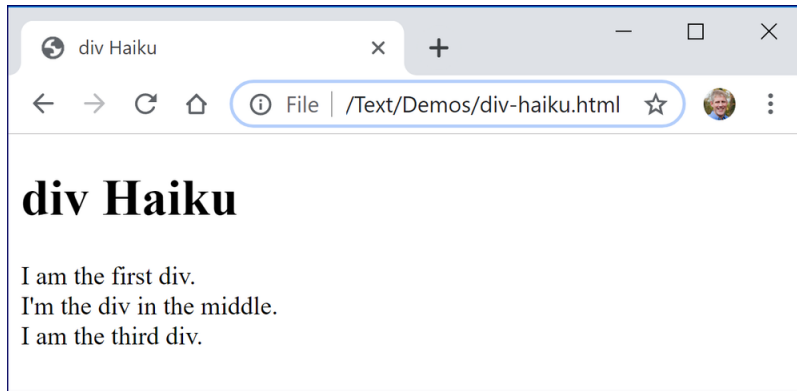
The `div` element (`<div>`) is used to create a content **division**. That is, it divides a segment of content from the surrounding content. Visually, this results in placing the content on its own block, similar in effect to putting a `
` tag before and after the content.

The following demo shows how `div` elements work:

Demo 3.4: Text/Demos/div-haiku.html

```
-----Lines 1 through 7 Omitted-----
8.  <body>
9.    <h1>div Haiku</h1>
10.  <div>I am the first div.</div>
11.  <div>I'm the div in the middle.</div>
12.  <div>I am the third div.</div>
13. </body>
-----Line 14 Omitted-----
```

The following screenshot shows how this will appear in the browser:

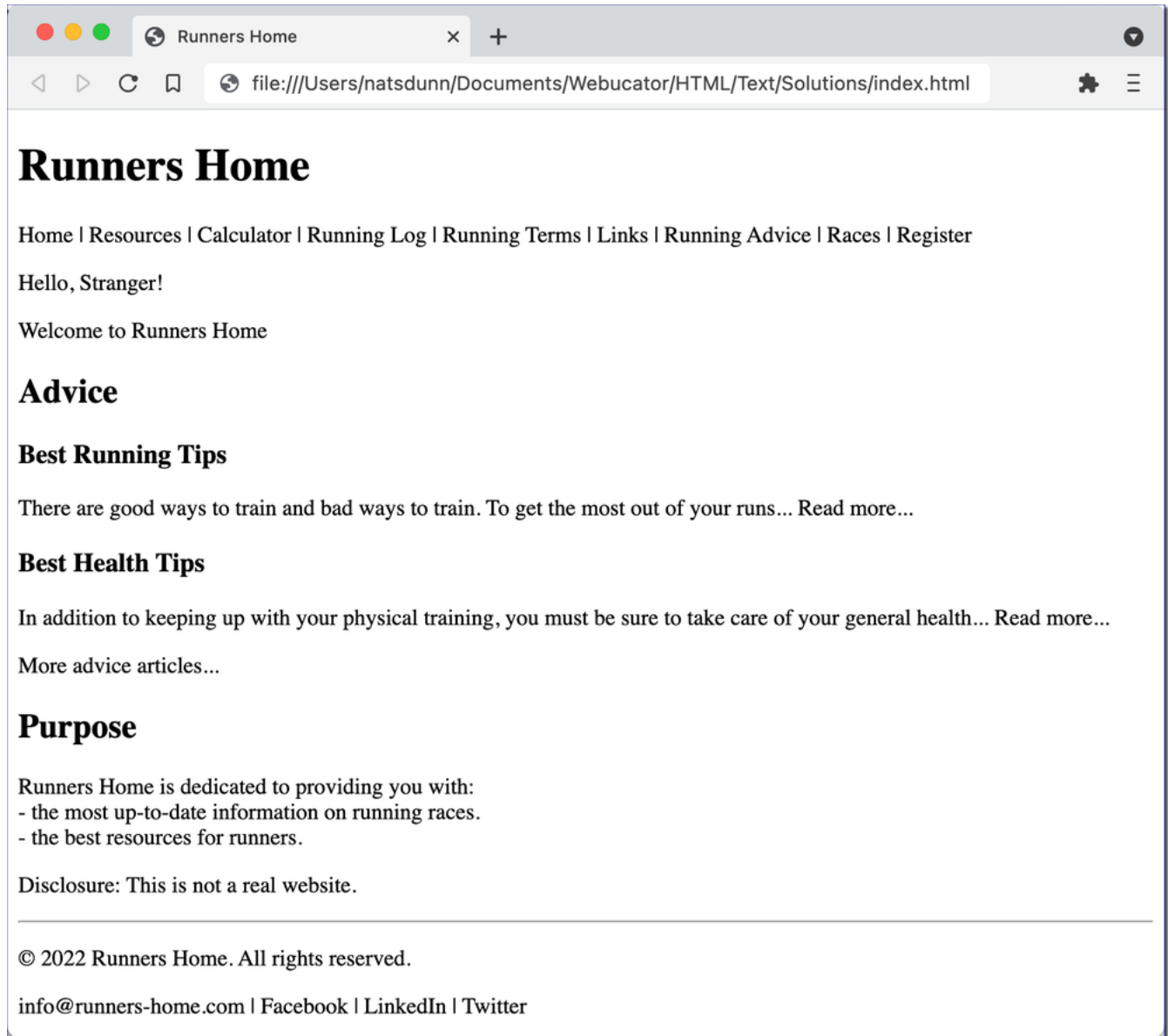


While the preceding demo illustrates how the `div` element works, it is more commonly used to separate larger blocks of content. You will learn more about this in the Sectioning a Web Page lesson (see page 101).

Exercise 2: Creating an HTML Page

🕒 15 to 25 minutes

In this exercise, you will create an HTML page from scratch. It should look like this:



1. Create a new page in Visual Studio Code and save it as `index.html` in the `Text/Exercises` directory.
2. Write code to make the page look like the one in the screenshot above.

3. Save your work and open your new page in a browser to test it.

Challenge

Use special characters instead of the dashes to make more interesting bullets. Try •

Solution: Text/Solutions/index.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.  <title>Runners Home</title>
7.  </head>
8.  <body>
9.  <h1>Runners Home</h1>
10. <div>
11.   Home | Resources | Calculator | Running Log | Running Terms |
12.   Links | Running Advice | Races | Register
13. </div>
14. <p>Hello, Stranger!</p>
15. <p>Welcome to Runners Home</p>
16. <h2>Advice</h2>
17. <h3>Best Running Tips</h3>
18. <p>There are good ways to train and bad ways to train. To get
19.   the most out of your runs... Read more...</p>
20. <h3>Best Health Tips</h3>
21. <p>In addition to keeping up with your physical training, you
22.   must be sure to take care of your general health...
23.   Read more...</p>
24. <p>More advice articles...</p>
25. <h2>Purpose</h2>
26. <p>Runners Home is dedicated to providing you with:<br>
27.   - the most up-to-date information on running races.<br>
28.   - the best resources for runners.
29. </p>
30. <p>Disclosure: This is not a real website.</p>
31. <hr>
32. <p>&copy; 2022 Runners Home. All rights reserved.</p>
33. <div>
34.   info@runners-home.com |
35.   Facebook |
36.   LinkedIn |
37.   Twitter
38. </div>
39. </body>
40. </html>
```

Challenge Solution: Text/Solutions/index-challenge.html

```
-----Lines 1 through 25 Omitted-----
26. <p>Runners Home is dedicated to providing you with:<br>
27.     &#8226; the most up-to-date information on running races.<br>
28.     &#8226; the best resources for runners.
29. </p>
-----Lines 30 through 40 Omitted-----
```

3.5. Quoted Text

*Evaluation
Copy*

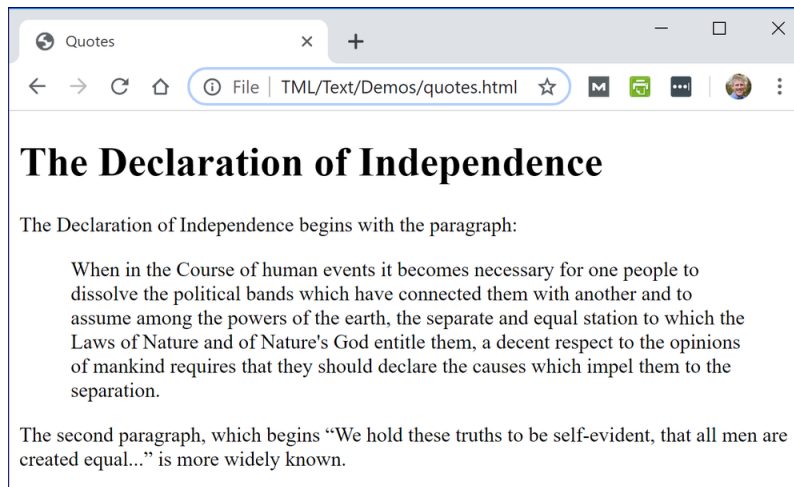
The `<blockquote>` and `<q>` tags are used to designate quoted text. Both elements can take the `cite` attribute, which is used to reference the source. The value of the `cite` attribute, which is used to point to a URL with information about the quote, will not be visible on the page by default, but could be made accessible using JavaScript.

`blockquote` is a block-level element, while `q` is an inline element. See the following example:

Demo 3.5: Text/Demos/quotes.html

```
-----Lines 1 through 7 Omitted-----
8.  <body>
9.    <h1>The Declaration of Independence</h1>
10.   <p>The Declaration of Independence begins with the paragraph:</p>
11.   <blockquote
12.     cite="https://www.ushistory.org/declaration/document/index.html">
13.     <p>When in the Course of human events it becomes necessary for one
14.       people to dissolve the political bands which have connected them
15.       with another and to assume among the powers of the earth, the
16.       separate and equal station to which the Laws of Nature and of
17.       Nature's God entitle them, a decent respect to the opinions of
18.       mankind requires that they should declare the causes which impel
19.       them to the separation.</p>
20.   </blockquote>
21.
22.   <p>The second paragraph, which begins
23.   <q cite="https://www.ushistory.org/declaration/document/index.html">We
24.   hold these truths to be self-evident, that all men are created
25.   equal...</q> is more widely known.</p>
26. </body>
-----Line 27 Omitted-----
```

Most browsers add margins to blockquotes on both the left and right and wrap text nested in <q> tags with quotes. Google Chrome renders this page as follows:



Some notes:

1. Modern browsers don't do anything visual with the cite attribute.

2. Blockquotes should **not** be used for formatting purposes. If you want to add margins around an element, you should use Cascading Style Sheets (CSS).
3. Blockquotes cannot be contained within paragraphs.
4. Blockquotes cannot have text as a direct child. Usually, blockquotes contain paragraphs (<p> tags).



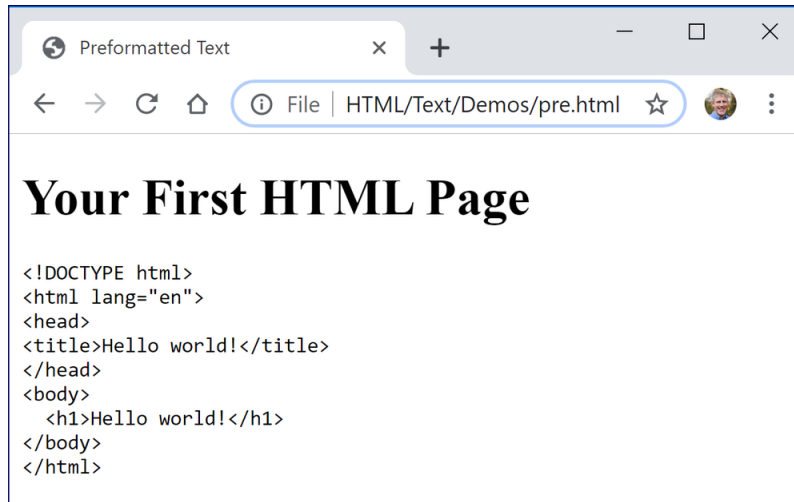
3.6. Preformatted Text

Occasionally, it is desirable to output content as it is laid out in the code, *whitespace and all*. The <pre> tag is used for this purpose. It is often used in online coding tutorials so that the whitespace shown in the tutorial reflects how it would appear in the document it represents. The following code shows how <pre> is used.

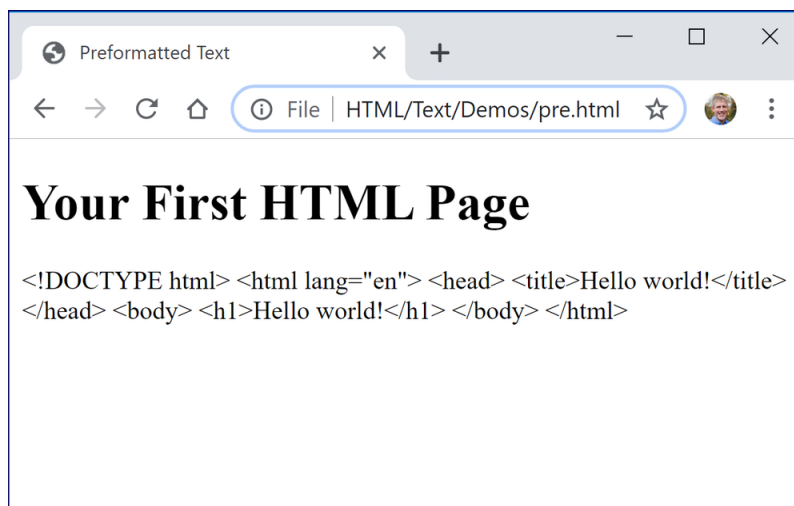
Demo 3.6: Text/Demos/pre.html

```
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.  <title>Preformatted Text</title>
7.  </head>
8.  <body>
9.  <h1>Your First HTML Page</h1>
10. <pre>
11.  &lt;!DOCTYPE html&gt;
12.  &lt;html lang="en"&gt;
13.  &lt;head&gt;
14.  &lt;title&gt;Hello world!&lt;/title&gt;
15.  &lt;/head&gt;
16.  &lt;body&gt;
17.    &lt;h1&gt;Hello world!&lt;/h1&gt;
18.  &lt;/body&gt;
19.  &lt;/html&gt;
20. </pre>
21. </body>
22. </html>
```

The page is rendered as follows:



Notice that the whitespace within the `<pre>` tags is not condensed. Remove the `<pre>` tags and this page will appear like this:



3.7. Inline Semantic Elements

Semantic (*adj.*): Of or relating to meaning, especially meaning in language.¹⁰

Inline semantic elements provide meaningful information about the content they contain. The most common elements of this type are `em` and `strong`. Both elements indicate that the content should be

¹⁰. <https://www.ahdictionary.com/word/search.html?q=semantic>

emphasized. `` indicates stronger emphasis than ``. Most browsers bold `` content and italicize `` content. Inline semantic elements are listed below:¹¹

1. `<abbr>` – Used for abbreviations. Use the `title` attribute for the unabbreviated version. Default styling varies.
2. `` – Used to stylistically offset text without conveying any extra importance to it. Default styling is usually **bold**. Use sparingly, if at all.
3. `<cite>` – Used to cite a creative work. Default styling is usually *italic*.
4. `<code>` – Used to denote computer code. Default styling is usually monospace.
5. `<dfn>` – Used to indicate a term being defined. Default styling is usually *italic*.
6. `` – Used to add emphasis to text. Default styling is usually *italic*.
7. `<i>` – Used to convey an alternate voice or mood. Default styling is usually *italic*. Use sparingly, if at all.
8. `<kbd>` – Used to denote user input (e.g., from a keyboard or a voice input). Default styling is usually monospace.
9. `<mark>` – Used to mark text of special interest or importance. Default styling is usually highlighted in some way.
10. `<s>` – Used to mark text as no longer accurate or relevant. Default styling is to put a line through the text.¹²
11. `<samp>` – Used to denote output from a computer program. Default styling is usually monospace.
12. `<small>` – Used to represent text as a side comment or “small print.” Default styling is usually smaller than surrounding text.
13. `` – Used as a generic wrapper of inline content. The `` tag can be used for grouping elements, and meaning and style can be added through adding attributes (e.g., `id`, `class`, and `lang`).
14. `` – Used to add extra emphasis or importance to text. Default styling is usually **bold**.
15. `<sub>` – Used to denote a subscript.
16. `<sup>` – Used to denote a superscript.

11. For a complete list, see https://developer.mozilla.org/en-US/docs/Web/HTML/Element#inline_text_semantics.

12. There are special elements used to indicate editorial changes: `del` (for deleted text) and `ins` (for inserted text). The `del` and `ins` elements are usually used in draft documents or in documents that have an updated version.

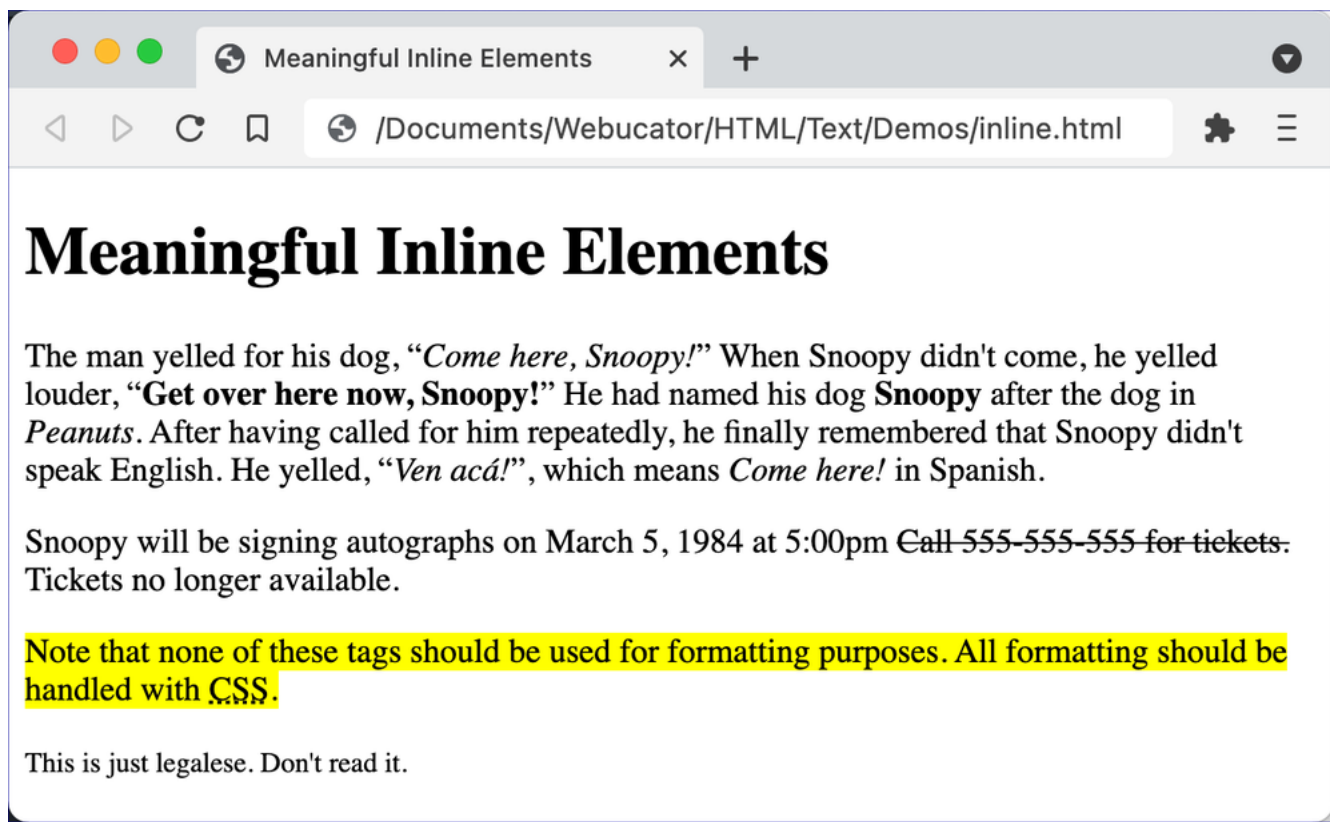
17. `<time>` – Used to denote a date and/or time. It can include a `datetime` attribute with a machine-readable format of the enclosed date and/or time.
18. `<u>` – Indicates that text should have some form of non-textual annotation applied. By default, this is an underline, but if you use this element, you should probably change the default rendering to something different, so that it doesn't look like a link. Avoid using the `<u>` tag unless you have a specific semantic use case such as drawing attention to spelling or grammatical errors.

The following example shows how these inline elements are used.

Demo 3.7: Text/Demos/inline.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.  <title>Meaningful Inline Elements</title>
7.  </head>
8.  <body>
9.  <h1>Meaningful Inline Elements</h1>
10. <p>
11.   The man yelled for his dog, <q><em>Come here, Snoopy!</em></q>
12.   When Snoopy didn't come, he yelled louder,
13.   <q><strong>Get over here now, Snoopy!</strong></q>
14.   He had named his dog <b>Snoopy</b> after the dog in
15.   <cite>Peanuts</cite>. After having called for him repeatedly,
16.   he finally remembered that Snoopy didn't speak English.
17.   He yelled, <q><dfn lang="es">Ven acá!</dfn></q>, which means
18.   <i>Come here!</i> in Spanish.
19. </p>
20. <p>
21.   Snoopy will be signing autographs on <time datetime="1984-03-05t17:00">March
22.   5, 1984 at 5:00pm</time>
23.   <s>Call 555-555-555 for tickets.</s> Tickets no longer available.
24. </p>
25. <p>
26.   <mark>
27.     Note that none of these tags should be used for formatting
28.     purposes. All formatting should be handled with
29.     <abbr title="Cascading Style Sheets">CSS</abbr>.
30.   </mark>
31. </p>
32. <p><small>This is just legalese. Don't read it.</small></p>
33. </body>
34. </html>
```

The page is rendered as follows:



All of these formatting effects can be created with CSS, so if you just want to change the formatting without implying any specific meaning, you should use CSS instead.

We recommend avoiding the `` and `<i>` tags. In most cases, `` and `` are more appropriate.

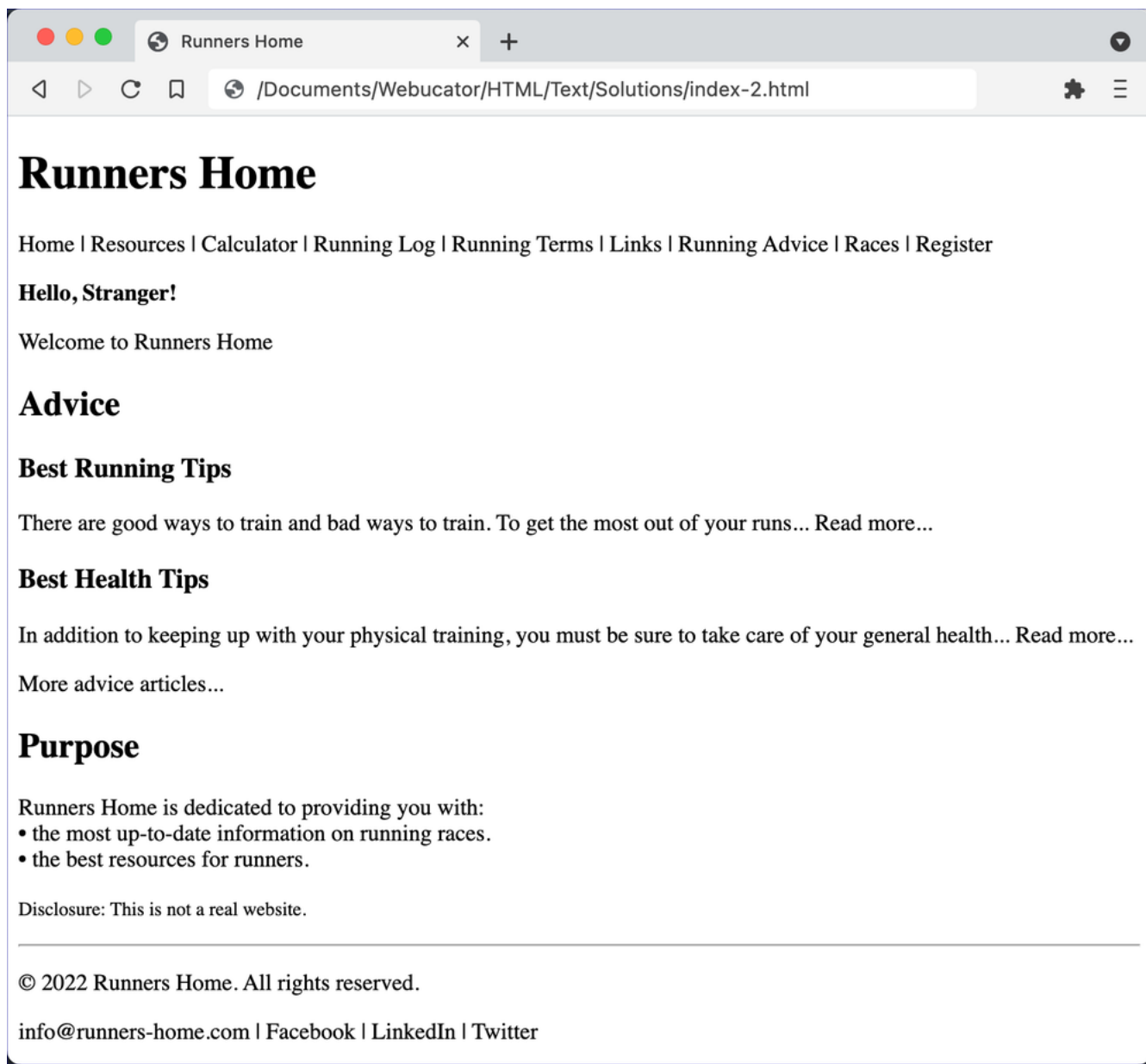
Exercise 3: Adding Inline Elements

⌚ 5 to 10 minutes

In this exercise, you will add inline elements to the Runners Home home page.

1. Open `index.html` from the `Text/Exercises` directory in Visual Studio Code.
2. Make the text “Hello, Stranger!” strongly emphasized.
3. Make the text “Disclosure: This is not a real website.” small.
4. Save your work and open your new page in a browser to test it.

Your finished page should look like this:



You are welcome to play around with additional tags. In fact, we encourage you to do so.

Solution: Text/Solutions/index-2.html

```
-----Lines 1 through 8 Omitted-----
9.  <h1>Runners Home</h1>
10. <div>
11.   Home | Resources | Calculator | Running Log | Running Terms |
12.   Links | Running Advice | Races | Register
13. </div>
14. <p><strong>Hello, Stranger!</strong></p>
15. <p>Welcome to Runners Home</p>
16. <h2>Advice</h2>
17. <h3>Best Running Tips</h3>
18. <p>There are good ways to train and bad ways to train. To get
19.   the most out of your runs... Read more...</p>
20. <h3>Best Health Tips</h3>
21. <p>In addition to keeping up with your physical training, you
22.   must be sure to take care of your general health...
23.   Read more...</p>
24. <p>More advice articles...</p>
25. <h2>Purpose</h2>
26. <p>Runners Home is dedicated to providing you with:<br>
27.   &#8226; the most up-to-date information on running races.<br>
28.   &#8226; the best resources for runners.
29. </p>
30. <p><small>Disclosure: This is not a real website.</small></p>
31. <hr>
32. <p>&copy; 2022 Runners Home. All rights reserved.</p>
33. <div>
34.   info@runners-home.com |
35.   Facebook |
36.   LinkedIn |
37.   Twitter
38. </div>
-----Lines 39 through 40 Omitted-----
```

Conclusion

In this lesson, you have learned to work with paragraphs, headings, and other text elements. You can now create a basic HTML page.

LESSON 4

HTML Links

Topics Covered

- ☑ Basic text links.
- ☑ Absolute and relative paths.
- ☑ Links that open in new tabs or windows.
- ☑ Email links.
- ☑ Links to specific locations on a page.

Introduction

The ability to link from one page to another is what makes HTML hyper. Calling it Hypertext, however, is a bit of a misnomer, as images can also be linked.



4.1. Text Links

The tag for a link is perhaps the least intuitive of all the HTML tags. It is `<a>`, and it comes from the word “anchor,” as `<a>` tags used to be used to create locations to link to, known as *anchors*, as well as the links themselves. By itself, the `<a>` tag does nothing. To create a link, it requires the `href` attribute, which takes as a value the path to the file or location to which to link. The syntax is as follows:

```
<a href="path_to_file">Link Text</a>
```

A couple of examples:

```
<a href="bios/john-lennon.html">John Lennon</a>  
<a href="https://www.webucator.com">Webucator</a>
```



4.2. Absolute vs. Relative Paths

Paths are absolute or relative:

- Absolute paths always start from the top-level directory (the *web root*) and work their way downward toward the referenced file.
- Relative paths start from the current location (the location of the file containing the path) and work their way to the referenced file from that location.

For the examples in this section, we will use the following directory tree:

```
└─ wwwroot  
  └─ about  
    <> company.html  
    <> contact.html  
    <> partners.html  
  └─ bios  
    <> george-harrison.html  
    <> john-lennon.html  
    <> paul-mccartney.html  
    <> ringo-starr.html  
  └─ images  
<> index.html
```

Things to notice:

1. The `index.html` file on the bottom is a direct child of the `wwwroot` folder.
2. The `about`, `bios`, and `images` folders are also children of the `wwwroot` folder. They each contain their own files.

Assume that this site is located at `https://www.example.com` and that the `wwwroot` folder is the web root, meaning that it is the top-level directory. This means that when a user visits `https://www.ex`

ample.com/index.html, the index.html page within the wwwroot folder will be downloaded to the browser.

❖ 4.2.1. Absolute Paths

An absolute path shows the complete path to a file starting from the web root.

The absolute path to the web root from a page on the same domain is simply a forward slash (/). So, given the folder structure shown above, a link on company.html to index.html could be written like this:

```
<a href="/index.html">Home Page</a>
```

The same link could be placed on any page in any folder below the wwwroot folder or on any page in the wwwroot folder itself.

Using an absolute path, a link to company.html would include the about directory, like this:

```
<a href="/about/company.html">About Our Company</a>
```

External Links

When linking to a file at a different domain, you must identify the location of the domain using the domain name (or IP address) of the site. Again, assume that the directory structure shown above is found at https://www.example.com. A link to company.html from another site would be written like this:

```
<a href="https://www.example.com/about/company.html">About the Beatles</a>
```

❖ 4.2.2. Relative Paths

Relative paths can only be used to link to other files under the same web root. A relative path indicates where a file is *relative to* the file that contains the link. The folder (or directory) that contains the file being worked on is called the *current directory*. The relative path to another file that is also in the current directory is just the name of that file. For example, since company.html and contact.html are found in the same directory, they can link to each other simply by specifying the file name. The following shows a link that could be used in contact.html to company.html:

```
<a href="company.html">About Our Company</a>
```

The relative path to a file in a subdirectory of the current directory must include the name of the subdirectory. For example, to link to `company.html` from `index.html` you must first point to the `about` directory, like so:

```
<a href="about/company.html">About Our Company</a>
```

The relative path to a file in a directory above the current directory should begin with `../`. For example, the following shows a link to `index.html` from `company.html`:

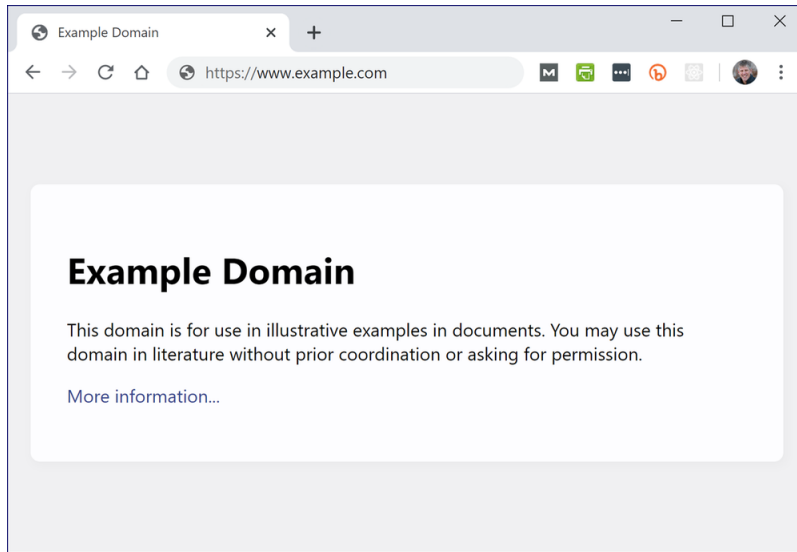
```
<a href="../../index.html">Home Page</a>
```

❖ 4.2.3. Default Pages

The web server administrator can set up default page names to look for when a path ends with a directory name without specifying a file. Often these files are called `index.html`. In this case, the following URLs would be identical, all loading `index.html`:

- `https://www.example.com`
- `https://www.example.com/`
- `https://www.example.com/index.html`

You can give this a try by visiting those pages at `example.com`, which is an actual website used for demonstrating just this sort of thing:



4.3. Targeting New Tabs

The `target` attribute is used to specify the browser tab (or window) in which the linked page will open. For example:

```
<a href="company.html" target="newtab">Our Company</a>
```

If there is no open browser tab with the specified `target` name, a new tab will be opened with that name. As long as that tab stays open, future links with the same `target` value will target that tab.

Note that “newtab” has no special meaning. We could name it “external,” “newwin,” “roxanne,” or anything else we want.

Try it out by doing the following:

1. Open `Links/Demos/links.html` in Visual Studio Code and review the first two links below the **Targeting New Tabs** heading:

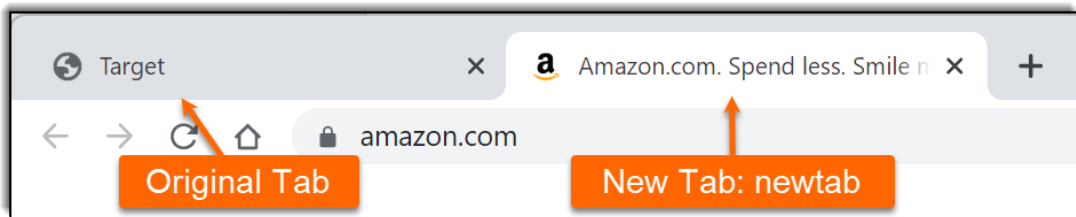
```
<a href="https://www.amazon.com" target="newtab">Amazon</a>  
<a href="https://www.nytimes.com" target="newtab">NY Times</a>
```

2. Open `Links/Demos/links.html` in your browser.

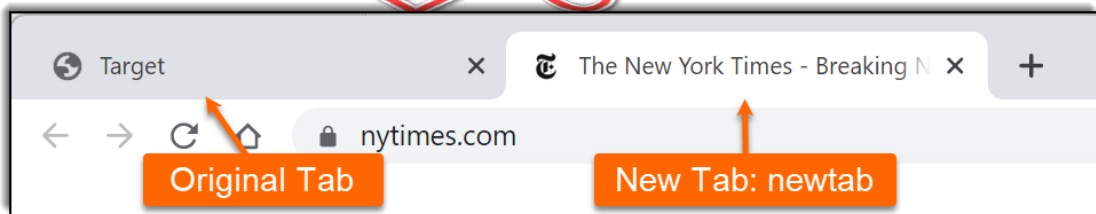
- Under **Targeting New Tabs**, click the **Amazon** link:



This link has a target of newtab. The page will open in a new tab:



- Without closing the tab with Amazon in it, go back to the tab that has `links.html` open.
- Under **Targeting New Tabs**, click the **NY Times** link. This link also has a target of newtab. The page will open in the same tab in which Amazon opened:



Other links targeting newtab would also open in that same tab.

❖ 4.3.1. `_blank` Target

To force each link to target a brand new tab or window, use `_blank` as the value of the `target` attribute as shown here:

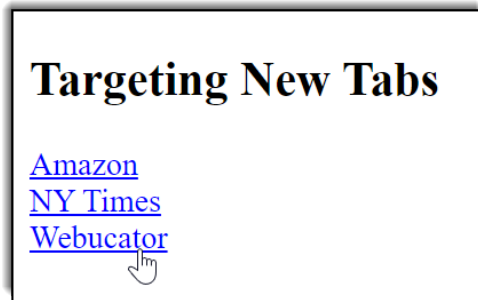
```
<a href="company.html" target="_blank">Our Company</a>
```

Try it out by doing the following:

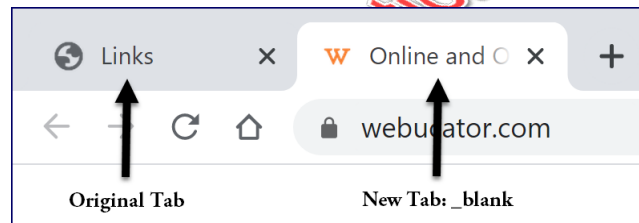
1. Open `Links/Demos/links.html` in Visual Studio Code and review the third link below the **Targeting New Tabs** heading:

```
<a href="https://www.webucator.com" target="_blank">Webucator</a>
```

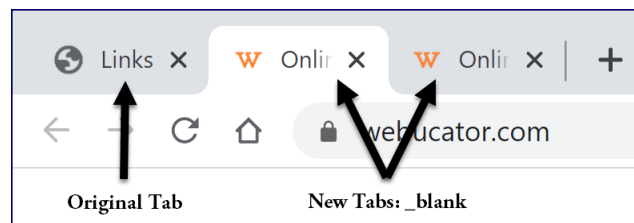
2. Open `Links/Demos/links.html` in your browser if it's not already open.
3. Under **Targeting New Tabs**, click the **Webucator** link:



This link has a target of `_blank`. The page will open in a new tab:



4. Without closing the tab with Webucator in it, go back to the tab that has `links.html` open.
5. Under **Targeting New Tabs**, click the **Webucator** link again. Rather than reusing the same tab, it will open the page in another brand new tab:



As a rule of thumb, if you're going to have links open in new tabs, we would use a named target rather than the generic `_blank`, so that the user doesn't get inundated with new tabs.

As an even more important rule of thumb, we would avoid targeting new tabs/windows altogether. Some websites do this so that the linked page will not replace their web page, but it is bad design. Generally, you do not want to surprise your users, who are accustomed to clicking the **Back** button to get back to a page they were just on. When a link opens in a new tab or window, users cannot click the **Back** button to get back to your page. Instead, they have to know to go back to the tab that your page is on, making it even more difficult to get back to your page. An exception would be when you are quite certain that the user wants to stay on your page because they are following a setup guide or a tutorial. In this case, it can be helpful to target new tabs, so that the user can easily toggle back and forth between the instructions they are following and the linked pages.



4.4. Email Links

Email links are used to open an email client to start a new email message. The syntax is similar to the links we have seen thus far. However, for email links, the value of the `href` attribute must begin with `mailto:` and ends with an email address. For example:

```
<a href="mailto:paul@example.com">Email Paul</a>
```

It is good practice to include the email address as the text of the link, so that people who are printing the page or whose setup does not support email links can see the actual email address. For example:

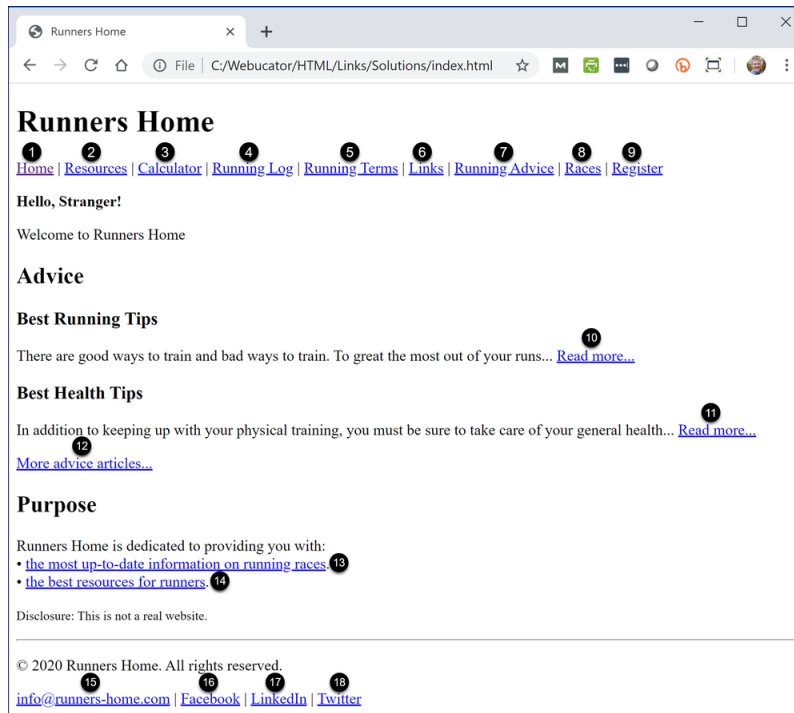
```
Email Paul at <a href="mailto:paul@example.com">paul@example.com</a>.
```



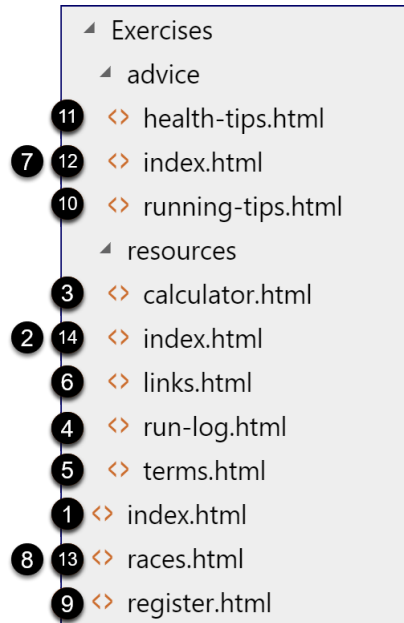
Exercise 4: Adding Links

⌚ 40 to 60 minutes

In this exercise, you will add links to the pages of the Runners Home website. The home page (Links/Exercises/index.html) looks like this:



The links above the footer should go to the following pages:



The footer links should go to:

15. info@runners-home.com. This should be an email link.
16. <https://www.facebook.com/webucator>
17. <https://www.linkedin.com/companies/webucator>
18. <https://twitter.com/webucator>

After you have finished adding links to the home page, open each of the other pages in the Links/Exercises folder and add the same header and footer links that you added in `index.html`. Copy and paste is your friend, but be careful: relative links on pages in subfolders will be different from links in the root folder and from links in other subfolders.

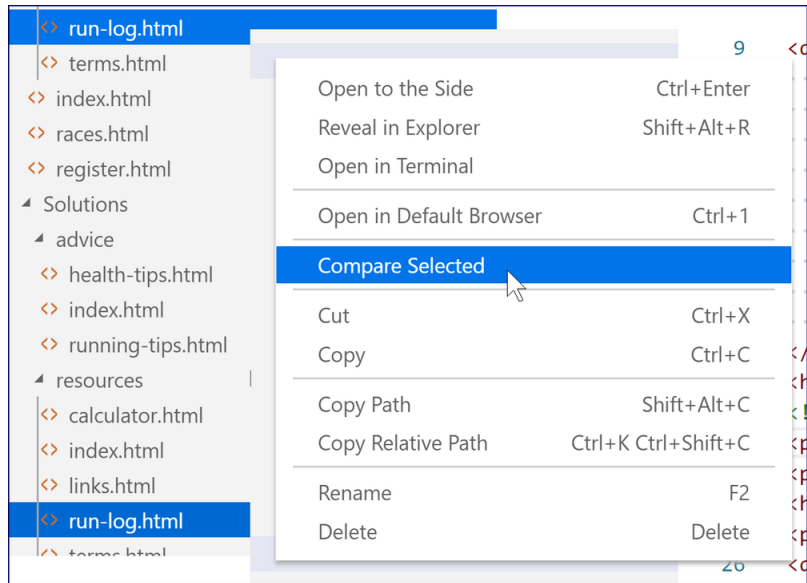
In addition to the header and footer links, add the following links:

1. `races.html`
 - A. “AJC Peachtree Road Race”: <https://www.atlantatrackclub.org/peachtree>
 - B. “Boilermaker”: <https://www.boilermaker.com>
 - C. “NYC Marathon”: <https://www.nyrr.org/>
2. `advice/index.html`
 - A. “Running Tips”: the **Running Tips** page.
 - B. “Health Tips”: the **Health Tips** page.

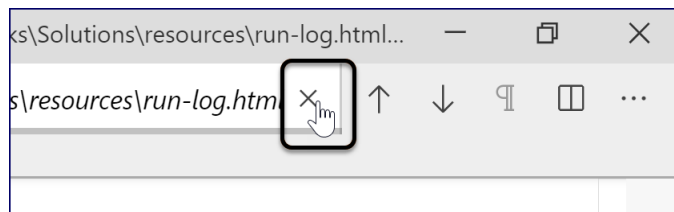
3. `advice/health-tips.html` and `advice/running-tips.html`
 - A. “More Advice Articles”: **Running Advice** index page.
4. `resources/index.html`
 - A. “Calculator”: the **Calculator** page.
 - B. “Running Log”: the **Running Log** page.
 - C. “Running Terms”: the **Running Terms** page.
 - D. “Links”: the **Links** page.
5. `resources/links.html`
 - A. “Map My Run”: <https://www.mapmyrun.com>
 - B. “Runners World”: <https://www.runnersworld.com>
 - C. “Strength Running”: <https://strengthrunning.com>
 - D. “More Resources”: **Resources** index page.
6. `resources/calculator.html`, `resources/run-log.html`, and `resources/terms.html`
 - A. “More Resources”: **Resources** index page.

You may find Visual Studio Code’s **Compare Selected** tool useful for comparing the solutions you did in the Exercises folder with the ones we included in the Solutions folder.

1. In Visual Studio Code’s **Explorer** panel, hold down the **Ctrl** key and click the two files you wish to compare. They should become highlighted.
2. Right-click one of the files and select **Compare Selected**:



3. You will get a side-by-side view of the files with differences highlighted.
4. Close the files when you are done comparing them:



Solution: Links/Solutions/index.html

```
-----Lines 1 through 9 Omitted-----
10. <div>
11.   <a href="index.html">Home</a> |
12.   <a href="resources/index.html">Resources</a> |
13.   <a href="resources/calculator.html">Calculator</a> |
14.   <a href="resources/run-log.html">Running Log</a> |
15.   <a href="resources/terms.html">Running Terms</a> |
16.   <a href="resources/links.html">Links</a> |
17.   <a href="advice/index.html">Running Advice</a> |
18.   <a href="races.html">Races</a> |
19.   <a href="register.html">Register</a>
20. </div>
21. <p><strong>Hello, Stranger!</strong></p>
22. <p>Welcome to Runners Home</p>
23. <h2>Advice</h2>
24. <h3>Best Running Tips</h3>
25. <p>There are good ways to train and bad ways to train. To get
26.   the most out of your runs...
27.   <a href="advice/running-tips.html">Read more...</a>
28. </p>
29. <h3>Best Health Tips</h3>
30. <p>In addition to keeping up with your physical training, you
31.   must be sure to take care of your general health...
32.   <a href="advice/health-tips.html">Read more...</a>
33. </p>
34. <p><a href="advice/index.html">More advice articles...</a></p>
35. <h2>Purpose</h2>
36. <p>Runners Home is dedicated to providing you with:<br>
37.   &#8226; <a href="races.html">the most up-to-date
38.     information on running races</a>.<br>
39.   &#8226; <a href="resources/index.html">the best
40.     resources for runners</a>.
41. </p>
42. <p><small>Disclosure: This is not a real website.</small></p>
43. <hr>
44. <p>&copy; 2022 Runners Home. All rights reserved.</p>
45. <div>
46.   <a href="mailto:info@runners-home.com">info@runners-home.com</a> |
47.   <a href="https://www.facebook.com/webucator">Facebook</a> |
48.   <a href="https://www.linkedin.com/companies/webucator">LinkedIn</a> |
49.   <a href="https://twitter.com/webucator">Twitter</a>
50. </div>
-----Lines 51 through 52 Omitted-----
```

Solution: Links/Solutions/races.html

```
-----Lines 1 through 20 Omitted-----
21. <h2>Races</h2>
22. <div>
23.     March 15th,
24.     <a href="https://www.atlantatrackclub.org/peachtree">
25.         AJC Peachtree Road Race
26.     </a>,
27.     10K, Atlanta, GA<br>
28.     July 11th,
29.     <a href="https://www.boilermaker.com">Boilermaker</a>,
30.     15K, Utica, NY<br>
31.     November 3rd, <a href="https://www.nyrr.org/">NYC Marathon</a>,
32.     26.22 mi, New York, NY
33. </div>
-----Lines 34 through 44 Omitted-----
```

The links in the header and footer of `races.html` are the same as they are in `index.html`.

The solution to `register.html` is not shown as it only has links in the header and footer, which are also the same as they are in `index.html`.

Solution: Links/Solutions/advice/index.html

```
-----Lines 1 through 9 Omitted-----
10. <div>
11.   <a href="../index.html">Home</a> |
12.   <a href="../resources/index.html">Resources</a> |
13.   <a href="../resources/calculator.html">Calculator</a> |
14.   <a href="../resources/run-log.html">Running Log</a> |
15.   <a href="../resources/terms.html">Running Terms</a> |
16.   <a href="../resources/links.html">Links</a> |
17.   <a href="index.html">Running Advice</a> |
18.   <a href="../races.html">Races</a> |
19.   <a href="../register.html">Register</a>
20. </div>
21. <h2>Running Advice</h2>
22. <div>
23.   <a href="running-tips.html">Running Tips</a><br>
24.   <a href="health-tips.html">Health Tips</a>
25. </div>
26. <p><small>Disclosure: This is not a real website.</small></p>
27. <hr>
28. <p>&copy; 2022 Runners Home. All rights reserved.</p>
29. <div>
30.   <a href="mailto:info@runners-home.com">info@runners-home.com</a> |
31.   <a href="https://www.facebook.com/webucator">Facebook</a> |
32.   <a href="https://www.linkedin.com/companies/webucator">LinkedIn</a> |
33.   <a href="https://twitter.com/webucator">Twitter</a>
34. </div>
-----Lines 35 through 36 Omitted-----
```

Solution: Links/Solutions/advice/health-tips.html

```
-----Lines 1 through 20 Omitted-----
21. <h2>Health Tips</h2>
22. <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
23.   Aliquam et gravida sapien, facilisis condimentum arcu.
24.   Morbi eget dui iaculis, porttitor eros et, tincidunt erat...</p>
25. <p><a href="index.html">More Advice Articles</a></p>
26. <p><small>Disclosure: This is not a real website.</small></p>
-----Lines 27 through 36 Omitted-----
```

The links in advice/running-tips.html are the same as in health-tips.html.

Solution: Links/Solutions/resources/index.html

```
-----Lines 1 through 9 Omitted-----
10. <div>
11.   <a href="../index.html">Home</a> |
12.   <a href="index.html">Resources</a> |
13.   <a href="calculator.html">Calculator</a> |
14.   <a href="run-log.html">Running Log</a> |
15.   <a href="terms.html">Running Terms</a> |
16.   <a href="links.html">Links</a> |
17.   <a href="../advice/index.html">Running Advice</a> |
18.   <a href="../races.html">Races</a> |
19.   <a href="../register.html">Register</a>
20. </div>
21. <h2>Resources</h2>
22. <div>
23.   - <a href="calculator.html">Calculator</a><br>
24.   - <a href="run-log.html">Running Log</a><br>
25.   - <a href="terms.html">Running Terms</a><br>
26.   - <a href="links.html">Links</a>
27. </div>
28. <p><small>Disclosure: This is not a real website.</small></p>
29. <hr>
30. <p>&copy; 2022 Runners Home. All rights reserved.</p>
31. <div>
32.   <a href="mailto:info@runners-home.com">info@runners-home.com</a> |
33.   <a href="https://www.facebook.com/webucator">Facebook</a> |
34.   <a href="https://www.linkedin.com/companies/webucator">LinkedIn</a> |
35.   <a href="https://twitter.com/webucator">Twitter</a>
36. </div>
-----Lines 37 through 38 Omitted-----
```

Solution: Links/Solutions/resources/links.html

```
-----Lines 1 through 9 Omitted-----
10. <div>
11.   <a href="../index.html">Home</a> |
12.   <a href="index.html">Resources</a> |
13.   <a href="calculator.html">Calculator</a> |
14.   <a href="run-log.html">Running Log</a> |
15.   <a href="terms.html">Running Terms</a> |
16.   <a href="links.html">Links</a> |
17.   <a href="../advice/index.html">Running Advice</a> |
18.   <a href="../races.html">Races</a> |
19.   <a href="../register.html">Register</a>
20. </div>
21. <h2>Useful Links</h2>
22. <div>
23.   - <a href="https://www.mapmyrun.com">Map My Run</a><br>
24.   - <a href="https://www.runnersworld.com">Runners World</a><br>
25.   - <a href="https://strengthrunning.com">Strength Running</a>
26. </div>
27. <p><a href="index.html">More Resources</a></p>
28. <p><small>Disclosure: This is not a real website.</small></p>
29. <hr>
30. <p>&copy; 2022 Runners Home. All rights reserved.</p>
31. <div>
32.   <a href="mailto:info@runners-home.com">info@runners-home.com</a> |
33.   <a href="https://www.facebook.com/webucator">Facebook</a> |
34.   <a href="https://www.linkedin.com/companies/webucator">LinkedIn</a> |
35.   <a href="https://twitter.com/webucator">Twitter</a>
36. </div>
-----Lines 37 through 38 Omitted-----
```

The links in the header and footer of resources/links.html are the same as they are in resources/index.html.

The header, footer, and “More Resources” links in resources/calculator.html, resources/run-log.html, and resources/terms.html are the same as in resources/links.html.



4.5. Lorem Ipsum

You may have noticed that some of the pages on our site use the following text:

Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Aliquam et gravida sapien, facilisis condimentum arcu.
Morbi eget dui iaculis, porttitor eros et, tincidunt erat...

This *lorem ipsum* text is commonly used as dummy placeholder text. You can copy it from <https://www.lipsum.com>.



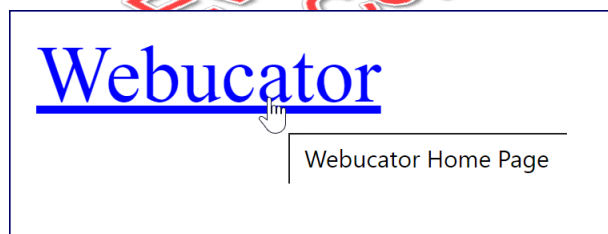
4.6. The title Attribute

The `title` attribute can be used to provide a description about a link. The description is displayed by the browser as a tooltip. Screen readers might read the description to a blind user.

Here's an example:

```
<a href="https://www.webucator.com" title="Webucator Home Page">Webucator</a>
```

When you hover over this link, the `title` appears as a tip.



This is especially useful when using an image as a link as it might not be clear from the image itself where the link points. The `title` attribute can be used to make the link destination clear. You will learn how to do this in the next lesson.



4.7. Targeting a Specific Location on the Page

Every HTML element can take an `id` attribute, which uniquely identifies that element on the page. The value of the `id` attribute must not contain any whitespace.

To target a specific element on the page, the link's href value should point to that element's id attribute prefaced with a number sign.

For example, assume you have an <h2> tag with the id of "john":

```
<h2 id="john">John Lennon</h2>
```

To target that location, use:

```
<a href="#john">Read about John</a>
```

You can also link to locations on other pages:

```
<a href="about.html#john">Read about John</a>
```

Or:

```
<a href="https://www.example.com/about.html#john">Read about John</a>
```

The following file shows more examples:

Demo 4.1: Links/Demos/location-links.html

```
-----Lines 1 through 8 Omitted-----
9.  <h1>Targeting Locations on a Page</h1>
10. <h2>Links to Locations on Remote Pages</h2>
11. <div>
12.   <a href="https://www.runners-home.com/resources/terms.html#dnf">DNF</a>
13. </div>
14. <h2>Links to Locations on This Page</h2>
15. <div>
16.   <a href="#alice"
17.     title="A MAD TEA-PARTY - Lewis Carroll">A Mad Tea-Party</a><br>
18.   <a href="#cinderella"
19.     title="CINDERELLA - the brothers Grimm">Cinderella</a><br>
20.   <a href="#naughtyboy"
21.     title="THE NAUGHTY BOY - H.C. Andersen">The Naughty Boy</a>
22. </div>
23. <hr>
24. <h2>Locations on This Page</h2>
25. <p>Each title below has an id attribute.</p>
26. <h3 id="alice">A MAD TEA-PARTY - Lewis Carroll</h3>
27. <p>There was a table set out under a tree in front of the house, and
-----Lines 28 through 31 Omitted-----
32.   asleep, I suppose it doesn't mind."
33.   <a href="https://www.gutenberg.org/files/11/11-h/11-h.htm#link2HCH0007">
34.     Continue reading</a></p>
35. <hr>
36. <h3 id="cinderella">CINDERELLA - the brothers Grimm</h3>
37. <p>The wife of a rich man fell sick: and when she felt that her end
-----Lines 38 through 48 Omitted-----
49.   on, and laughed at her and turned her into the kitchen.
50.   <a href="https://www.gutenberg.org/files/11027/11027-h/11027-h.htm#cinderella">
51.     Continue reading</a></p>
52. <hr>
53. <h3 id="naughtyboy">THE NAUGHTY BOY - Hans Christian Andersen</h3>
54. <p>Along time ago, there lived an old poet, a thoroughly kind old
-----Lines 55 through 57 Omitted-----
58.   blazed and the roasting apple hissed.
59.   <a href="https://www.gutenberg.org/files/1597/1597-h/1597-h.htm#link2H_4_0017">
60.     Continue reading</a></p>
61. <hr>
62. <div><a href="#top">Back to top</a></div>
63. </body>
64. </html>
```

The “top” Keyword

You will notice in the demo that the last link is to “#top”, but there is no element on the page with the id “top”. That is because “top” is a keyword. Browsers know that “top” references the top of the page.

Conclusion

In this lesson, you have learned to create text links, to work with absolute and relative paths, to target new tabs, to create email links, and to create and link to specific locations on a page.

LESSON 5

HTML Images

Topics Covered

- ☑ Adding images to a website.
- ☑ Creating image links.
- ☑ Making images accessible.
- ☑ Providing image fallbacks.

Introduction

Modern browsers support several types of images, including:

- Graphics Interchange Format (GIF)
- Joint Photographic Expert Group image (JPEG)
- Portable Network Graphics (PNG)
- Scalable Vector Graphics (SVG)
- Web Picture format (WebP)

WebP is generally the best choice for both image quality and compression and is supported by Chrome, Edge, Firefox, Opera, and Safari, but not by Internet Explorer.



5.1. Inserting Images

The `` tag is used to include an image in an HTML page. The `` tag is an empty tag, meaning it has no closing tag. Its `src` attribute is used to reference an image file using a relative or absolute path. Here is the syntax:

```

```

The following demo shows how to use the `` tag:

Demo 5.1: Images/Demos/images.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.  <title>Images</title>
7.  </head>
8.  <body>
9.  <h1>Images</h1>
10. <p></p>
11. <hr>
12. <p></p>
13. <hr>
14. <p></p>
15. </body>
16. </html>
```

The page will render as follows:¹³

13. The https://commons.wikimedia.org/wiki/File:R._John_Wright_Winnie_the_Pooh_Bear.jpg image is used under the terms of GNU Free Documentation License, version 1.2 (https://commons.wikimedia.org/wiki/ Commons:GNU_Free_Documentation_License,_version_1.2).



❖ 5.1.1. Making Images Accessible

Alternative Text

To add alternative text for an image, use the `alt` attribute as shown below:

```

```

Alternative text is displayed...

1. When the user's browser does not support images.
2. As the image is downloading.
3. When the user hovers over the image with the mouse (in some browsers).

Most importantly, alternative text is used by screen readers to describe an image for the visually impaired.

Providing Longer Descriptions

If an image depicts something complicated, such as a graph or chart, a longer description of the image can be provided using the `aria-describedby` attribute.¹⁴

❖ 5.1.2. height and width Attributes

The `img` element also takes `height` and `width` attributes that set the dimensions (in pixels) to use to display the image on the page. It is not good practice to “resize” the image using these attributes. If you use these attributes, you should set the values to the actual height and width of the image. Using the `height` and `width` attributes lets the browser know how much space to allocate for the image without having to wait for the image itself to download. Use of these attributes may promote faster rendering of the web page:

```

```



5.2. Image Links

To create an image link, wrap an `<a>` tag around your image, like so:

¹⁴. See <https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Attributes/aria-describedby> for information on the `aria-describedby` attribute.

```
<a href="index.html" title="Link to Home Page">
  
</a>
```

The following demo shows some image links:

Demo 5.2: Images/Demos/image-links.html

```
-----Lines 1 through 8 Omitted-----
9.  <h1>Image Links</h1>
10. <a href="https://www.google.com" title="Visit Google">
11.   
12. </a>
13. <hr>
14. <a href="https://en.wikipedia.org/wiki/Hundred_Acre_Wood"
15.   title="Visit site about Winnie the Pooh">
16.   
17. </a>
18. <hr>
19. <a href="../Solutions/index.html" title="Link to Home Page">
20.   
21. </a>
-----Lines 22 through 23 Omitted-----
```

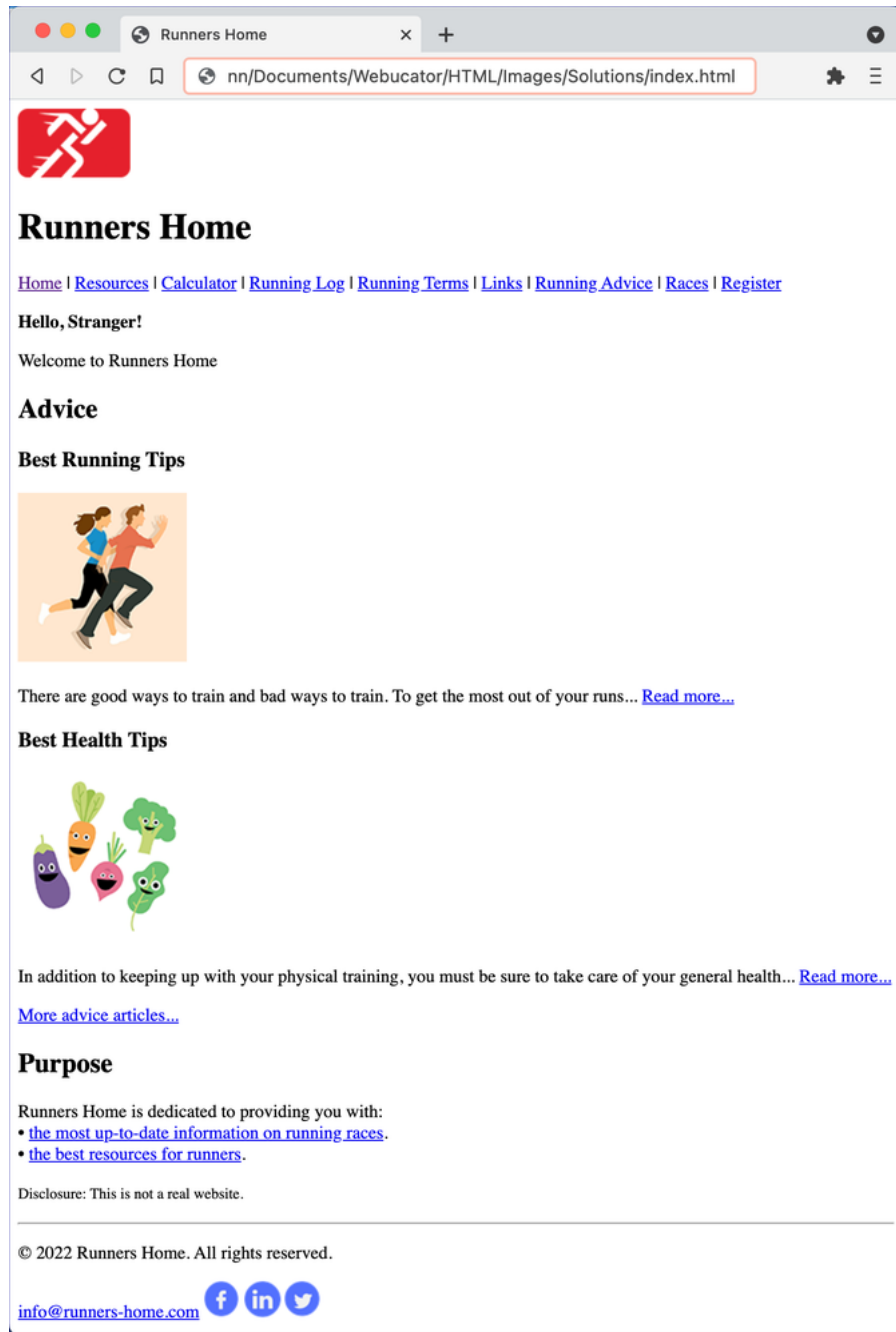
Including the title attribute results in a tooltip in many browsers:



Exercise 5: Adding Images to the Page

 15 to 25 minutes

In this exercise, you will add images to `index.html`. Here is the resulting page:



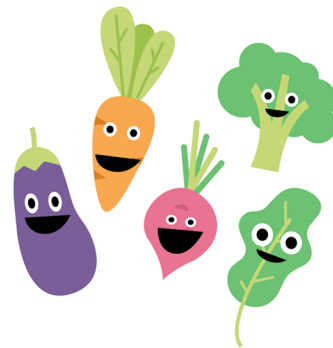
1. Open `Images/Exercises/index.html` for editing.
2. Add the following images, all of which are in the `Images/Exercises/images` directory. Be sure to include alternative text.
 - A. `runners-home.png` should link to `index.html`.



- B. `tips-running.png`¹⁵ should link to `advice/running-tips.html`.



- C. `tips-health.png`¹⁶ should link to `advice/health-tips.html`.



-
15. The <https://pixabay.com/illustrations/running-woman-man-together-family-2897357> image is used under the terms of Pixabay License (<https://pixabay.com/service/license/>).
 16. The <https://pixabay.com/illustrations/veggies-vegetables-healthy-food-2340299> image is used under the terms of Pixabay License (<https://pixabay.com/service/license/>).

- D. `facebook-icon.png`, `linkedin-icon.png`, and `twitter-icon.png`¹⁷ should replace the text used in the footer of the exercise file.



17. The <https://pixabay.com/de/illustrations/icon-social-media-linkedin-facebook-2083456> image is used under the terms of Pixabay License (<https://pixabay.com/service/license/>).

Solution: Images/Solutions/index.html

```
-----Lines 1 through 8 Omitted-----
9.  <a href="index.html">
10.    
11.  </a>
    -----Lines 12 through 26 Omitted-----
27. <h3>Best Running Tips</h3>
28. <a href="advice/running-tips.html">
29.    
30. </a>
31. <p>There are good ways to train and bad ways to train. To get
32.    the most out of your runs...
33.    <a href="advice/running-tips.html">Read more...</a>
34. </p>
35. <h3>Best Health Tips</h3>
36. <a href="advice/health-tips.html">
37.    
38. </a>
    -----Lines 39 through 53 Omitted-----
54. <div>
55.    <a href="mailto:info@runners-home.com">info@runners-home.com</a>
56.    <a href="https://www.facebook.com/webucator">
57.        </a>
58.    <a href="https://www.linkedin.com/companies/webucator">
59.        </a>
60.    <a href="https://twitter.com/webucator">
61.        </a>
62. </div>
    -----Lines 63 through 64 Omitted-----
```



5.3. Providing Alternative Images

All modern browsers support the newer *WebP* image type, which provides better quality and compression than PNG, JPEG, and GIF images. Internet Explorer, however, does not. That may not be an issue as Internet Explorer has a very small share of the market. However, if you do need to support Internet Explorer, that doesn't mean you cannot use WebP images. To do so, you will need to use the picture element, like so:

```
<picture>
  <source srcset="images/logo.webp" type="image/webp">
  
</picture>
```

Notice that the `picture` element has a child `source` element with a `srcset` attribute that points to the image file and a `type` attribute that indicates what type of image it is. If the browser supports that type of image, it will use that file. If not, it will fall back on the image file in the subsequent `img` element.

More on WebP

For more information on WebP and a free tool for converting PNG and JPG files to WebP, see <https://developers.google.com/speed/webp>.

Conclusion

In this lesson, you have learned to add images to a web page, to make those images accessible, to create image links, and to use the `picture` element to provide image fallbacks.

LESSON 6

HTML Lists

Topics Covered

- ☒ Unordered lists.
- ☒ Ordered lists.
- ☒ Definition lists.

Introduction

There are three types of lists in HTML: unordered, ordered, and definition lists. In this lesson, you will learn how to create all three.

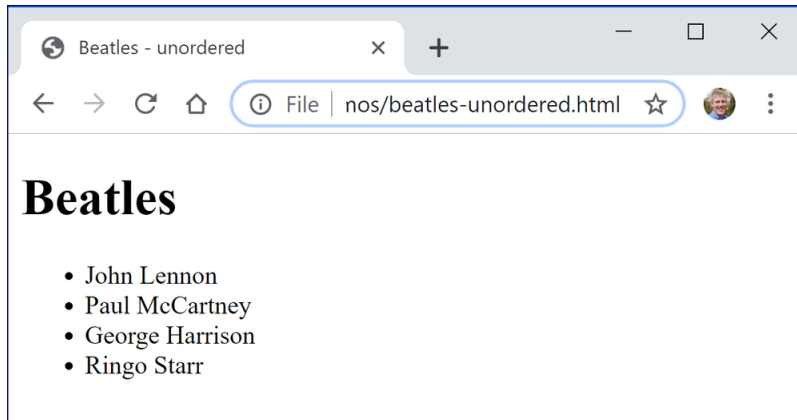
6.1. Unordered Lists

Unordered lists are rendered as bulleted lists. Take a look at the following code sample:

Demo 6.1: Lists/Demos/beatles-unordered.html

```
-----Lines 1 through 8 Omitted-----
9.      <h1>Beatles</h1>
10.     <ul>
11.         <li>John Lennon</li>
12.         <li>Paul McCartney</li>
13.         <li>George Harrison</li>
14.         <li>Ringo Starr</li>
15.     </ul>
-----Lines 16 through 17 Omitted-----
```

The `` tag starts an unordered list. Each list item is contained in `` tags. The following screenshot shows how this code would be rendered:



❖ 6.1.1. Nesting Unordered Lists

Unordered lists can also be nested. The browsers use indentation and different styles of bullets¹⁸ to display the nested lists. The following example shows how this works:

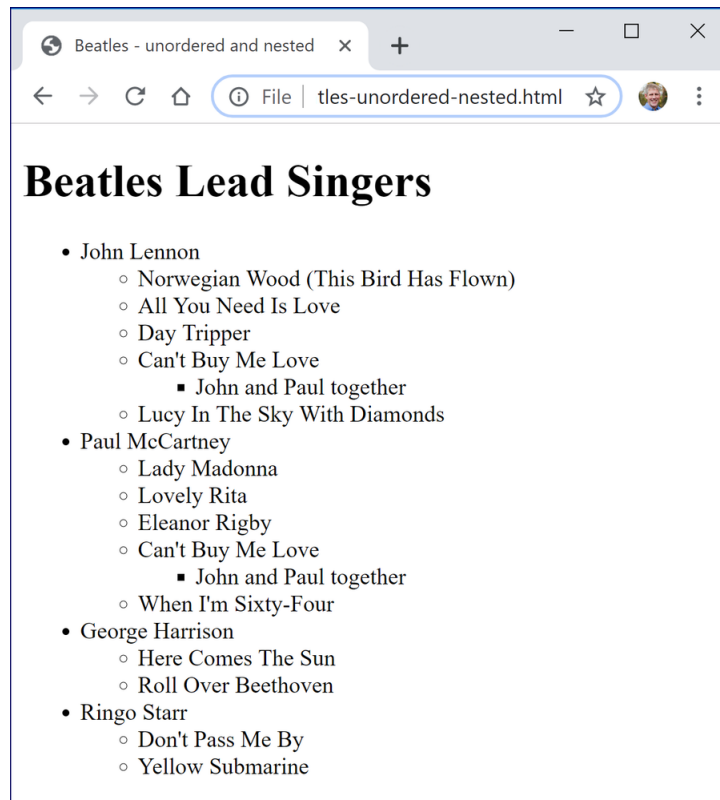
18. Both the indentation and the style of bullet can be controlled with CSS.

Demo 6.2: Lists/Demos/beatles-unordered-nested.html

```
-----Lines 1 through 8 Omitted-----
9.    <h1>Beatles Lead Singers</h1>
10.   <ul>
11.     <li>John Lennon
12.       <ul>
13.         <li>Norwegian Wood (This Bird Has Flown)</li>
14.         <li>All You Need Is Love</li>
15.         <li>Day Tripper</li>
16.         <li>Can't Buy Me Love
17.           <ul>
18.             <li>John and Paul together</li>
19.           </ul>
20.         </li>
21.         <li>Lucy In The Sky With Diamonds</li>
22.       </ul>
23.     </li>
24.     <li>Paul McCartney
25.       <ul>
26.         <li>Lady Madonna</li>
27.         <li>Lovely Rita</li>
28.         <li>Eleanor Rigby</li>
29.         <li>Can't Buy Me Love
30.           <ul>
31.             <li>John and Paul together</li>
32.           </ul>
33.         </li>
34.         <li>When I'm Sixty-Four</li>
35.       </ul>
36.     </li>
37.     <li>George Harrison
38.       <ul>
39.         <li>Here Comes The Sun</li>
40.         <li>Roll Over Beethoven</li>
41.       </ul>
42.     </li>
43.     <li>Ringo Starr
44.       <ul>
45.         <li>Don't Pass Me By</li>
46.         <li>Yellow Submarine</li>
47.       </ul>
48.     </li>
49.   </ul>
-----Lines 50 through 51 Omitted-----
```

Notice that the nested unordered lists are siblings to plain text. For example, the text “George Harrison” and the unordered list that follows that text both are contained within the same parent `` tag. Only list items, not lists themselves, can contain nested (i.e., child) lists. In other words, lists contain list items, which can contain lists, which contain list items, which can contain lists, which contain list items, and so on and so forth, ad infinitum.

Here is the resulting page:



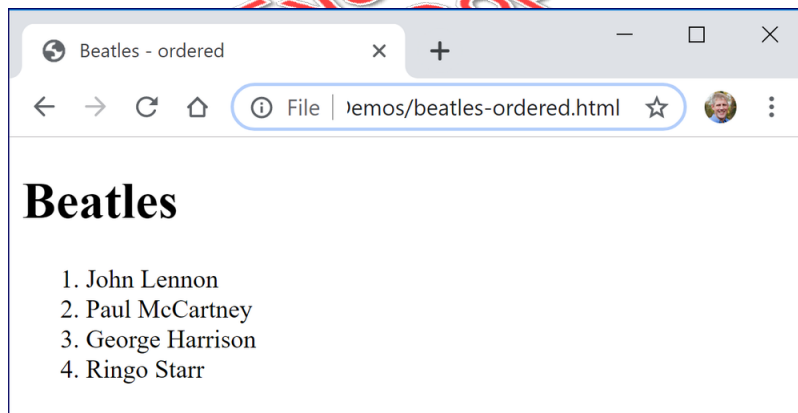
6.2. Ordered Lists

Ordered lists are similar to unordered lists. They are created with the `` tag and, by default, will display list items with numbers. Take a look at the following code:

Demo 6.3: Lists/Demos/beatles-ordered.html

```
-----Lines 1 through 8 Omitted-----
9.     <h1>Beatles</h1>
10.    <ol>
11.        <li>John Lennon</li>
12.        <li>Paul McCartney</li>
13.        <li>George Harrison</li>
14.        <li>Ringo Starr</li>
15.    </ol>
-----Lines 16 through 17 Omitted-----
```

The following screenshot shows how the code will be rendered:



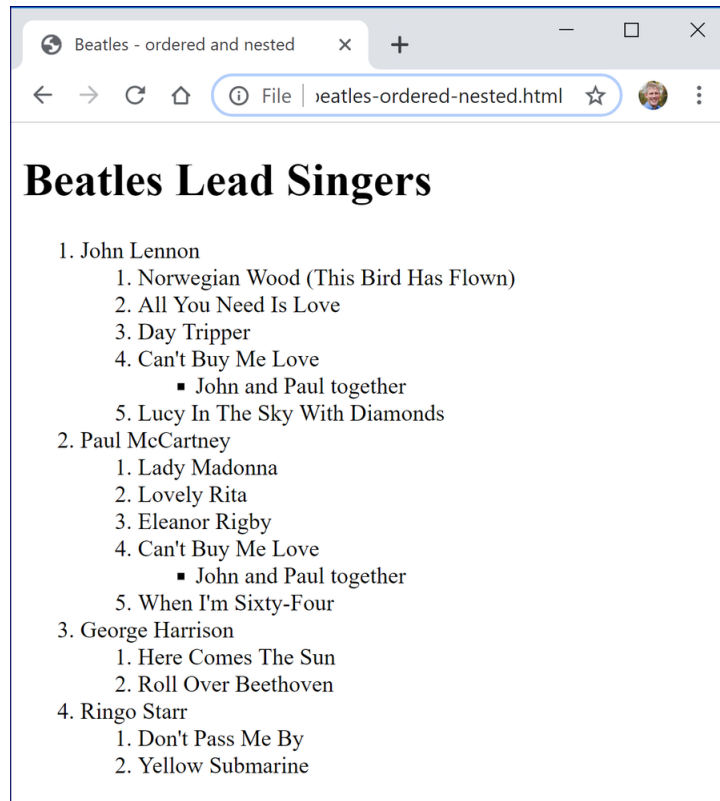
❖ 6.2.1. Nesting Ordered Lists

Like unordered lists, ordered lists can be nested. However, unlike in some word processing applications, nested ordered lists will continue to be displayed using standard numbers:

Demo 6.4: Lists/Demos/beatles-ordered-nested.html

```
-----Lines 1 through 8 Omitted-----
9.    <h1>Beatles Lead Singers</h1>
10.   <ol>
11.     <li>John Lennon
12.       <ol>
13.         <li>Norwegian Wood (This Bird Has Flown)</li>
14.         <li>All You Need Is Love</li>
15.         <li>Day Tripper</li>
16.         <li>Can't Buy Me Love
17.           <ul>
18.             <li>John and Paul together</li>
19.           </ul>
20.         </li>
21.         <li>Lucy In The Sky With Diamonds</li>
22.       </ol>
23.     </li>
24.     <li>Paul McCartney
25.       <ol>
26.         <li>Lady Madonna</li>
27.         <li>Lovely Rita</li>
28.         <li>Eleanor Rigby</li>
29.         <li>Can't Buy Me Love
30.           <ul>
31.             <li>John and Paul together</li>
32.           </ul>
33.         </li>
34.         <li>When I'm Sixty-Four</li>
35.       </ol>
36.     </li>
37.     <li>George Harrison
38.       <ol>
39.         <li>Here Comes The Sun</li>
40.         <li>Roll Over Beethoven</li>
41.       </ol>
42.     </li>
43.     <li>Ringo Starr
44.       <ol>
45.         <li>Don't Pass Me By</li>
46.         <li>Yellow Submarine</li>
47.       </ol>
48.     </li>
49.   </ol>
-----Lines 50 through 51 Omitted-----
```

The resulting page looks like this:



As you can see, ordered lists can contain nested unordered lists (below Can't Buy Me Love). The reverse is also true.

❖ 6.2.2. The type Attribute

The type attribute is used to change the numbering type. Possible values are shown in the following table:

Values of the type Attribute

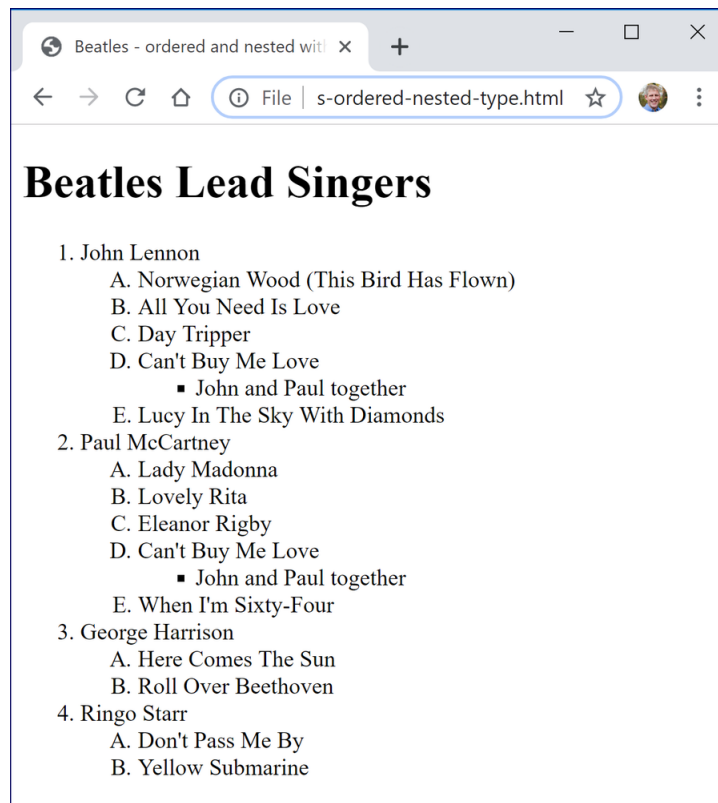
Value	Description
i	Lowercase Roman Numerals
I	Uppercase Roman Numerals
a	Lowercase Letters
A	Uppercase Letters
1	Numbers (default)

The following code illustrates how type is used:

Demo 6.5: Lists/Demos/beatles-ordered-nested-type.html

```
-----Lines 1 through 9 Omitted-----
10.  <ol>
11.    <li>John Lennon
12.      <ol type="A">
13.        <li>Norwegian Wood (This Bird Has Flown)</li>
14.        <li>All You Need Is Love</li>
15.        <li>Day Tripper</li>
16.        <li>Can't Buy Me Love
17.          <ul>
18.            <li>John and Paul together</li>
19.          </ul>
20.        </li>
21.        <li>Lucy In The Sky With Diamonds</li>
22.      </ol>
23.    </li>
24.    <li>Paul McCartney
25.      <ol type="A">
26.        <li>Lady Madonna</li>
27.        <li>Lovely Rita</li>
28.        <li>Eleanor Rigby</li>
29.        <li>Can't Buy Me Love
30.          <ul>
31.            <li>John and Paul together</li>
32.          </ul>
33.        </li>
34.        <li>When I'm Sixty-Four</li>
35.      </ol>
36.    </li>
37.    <li>George Harrison
38.      <ol type="A">
39.        <li>Here Comes The Sun</li>
40.        <li>Roll Over Beethoven</li>
41.      </ol>
42.    </li>
43.    <li>Ringo Starr
44.      <ol type="A">
45.        <li>Don't Pass Me By</li>
46.        <li>Yellow Submarine</li>
47.      </ol>
48.    </li>
49.  </ol>
-----Lines 50 through 51 Omitted-----
```

Here is the resulting page:



List Types and CSS

As a rule, it is better to set the type of numbering using the CSS `list-style-type` property. The exception is when the value of the list item is meaningful as it sometimes is in legal or technical documents. This is because you cannot be sure that CSS will be enabled.

Also note that the unordered tag (``) used to have a `type` attribute as well, but this has been deprecated in favor of CSS, so you should not use it.

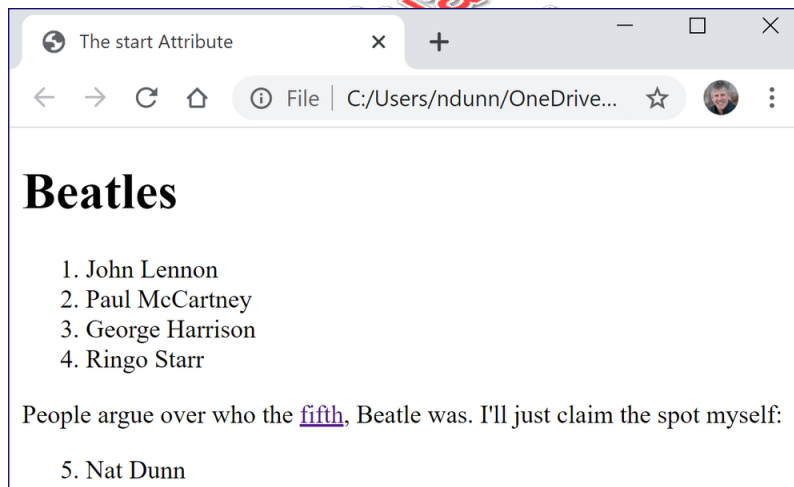
❖ 6.2.3. The `start` Attribute

The `start` attribute is used to specify what number the list should start on. It takes an integer value. For example:

Demo 6.6: Lists/Demos/fifth-beatle.html

```
-----Lines 1 through 9 Omitted-----
10.     <ol>
11.         <li>John Lennon</li>
12.         <li>Paul McCartney</li>
13.         <li>George Harrison</li>
14.         <li>Ringo Starr</li>
15.     </ol>
16.     <p>
17.         People argue over who the
18.         <a href="https://en.wikipedia.org/wiki/Fifth_Beatle">fifth</a>,
19.         Beatle was. I'll just claim the spot myself:
20.     </p>
21.     <ol start="5">
22.         <li>Nat Dunn</li>
23.     </ol>
-----Lines 24 through 25 Omitted-----
```

The following screenshot shows how the code will be rendered:



6.3. Definition Lists

Definition lists are used to define a list of terms. The following example is a modified version of an example from the W3C Recommendation:¹⁹

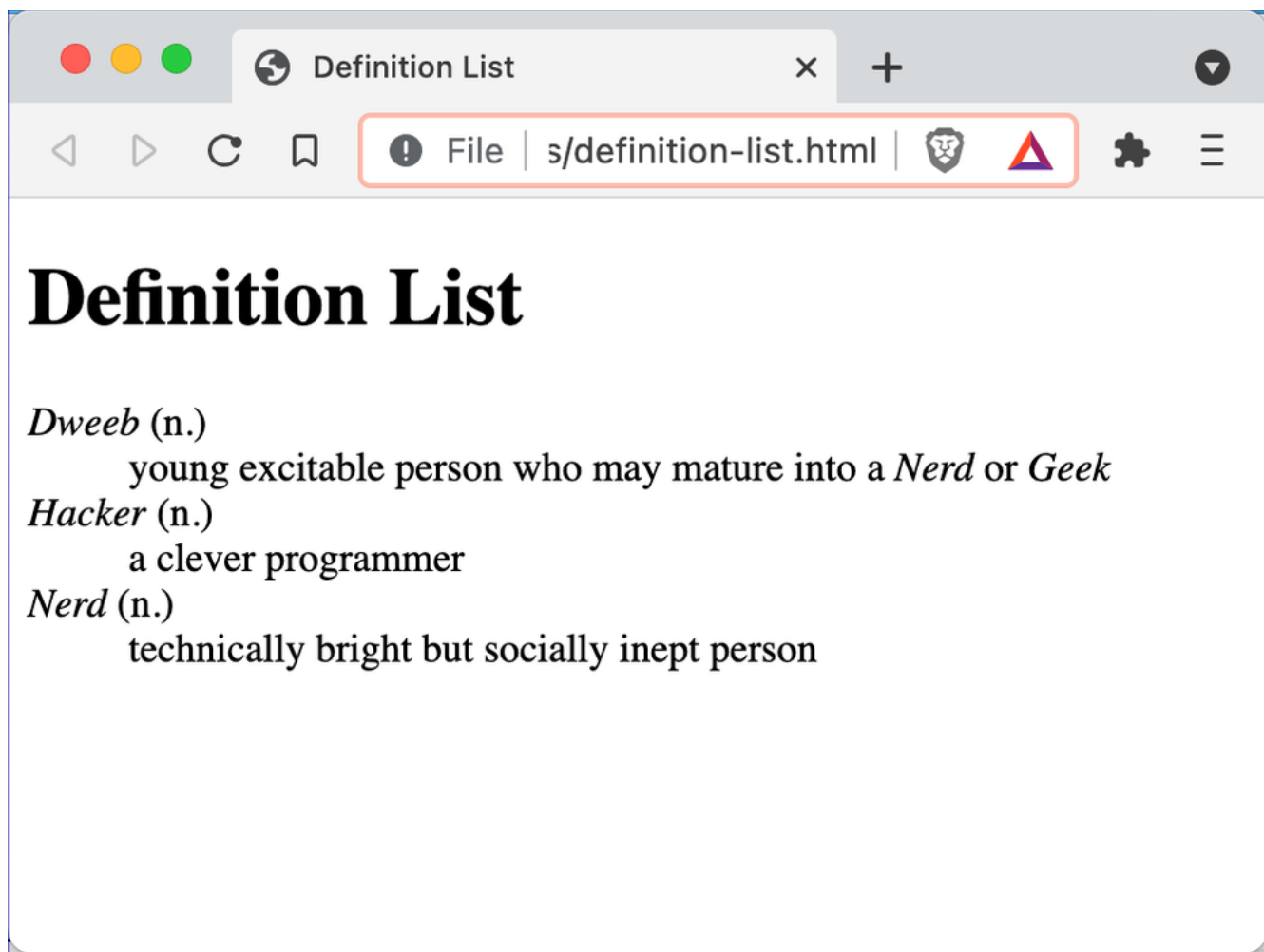
19. <https://www.w3.org/TR/html4/struct/lists.html#edef-DD>

Demo 6.7: Lists/Demos/definition-list.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. <title>Definition List</title>
7. </head>
8. <body>
9.   <h1>Definition List</h1>
10.  <dl>
11.    <dt><dfn>Dweeb</dfn> (n.)</dt>
12.    <dd>young excitable person who may mature into a
13.      <em>Nerd</em> or <em>Geek</em></dd>
14.    <dt><dfn>Hacker</dfn> (n.)</dt>
15.    <dd>a clever programmer</dd>
16.    <dt><dfn>Nerd</dfn> (n.)</dt>
17.    <dd>technically bright but socially inept person</dd>
18.  </dl>
19. </body>
20. </html>
```

1. The `<dl>` tag contains the **definition list**.
2. The `<dt>` tags contain the **definition terms**. Notice that these contain `<dfn></dfn>` tags, which are used to indicate the term being defined.
3. The `<dd>` tags contain the **definition descriptions**.

The following screenshot shows how the code will be rendered:





Exercise 6: Creating Lists

⌚ 40 to 60 minutes

In this exercise, you will add several lists to the **Runners Home** website.

Navigation Menu

On all the pages, you will update the main navigation at the top to look like this:

Runners Home

- [Home](#)
- [Resources](#)
 - [Calculator](#)
 - [Running Log](#)
 - [Running Terms](#)
 - [Links](#)
- [Running Advice](#)
- [Races](#)
- [Register](#)

Notice that there is a nested list under **Resources**.

In addition to the main navigation list, add the following lists:

1. `index.html` – Add an ordered list under **Purpose** on the home page:

Purpose

Runners Home is dedicated to providing you with:

1. [the most up-to-date information on running races.](#)
2. [the best resources for runners.](#)

2. `races.html` – Add an ordered list with nested unordered lists to the page:

Races

1. [AJC Peachtree Road Race](#)
 - **Date:** March 15th
 - **Distance:** 10K
 - **Location:** Atlanta, GA
2. [Boilermaker](#)
 - **Date:** July 11th
 - **Distance:** 15K
 - **Location:** Utica, NY
3. [NYC Marathon](#)
 - **Date:** November 3rd
 - **Distance:** 26.22 mi
 - **Location:** New York, NY

3. resources/index.html – Add an unordered list:

Resources

- [Calculator](#)
- [Running Log](#)
- [Running Terms](#)
- [Links](#)

4. resources/terms.html – Add a definition list to the page:

Running Terms

Aerobic

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam et gravida sapien, facilisis condimentum arcu. Morbi eget dui iaculis, porttitor eros et, tincidunt erat...

Bandit

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam et gravida sapien, facilisis condimentum arcu. Morbi eget dui iaculis, porttitor eros et, tincidunt erat...

C25K

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam et gravida sapien, facilisis condimentum arcu. Morbi eget dui iaculis, porttitor eros et, tincidunt erat...

DNE

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam et gravida sapien, facilisis condimentum arcu. Morbi eget dui iaculis, porttitor eros et, tincidunt erat...

Easy Run

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam et gravida sapien, facilisis condimentum arcu. Morbi eget dui iaculis, porttitor eros et, tincidunt erat...

5. resources/links.html – Add an unordered list to the page:

Useful Links

- [Map My Run](#)
- [Runners World](#)
- [Strength Running](#)

6. advice/index.html – Add an unordered list to the page:

Running Advice

- [Running Tips](#)
- [Health Tips](#)

Solution: Lists/Solutions/index.html

```
-----Lines 1 through 11 Omitted-----
12. <h1>Runners Home</h1>
13. <ul>
14.   <li><a href="index.html">Home</a></li>
15.   <li><a href="resources/index.html">Resources</a>
16.     <ul>
17.       <li><a href="resources/calculator.html">Calculator</a></li>
18.       <li><a href="resources/run-log.html">Running Log</a></li>
19.       <li><a href="resources/terms.html">Running Terms</a></li>
20.       <li><a href="resources/links.html">Links</a></li>
21.     </ul>
22.   </li>
23.   <li><a href="advice/index.html">Running Advice</a></li>
24.   <li><a href="races.html">Races</a></li>
25.   <li><a href="register.html">Register</a></li>
26. </ul>
-----Lines 27 through 46 Omitted-----
47. <h2>Purpose</h2>
48. <p>Runners Home is dedicated to providing you with:</p>
49. <ol>
50.   <li><a href="races.html">the most up-to-date
51.     information on running races</a>.</li>
52.   <li><a href="resources/index.html">the best
53.     resources for runners</a>.</li>
54. </ol>
-----Lines 55 through 68 Omitted-----
```

The main navigation list should be included on all the pages. That is not shown in the solutions below.

Solution: Lists/Solutions/races.html

```
-----Lines 1 through 26 Omitted-----
27. <h2>Races</h2>
28. <ol>
29.   <li>
30.     <a href="https://www.atlantatrackclub.org/peachtree">
31.       AJC Peachtree Road Race
32.     </a>
33.     <ul>
34.       <li><strong>Date</strong>: March 15th</li>
35.       <li><strong>Distance</strong>: 10K</li>
36.       <li><strong>Location</strong>: Atlanta, GA</li>
37.     </ul>
38.   </li>
39.   <li>
40.     <a href="https://www.boilermaker.com">Boilermaker</a>
41.     <ul>
42.       <li><strong>Date</strong>: July 11th</li>
43.       <li><strong>Distance</strong>: 15K</li>
44.       <li><strong>Location</strong>: Utica, NY</li>
45.     </ul>
46.   </li>
47.   <li>
48.     <a href="https://www.nyrr.org/">NYC Marathon</a>
49.     <ul>
50.       <li><strong>Date</strong>: November 3rd</li>
51.       <li><strong>Distance</strong>: 26.22 mi</li>
52.       <li><strong>Location</strong>: New York, NY</li>
53.     </ul>
54.   </li>
55. </ol>
-----Lines 56 through 69 Omitted-----
```

Solution: Lists/Solutions/resources/index.html

```
-----Lines 1 through 26 Omitted-----
27. <h2>Resources</h2>
28. <ul>
29.   <li><a href="calculator.html">Calculator</a></li>
30.   <li><a href="run-log.html">Running Log</a></li>
31.   <li><a href="terms.html">Running Terms</a></li>
32.   <li><a href="links.html">Links</a></li>
33. </ul>
-----Lines 34 through 47 Omitted-----
```

Solution: Lists/Solutions/resources/terms.html

```
-----Lines 1 through 26 Omitted-----
27. <h2>Running Terms</h2>
28. <dl>
29.   <dt id="aerobic">Aerobic</dt>
30.   <dd>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
31.     Aliquam et gravida sapien, facilisis condimentum arcu.
32.     Morbi eget dui iaculis, porttitor eros et, tincidunt erat...</dd>
33.
34.   <dt id="bandit">Bandit</dt>
35.   <dd>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
36.     Aliquam et gravida sapien, facilisis condimentum arcu.
37.     Morbi eget dui iaculis, porttitor eros et, tincidunt erat...</dd>
38.
39.   <dt id="C25K"><abbr title="couch to 5K">C25K</abbr></dt>
40.   <dd>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
41.     Aliquam et gravida sapien, facilisis condimentum arcu.
42.     Morbi eget dui iaculis, porttitor eros et, tincidunt erat...</dd>
43.
44.   <dt id="dnf"><abbr title="Did not finish">DNF</abbr></dt>
45.   <dd>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
46.     Aliquam et gravida sapien, facilisis condimentum arcu.
47.     Morbi eget dui iaculis, porttitor eros et, tincidunt erat...</dd>
48.
49.   <dt id="easy-run">Easy Run</dt>
50.   <dd>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
51.     Aliquam et gravida sapien, facilisis condimentum arcu.
52.     Morbi eget dui iaculis, porttitor eros et, tincidunt erat...</dd>
53. </dl>
-----Lines 54 through 68 Omitted-----
```

Solution: Lists/Solutions/resources/links.html

```
-----Lines 1 through 26 Omitted-----
27. <h2>Useful Links</h2>
28. <ul>
29.   <li><a href="https://www.mapmyrun.com/">Map My Run</a></li>
30.   <li><a href="https://www.runnersworld.com">Runners World</a></li>
31.   <li><a href="https://strengthrunning.com/">Strength Running</a></li>
32. </ul>
-----Lines 33 through 47 Omitted-----
```

Solution: Lists/Solutions/advice/index.html

```
-----Lines 1 through 26 Omitted-----
27. <h2>Running Advice</h2>
28. <ul>
29.   <li><a href="running-tips.html">Running Tips</a></li>
30.   <li><a href="health-tips.html">Health Tips</a></li>
31. </ul>
-----Lines 32 through 45 Omitted-----
```

Conclusion

In this lesson, you have learned to create unordered, ordered and definition lists.

LESSON 7

Sectioning a Web Page

Topics Covered

- ☒ Separating page content into semantic sections.

Introduction

You have already worked with `div` elements to break out a block of content. But the `<div>` tag doesn't tell us anything about the content it contains. In this lesson, you will learn about *semantic* (i.e., logical and meaningful) block-level elements.



7.1. Semantic Block-Level Elements

Remember that HTML is used to mark up content in a meaningful way. We learned earlier about inline semantic elements (see page 37) like `strong` and `em`. HTML block-level semantic elements are listed below:

1. `address` – Contains contact information.
2. `article` – Contains a self-contained article (e.g., for a blog or news site) that is meant to be able to stand on its own.
3. `aside` – Contains content that is not directly related to the page's main content.
4. `div` – Contains a meaningless block of content. Content within a `<div>` tag is not necessarily related to other content in the same tag. Like the inline `` tag, the `<div>` tag can be used for grouping elements, and meaning and style can be added through adding attributes (e.g., `id`, `class`, and `lang`).
5. `footer` – Contains the footer of the page. Often used for the company's address or a copyright message or legalese. Other page elements (e.g., `article`, `aside`) can contain their own footers.
6. `header` – Contains the header of the page. Often used for page headings, possibly some introductory content, and to hold the main navigation of the page. Other page elements (e.g., `article`, `aside`) can contain their own headers.

7. `main` – Contains the dominant content of the page. You should have only one `main` element on a page.
8. `nav` – Contains page or site navigation.
9. `section` – Contains a section of the page that doesn't have any other tag that more meaningfully describes the contained content. All content within a `<section>` tag should be related.

The elements above are all block-level elements with no default formatting. Unless they are styled with CSS, there will be no indication that they are included on the page.²⁰ But the purpose of these elements is not to style the page; it is to provide meaningful structure. They make it easier for screen readers and other devices that inspect or read the page to understand the page content.



7.2. Articles vs. Sections

Most of the sectioning elements are easily distinguishable from each other. You're not likely to be confused as to when to use the header or footer elements. But the difference between a `section` and an `article` is more subtle.

The major difference between the `article` element and the `section` element is that an `article` element encapsulates content that could stand alone and might be of interest outside the context of the page. Mozilla²¹ puts it this way:

The `article` element “represents a self-contained composition in a document, page, application, or site, which is intended to be independently distributable or reusable (e.g., in syndication).”

The most obvious example is a blog entry or a newspaper article; however, you could also apply this to a product description as you can imagine wanting to syndicate all of your products to a website that aggregates product information from different companies.

A `section`, on the other hand, is used to encapsulate content that fits together in some identifiable way. One test for whether content should go together in a `section` is to see if you can give that `section` a meaningful `id`, using the `id` attribute.

^{20.} The only exception is `address`, which is italic by default.

^{21.} <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/article>

Note that section elements can contain article elements, and article elements can contain section elements. Both section and article elements can also contain header and footer elements.

Evaluation
Copy

7.3. Sectioning the Home Page

In the following code, we show how to use these tags to break up the **Runners Home** home page. We have added a newsletter sign-up link to show how to use the `aside` element.

Demo 7.1: SectioningPage/Demos/index.html

```
-----Lines 1 through 7 Omitted-----
8.  <body>
9.  <header>
10.    <a href="index.html">
11.      
12.    </a>
13.    <h1>Runners Home</h1>
14.    <nav>
15.      <ul>
16.        <li><a href="index.html">Home</a></li>
17.        <li><a href="resources/index.html">Resources</a>
18.          <ul>
19.            <li><a href="resources/calculator.html">Calculator</a></li>
20.            <li><a href="resources/run-log.html">Running Log</a></li>
21.            <li><a href="resources/terms.html">Running Terms</a></li>
22.            <li><a href="resources/links.html">Links</a></li>
23.          </ul>
24.        </li>
25.        <li><a href="advice/index.html">Running Advice</a></li>
26.        <li><a href="races.html">Races</a></li>
27.        <li><a href="register.html">Register</a></li>
28.      </ul>
29.    </nav>
30.  </header>
31.  <main>
32.    <section id="welcome">
33.      <p><strong>Hello, Stranger!</strong></p>
34.      <p>Welcome to Runners Home</p>
35.    </section>
36.    <section id="advice">
37.      <h2>Advice</h2>
38.      <article id="running-tips">
39.        <h3>Best Running Tips</h3>
40.        <a href="advice/running-tips.html">
41.          
42.        </a>
43.        <p>There are good ways to train and bad ways to train. To get
44.          the most out of your runs...
45.          <a href="advice/running-tips.html">Read more...</a>
46.        </p>
47.      </article>
48.      <article id="health-tips">
49.        <h3>Best Health Tips</h3>
50.        <a href="advice/health-tips.html">
```

```

51.         
52.     </a>
53.     <p>In addition to keeping up with your physical training, you
54.         must be sure to take care of your general health...
55.         <a href="advice/health-tips.html">Read more...</a>
56.     </p>
57. </article>
58. <p><a href="advice/index.html">More advice articles...</a></p>
59. </section>
60. <section id="purpose">
61.     <h2>Purpose</h2>
62.     <p>Runners Home is dedicated to providing you with:</p>
63.     <ol>
64.         <li><a href="races.html">the most up-to-date
65.             information on running races</a>.</li>
66.         <li><a href="resources/index.html">the best
67.             resources for runners</a>.</li>
68.     </ol>
69.     <aside>
70.         <h3>Newsletter</h3>
71.         <p>Be the first to hear about our great offers.<br>
72.             <a href="newsletter.html">Sign up for our newsletter today!</a>
73.         </p>
74.     </aside>
75.     <p><small>Disclosure: This is not a real website.</small></p>
76. </section>
77. </main>
78. <footer>
79.     <p>&copy; 2022 Runners Home. All rights reserved.</p>
80.     <address>
81.         <a href="mailto:info@runners-home.com">info@runners-home.com</a>
82.         <a href="https://www.facebook.com/webucator">
83.             </a>
84.         <a href="https://www.linkedin.com/companies/webucator">
85.             </a>
86.         <a href="https://twitter.com/webucator">
87.             </a>
88.     </address>
89. </footer>
90. </body>
91. </html>

```

Things to notice:

1. We broke the body into three parts: header, main, and footer.

2. We removed the `<hr>` tag after the disclosure paragraph. The purpose of the `<hr>` tag is to provide a thematic break (e.g., between stanzas in a poem) rather than a visual break. Now that we are breaking up the page semantically, we no longer need the `<hr>` tag.
3. In the header, we added a `nav` element to hold the navigation.
4. The `main` element holds all of the main content on the page. Typically, that is the content that is specific to this page; whereas, the header and footer are more likely to contain content that is relevant to the whole website, or at least a larger part of it.
5. In the footer, we placed the contact information in an `address` element.
6. The `main` element is broken into `section` elements to separate structurally meaningful parts of the page. We have used the `id` attribute to uniquely identify these sections as “welcome”, “advice”, and “purpose.” While this is helpful in and of itself as a way of making the code clearer, the larger benefit is that it makes these sections easily identifiable when it’s time to style the page with CSS.
7. We added an `aside` element to hold the newsletter aside.

The following screenshot shows the difference between the file before and after sectioning. This was created by comparing `index-unsectioned.html` with `index.html` from the `SectioningPage/Demos` folder using Visual Studio Code’s **Compare Selected** feature:

```

11 <body>
12 <header>
13   <a href="index.html">
14     
15   </a>
16   <h1>Runners Home</h1>
17   <nav>
18     <ul>
19       <li><a href="index.html">Home</a></li>
20       <li><a href="resources/index.html">Resources</a>
21     </ul>
22     <li><a href="resources/calculator.html">Calculator</a></li>
23     <li><a href="resources/run-log.html">Running Log</a></li>
24     <li><a href="resources/terms.html">Running Terms</a></li>
25     <li><a href="resources/links.html">Links</a></li>
26   </ul>
27 </li>
28 <li><a href="advice/index.html">Running Advice</a></li>
29 <li><a href="races.html">Races</a></li>
30 <li><a href="register.html">Register</a></li>
31 </ul>
32 </nav>
33 </header>
34 <main>
35   <section id="welcome">
36     <p><strong>Hello, Stranger!</strong></p>
37     <p>Welcome to Runners Home</p>
38   </section>
39   <section id="advice">
40     <h2>Advice</h2>
41     <article id="running-tips">
42       <h3>Best Running Tips</h3>
43       <a href="advice/running-tips.html">
44         
45       </a>
46       <p>There are good ways to train and bad ways to train. To get
47         the most out of your runs...
48       <a href="advice/running-tips.html">Read more...</a>
49     </p>
50   </article>

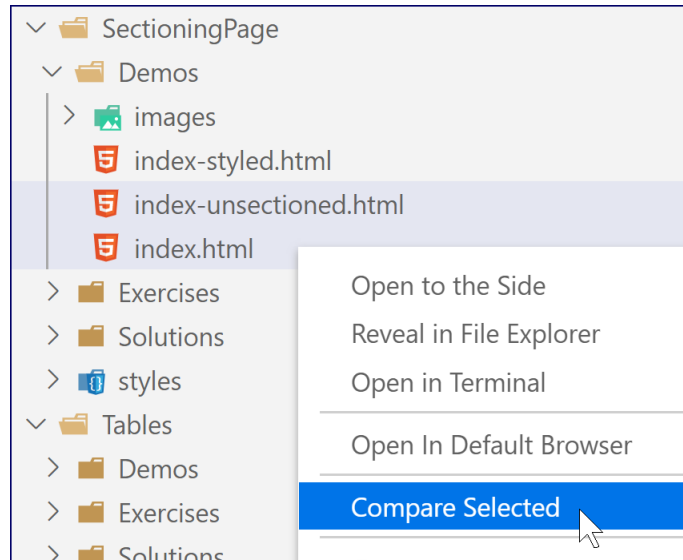
```

```

51 <article id="health-tips">
52   <h3>Best Health Tips</h3>
53   <a href="advice/health-tips.html">
54     
55   </a>
56   <p>In addition to keeping up with your physical training, you
57   must be sure to take care of your general health...
58   <a href="advice/health-tips.html">Read more...</a>
59 </p>
60 </article>
61 <p><a href="advice/index.html">More advice articles...</a></p>
62 </section>
63 <section id="purpose">
64   <h2>Purpose</h2>
65   <p>Runners Home is dedicated to providing you with:</p>
66   <ol>
67     <li><a href="races.html">the most up-to-date
68     information on running races</a>.</li>
69     <li><a href="resources/index.html">the best
70     resources for runners</a>.</li>
71   </ol>
72   <aside>
73     <h3>Newsletter</h3>
74     <p>Be the first to hear about our great offers.<br>
75     <a href="newsletter.html">Sign up for our newsletter today!</a>
76   </p>
77   </aside>
78   <p><small>Disclosure: This is not a real website.</small></p>
79 </section>
80 </main>
81 <footer>
82   <p>&copy; 2022 Runners Home. All rights reserved.</p>
83   <address>
84     <a href="mailto:info@runners-home.com">info@runners-home.com</a>
85     <a href="https://www.facebook.com/webucator">
86       </a>
87     <a href="https://www.linkedin.com/companies/webucator">
88       </a>
89     <a href="https://twitter.com/webucator">
90       </a>
91   </address>
92 </footer>
93 </body>

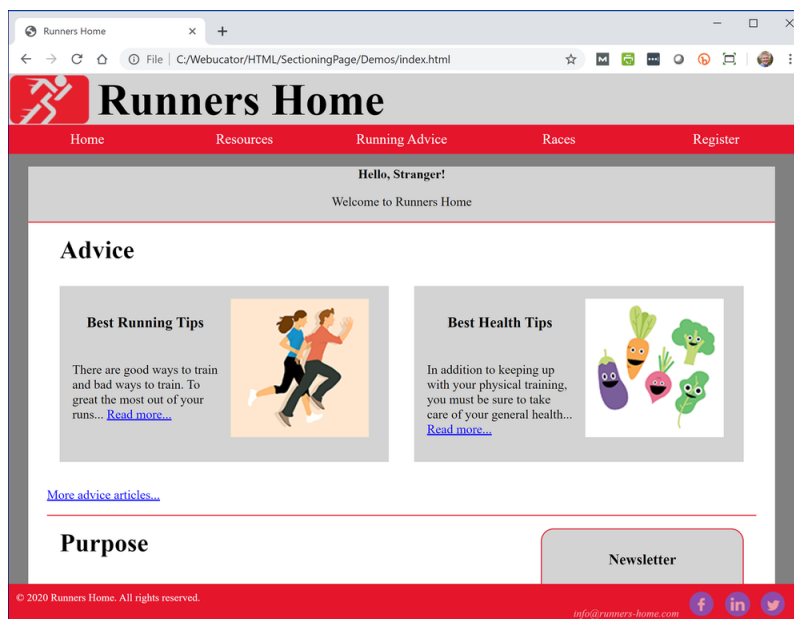
```

To see this comparison side-by-side with the unsectioned version, select the two files in Visual Studio Code's **Explorer**, right-click, and select **Compare Selected**:



7.4. Sectioning Content and Styling

Because these semantic elements don't have default styles, adding them will not change the way the page appears. As such, we have used `<link>` tags in the head to reference CSS files to style the page. This should help you to get a feel for what can be done with CSS on a well-structured page:



From SectioningPage/Demos, open index.html and index-styled.html in your browser to compare the unstyled and styled versions.

7.4. Heading Levels and Sectioning Elements

Our recommended approach to using heading levels in sectioned content is as follows:

1. Do not put any content directly in the body. Break the body up into header, main, and footer elements.
2. In the header, use an h1 heading for the heading of the entire page. Do not use any other h1 headings on the page.
3. Within main, section, article, aside, nav, and address elements, headings should start at the h2 level and then decrease in level with each nested element that contains a heading. To illustrate, consider our main element on the Runners Home home page. A simplified version follows. Notice that the heading levels decrease with each nested level:

```
<main>
  <section id="advice">
    <h2>Advice</h2>
    <article>
      <h3>Best Running Tips</h3>
    </article>
    <article>
      <h3>Best Health Tips</h3>
    </article>
  </section>
  <section id="purpose">
    <h2>Purpose</h2>
    <aside>
      <h3>Newsletter</h3>
    </aside>
  </section>
</main>
```

❖ 7.4.1. For People Who Like History

Guidance for selecting heading levels within nested elements has changed over the years and, as a result, it is done differently on many websites. If you are interested in the reasoning behind the current recommendation, see <https://www.webucator.com/article/html-heading-levels-and-sectioning-content/>.

Exercise 7: Sectioning the Pages

🕒 25 to 40 minutes

In this exercise, you will break up the content of the rest of the pages on the site using semantic elements. Open each page in the `SectioningPage/Exercises` directory and use sectioning tags to break up the body. Note that the `index.html` page is already done. You can use that as a reference.

The pages have references to CSS files included so that they will be nicely formatted when you're through.

Solution

Open the pages in the `SectioningPage/Solutions` directory to see the solutions.

You can compare your solutions with ours using Visual Studio Code's [Compare Selected](#) feature.

Conclusion

In this lesson, you have learned to work with semantic structural elements.

LESSON 8

Crash Course in CSS

Topics Covered

- ☒ Benefits of Cascading Style Sheets.
- ☒ Redefining how elements are formatted.
- ☒ CSS selectors.
- ☒ CSS combinators.
- ☒ The CSS cascade.
- ☒ CSS resets and normalizers.
- ☒ External stylesheets, embedded stylesheets, and inline styles.
- ☒ The `div` and `span` elements.
- ☒ Media types.
- ☒ The viewport `<meta>` tag.
- ☒ Units of measurement.
- ☒ How browsers style pages.
- ☒ Inheritance.

Introduction

Since HTML 4.0, most HTML formatting elements (e.g., `center` for centering content) and attributes (e.g., `bgcolor` for background color) have been deprecated, meaning that, although they may still be supported by browsers, the WHATWG (the maintainer of the HTML specification) recommends that they no longer be used. Web designers are to use CSS instead. In this lesson, you will get a high-level overview of CSS.



8.1. Benefits of Cascading Style Sheets

The major benefits of CSS are:

1. Cleaner Code
 - Easier to maintain.
 - Speedier download.
 - Better for search engine optimization.
2. Modular Code
 - Style rules can be applied to multiple pages.
 - Consistency of design.
 - Easier to maintain.
3. Design Power
 - Precise control of position, size, margins, etc.
4. Division of Labor
 - Developers develop / Designers design.
5. Better Accessibility²²
 - No need to misuse tags (e.g., <blockquote> for formatting).
 - No need for invisible images for positioning.
 - Users' stylesheets override authors' styles.



8.2. CSS Rules

CSS *rules* are statements that define the style of an element or group of elements. The syntax is as follows:

22. See <https://www.w3.org/TR/CSS-access>.


```
selector {  
  property: value;  
  property: value;  
  property: value;  
}
```

Each *property: value* pair is a *declaration*. Multiple declarations are separated by semicolons. The *selector* defines which elements are affected by the rule. Take a look at the following rule:

```
p {  
  color: darkgreen;  
  font-family: Verdana;  
  font-size: 1.5em;  
}
```

This rule specifies that all paragraph text should be darkgreen and use a 1.5em Verdana font. That's one and a half times the size that the font would normally be. You will learn about units of measurement soon.

❖ 8.2.1. CSS Comments

Comments in CSS begin with “/*” and end with “*/”. See the example below:

```
p {  
  color: red; /* All paragraphs should be red */  
}
```



8.3. Selectors

Selectors identify the element(s) affected by the CSS rule. There are several types of selectors:

1. Type
2. Class
3. ID
4. Attribute
5. Universal

In this section, we will give a high-level explanation of each of these types of selectors. It is a lot of information all at once. You should read through this section slowly and carefully. While it is important that you have an understanding of the different types of selectors, you do not need to commit the syntax for each type to memory. That will happen over time as you use them in practice.

❖ 8.3.1. Type Selectors

Type selectors specify elements by tag name and affect every instance of that element type. Looking again at the previous example:

```
p {  
  color: darkgreen;  
  font-family: Verdana;  
  font-size: 1.5em;  
}
```

Again, this rule specifies that the text of **every** p element should be darkgreen and use a 1.5em Verdana font.

❖ 8.3.2. Class Selectors

In HTML, all elements can take the `class` attribute, which is used to assign one or more class names 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 {  
  color: red;  
  font-weight: bold;  
}
```

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” will not be:

```
h1.warning {
  color: red;
  text-decoration: underline;
}

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

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

```
.warning {
  color: red;
}

h1.warning {
  text-decoration: underline;
}

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

Evaluation
Copy

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">...
```

❖ 8.3.3. ID Selectors

As with the `class` attribute, in HTML, 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 number sign (`#`) and have arbitrary names. The following rule will give the element with the “main-text” id a margin of 1.2em on the left and right:

```
#main-text {  
  margin-left: 1.2em;  
  margin-right: 1.2em;  
}
```

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

❖ 8.3.4. Attribute Selectors

Attribute selectors specify elements that contain a specific attribute. They can also specify what the value of that attribute should or should not be.

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

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

The `=` (equals) operator can be used to specify the attribute value. The following selector would only affect links whose `target` attribute is exactly `“_blank”`:

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

You can get much more specific about attribute values...

The `^=` (starts-with) operator can be used to specify the beginning text of the attribute value. The following selector will only affect links whose `href` attribute starts with `“mailto”`:

```
a[href^='mailto'] {  
  color: red;  
}
```

The `$=` (ends-with) operator can be used to specify the ending text of the attribute value. The following selector will only affect links whose `class` attribute ends with `“link”`:

```
a[class$='link'] {  
  color: red;  
}
```

The `*=` (contains) operator can be used to specify text the attribute value must contain. The following selector will only affect links whose `class` attribute contains “top”:

```
a[class*='top'] {  
  color: red;  
}
```

❖ 8.3.5. The Universal Selector

The universal selector is an asterisk (*). It matches every element:

```
* {  
  color: red;  
}
```

Evaluation
Copy

❖ 8.3.6. Grouping

Selectors can share the same declarations by separating them with commas. The following rule will underline all `em` elements, all elements of the class “warning” and the element with the `id` of “important”:

```
em,  
.warning,  
#important {  
  text-decoration: underline;  
}
```



8.4. Combinators

Combinators allow for the selection of elements based on the relationships between selectors.

There are several types of combinators:

1. Descendant
2. Child
3. General sibling
4. Adjacent sibling

❖ 8.4.1. Descendant Combinators

Descendant combinators 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.

❖ 8.4.2. Child Combinators

Child combinators specify a direct parent-child relationship. They are indicated by placing a `>` sign between the two element names:

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

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

❖ 8.4.3. General Sibling Combinators

General sibling combinators specify a sibling relationship²³ between two elements where the second element specified comes after the first. They are indicated by placing a `~` sign between the two element names:

23. Sibling elements have the same parent element.

```
em ~ strong {  
  color: red;  
}
```

In this case only `strong` elements that are siblings of and follow `em` elements are affected. For example, in the following code: both `strong` elements would be red:

```
<em>Hello!</em>  
<strong>Hi there!</strong>  
I'm not in a strong or em element.  
<strong>Howdy!</strong>
```

❖ 8.4.4. Adjacent Sibling Combinators

Adjacent sibling combinators specify a sibling relationship between two elements where the second element specified comes *immediately* after the first. They are indicated by placing a `+` sign between the two element names:

```
em + strong {  
  color: red;  
}
```

In this case only `strong` elements that are siblings of and immediately follow `em` elements are affected. In the HTML code just shown above, only the first `strong` element (with the text “Hi there!”) would be red, because it immediately follows an `em` element.



8.5. Precedence of Selectors

In the event that rules conflict:

- The rule with the more specific selector takes precedence.
- If two selectors have the same specificity, the rule specified later in the document takes precedence.

!important

Adding !important to any CSS declaration will give that declaration the highest specificity, but using !important is bad practice, and should be avoided.²⁴

Evaluation
*
Copy

8.6. How Browsers Style Pages

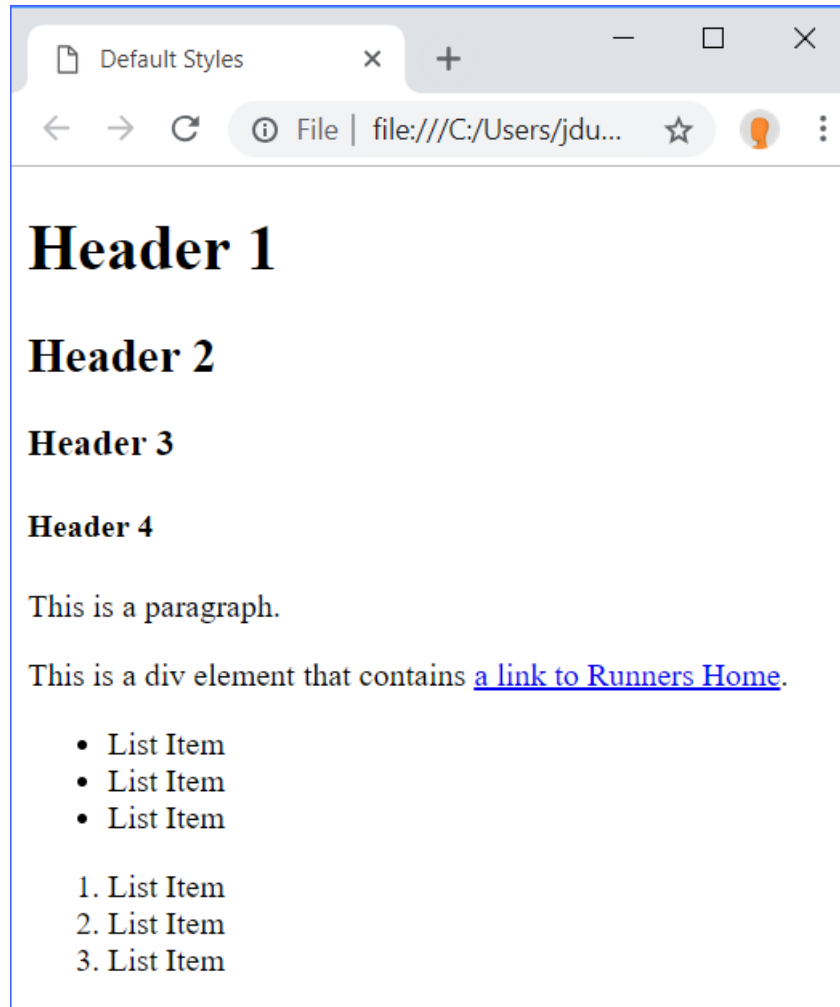
Browsers have built-in styles to make web pages readable by default (i.e., without CSS). Without any styles, an HTML page would just be one big line of text. For example, look at the code below:

²⁴. See https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity#the_!important_exception.

Demo 8.1: CrashCourse/Demos/default-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.  <title>Default Styles</title>
7.  </head>
8.  <body>
9.    <h1>Header 1</h1>
10.   <h2>Header 2</h2>
11.   <h3>Header 3</h3>
12.   <h4>Header 4</h4>
13.   <p>This is a paragraph.</p>
14.   <div>This is a div element that contains
15.     <a href="https://www.runners-home.com">a link to Runners Home</a>.
16.   </div>
17.   <ul>
18.     <li>List Item</li>
19.     <li>List Item</li>
20.     <li>List Item</li>
21.   </ul>
22.   <ol>
23.     <li>List Item</li>
24.     <li>List Item</li>
25.     <li>List Item</li>
26.   </ol>
27. </body>
28. </html>
```

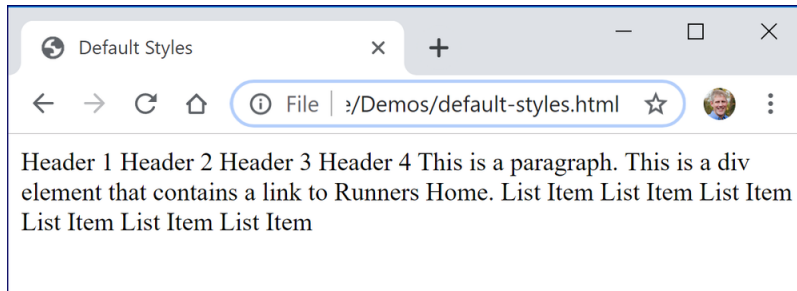
To make pages readable by default, the browsers add styles to render the page like this:



Notice the default styles:

1. The headings are bold, have varying font sizes, and have margin before and after them.
2. All of the elements, except the link, are on their own blocks as if they had `
` tags before and after them.
3. The unordered and ordered lists come with bullets and numbers, respectively.
4. The link is blue and underlined.

If browsers did not have default styles, they would display all the text in the body along a single line like this:



As that's not very readable, it's a good thing that browsers do provide default styles. Unfortunately, not all browsers provide the same default styles for all elements, so it's common practice for web designers to remove or override the browsers' default styles with default styles of their own.

Evaluation
Copy

8.7. CSS Resets

CSS resets are used to remove most default styles that browsers add to HTML elements, so that designers can decide on these styles themselves. For example, after applying a CSS reset, header tags will no longer be big and bold. By resetting the default styles, the designer gets rid of inconsistencies between browser default styles, opening the door for them to add styles that are rendered the same by all browsers.

A common CSS reset is the Meyer Reset²⁵, which you can see below:

25. <https://meyerweb.com/eric/tools/css/reset/>

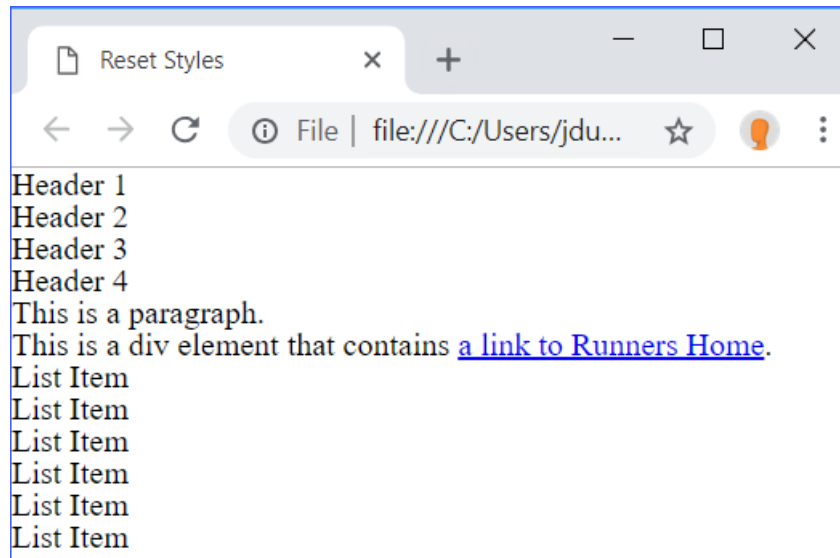
Demo 8.2: CrashCourse/Demos/reset.css

```
1.  /* http://meyerweb.com/eric/tools/css/reset/
2.      v2.0 | 20110126
3.      License: none (public domain)
4.  */
5.
6.  html, body, div, span, applet, object, iframe,
7.  h1, h2, h3, h4, h5, h6, p, blockquote, pre,
8.  a, abbr, acronym, address, big, cite, code,
9.  del, dfn, em, img, ins, kbd, q, s, samp,
10. small, strike, strong, sub, sup, tt, var,
11. b, u, i, center,
12. dl, dt, dd, ol, ul, li,
13. fieldset, form, label, legend,
14. table, caption, tbody, tfoot, thead, tr, th, td,
15. article, aside, canvas, details, embed,
16. figure, figcaption, footer, header, hgroup,
17. menu, nav, output, ruby, section, summary,
18. time, mark, audio, video {
19.     margin: 0;
20.     padding: 0;
21.     border: 0;
22.     font-size: 100%;
23.     font: inherit;
24.     vertical-align: baseline;
25. }
26. /* HTML5 display-role reset for older browsers */
27. article, aside, details, figcaption, figure,
28. footer, header, hgroup, menu, nav, section {
29.     display: block;
30. }
31. body {
32.     line-height: 1;
33. }
34. ol, ul {
35.     list-style: none;
36. }
37. blockquote, q {
38.     quotes: none;
39. }
40. blockquote:before, blockquote:after,
41. q:before, q:after {
42.     content: '';
43.     content: none;
44. }
```

Evaluation
Copy

```
45. table {  
46.     border-collapse: collapse;  
47.     border-spacing: 0;  
48. }
```

Attaching this stylesheet to `default-styles.html` will render the page like this:



8.8. CSS Normalizers

CSS normalizers are slightly different from CSS resets. Instead of getting rid of all styles, they adjust styles so that they are consistent between browsers. In doing so, normalizers have to choose some styles over others. As such, normalizers are *opinionated*, meaning that the authors of these normalizers have made design choices based on their own preferences. For the rest of this course, we will be using a slightly modified version of a normalizer called `normalize.css`²⁶ in all of our files.

26. <https://necolas.github.io/normalize.css/>

Normalize

`normalize.css` is the normalizer used by Bootstrap²⁷, a popular open source toolkit for developing web pages, to improve cross-browser rendering.



8.9. External Stylesheets, Embedded Stylesheets, and Inline Styles

❖ 8.9.1. External Stylesheets

External stylesheets are created in separate documents with a `.css` extension. An external stylesheet is simply a listing of rules. It cannot contain HTML tags. Throughout this course, we will mainly be working with external stylesheets. `CrashCourse/Demos/styles.css` is an example of an external stylesheet.

Demo 8.3: `CrashCourse/Demos/styles.css`

```
1.  .warning {
2.      color: red;
3.  }
4.
5.  h1.warning {
6.      text-decoration: underline;
7.  }
8.
9.  p.warning {
10.     font-weight: bold;
11. }
```

The above CSS file can be included in any number of HTML pages using the `<link>` tag, which usually goes in the head of an HTML page:

27. <https://getbootstrap.com>

```
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link href="styles.css" rel="stylesheet">
<title>Page Title</title>
</head>
```

<link> Attributes

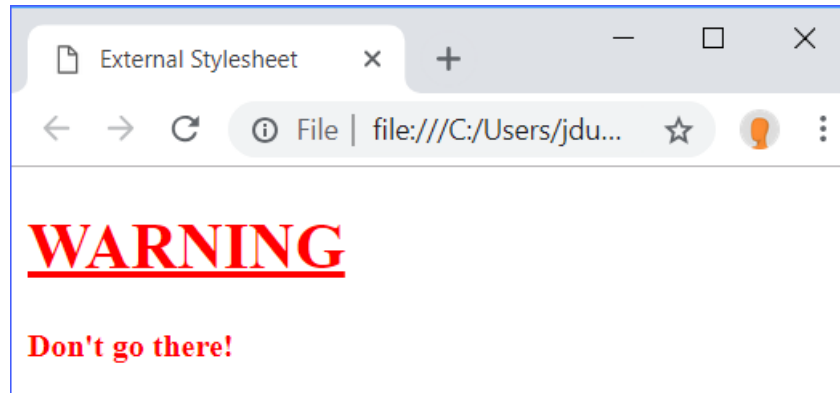
- href - points to the location of the external stylesheet.
- rel - must be set to “stylesheet” for linking stylesheets.

Notice the <link> tag in the code below that links to styles.css:

Demo 8.4: CrashCourse/Demos/external-stylesheet.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="../normalize.css" rel="stylesheet">
7. <link href="styles.css" rel="stylesheet">
8. <title>External Stylesheet</title>
9. </head>
10. <body class="webucator">
11.   <h1 class="warning">WARNING</h1>
12.   <p class="warning">Don't go there!</p>
13. </body>
14. </html>
```

This page will render as follows:



There is no limit to the number of external stylesheets a single HTML page can use. Notice in the preceding example that we linked to two external stylesheets:

```
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,initial-scale=1">
<link href="../normalize.css" rel="stylesheet">
<link href="styles.css" rel="stylesheet">
<title>External Stylesheet</title>
</head>
```

Also, as you will see, external stylesheets can be combined with embedded stylesheets and inline styles.

❖ 8.9.2. Embedded Stylesheets

Embedded stylesheets appear in the `style` element, which usually goes in the head of an HTML page. The code below shows a page with an embedded stylesheet:

Demo 8.5: CrashCourse/Demos/embedded-stylesheet.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="../normalize.css" rel="stylesheet">
7.  <style>
8.    .warning {
9.      color: red;
10.    }
11.
12.    h1.warning {
13.      text-decoration: underline;
14.    }
15.
16.    p.warning {
17.      font-weight: bold;
18.    }
19.  </style>
20.  <title>Embedded Stylesheet</title>
21. </head>
22. <body class="webucator">
23.   <h1 class="warning">WARNING</h1>
24.   <p class="warning">Don't go there!</p>
25. </body>
26. </html>
```

This page will render the same as the HTML page with the external stylesheet.

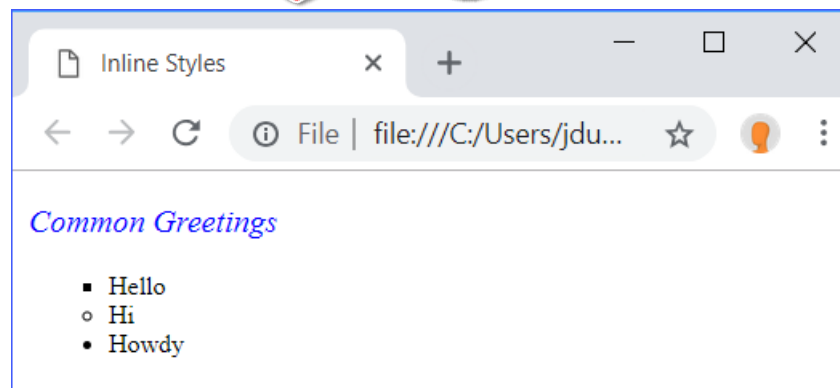
❖ 8.9.3. Inline Styles

Inline styles are created by adding the `style` attribute to a tag. As with the `class` and `id` attributes, all elements can take the `style` attribute. The value of the `style` attribute is a list of one or more declarations separated by semicolons. The code sample below illustrates how inline styles are used:

Demo 8.6: CrashCourse/Demos/inline-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 href="../normalize.css" rel="stylesheet">
7. <title>Inline Styles</title>
8. </head>
9. <body class="webucator">
10. <p style="color: blue; font-style: italic; font-size: 1em;">
11.   Common Greetings
12. </p>
13. <ul style="font-size: 0.8em;">
14.   <li style="list-style-type: square;">Hello</li>
15.   <li style="list-style-type: circle;">Hi</li>
16.   <li style="list-style-type: disc;">Howdy</li>
17. </ul>
18. </body>
19. </html>
```

This page will render as follows:



Avoid Using Inline Styles

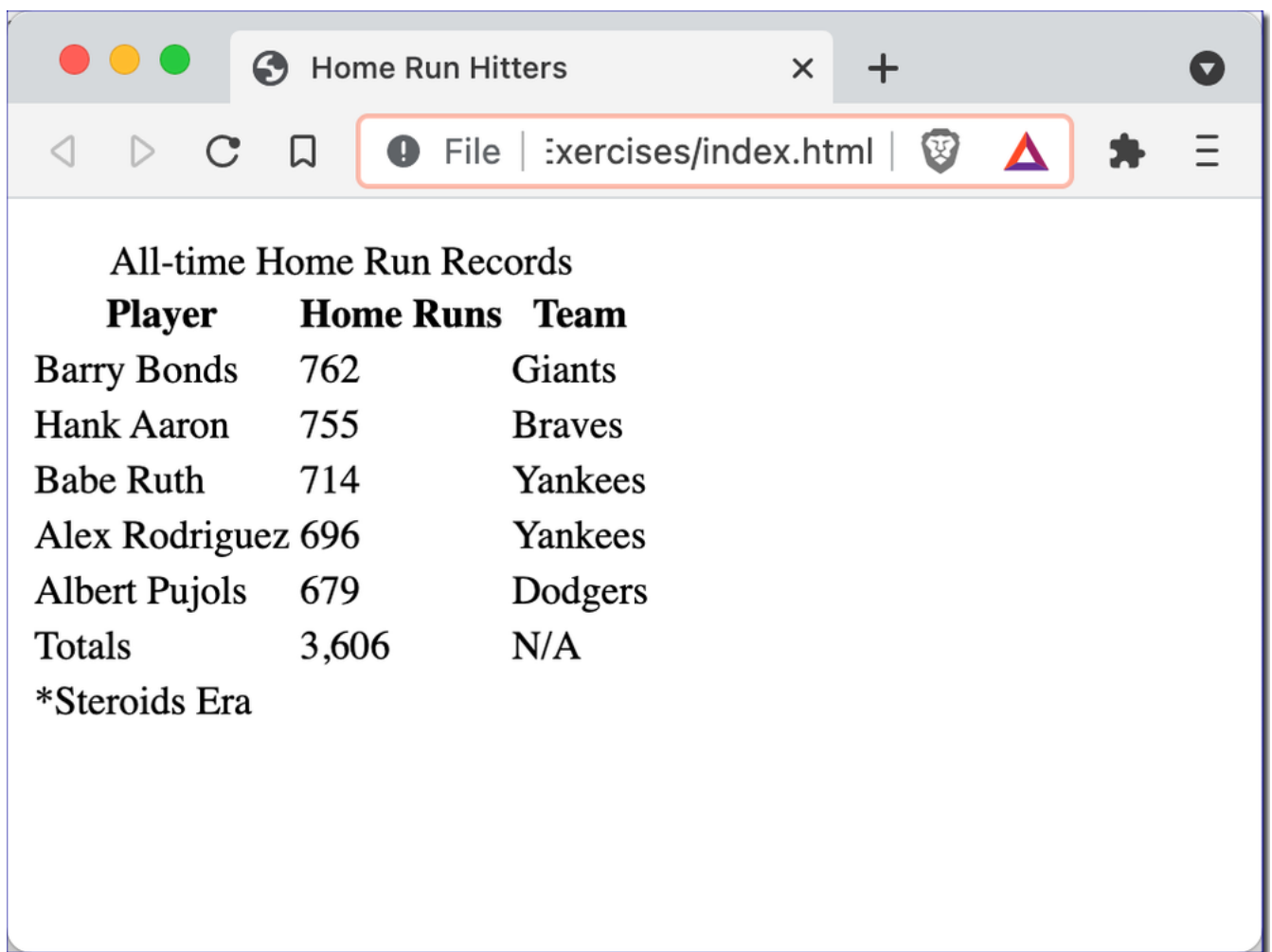
It is generally bad practice to use inline styles in production code; however, they can be useful for testing and debugging.

Exercise 8: Creating an External Stylesheet

 25 to 40 minutes

In this exercise, you will add several simple rules to an external stylesheet and link to it from `Crash Course/Exercises/index-external-styles.html`. Do not worry about learning the CSS properties and values at this point. This exercise is just to give you some practice creating a stylesheet.

1. Open `CrashCourse/Exercises/index.html` in a browser. It should look like this:



2. Open `CrashCourse/Exercises/index.html` in your editor and save it as `index-external-styles.html`.

3. Create a new file and save it as `styles.css` in the same directory.
4. Add a `<link>` tag with an `href` value of “`styles.css`” and a `rel` value of “stylesheet” in the head of `index-external-styles.html`.
5. In `styles.css`, add a rule for the `body` element that contains the following declarations:
 - `background-attachment: fixed;`
 - `background-image: url(images/baseball.png);`
 - `background-position: bottom;`
 - `background-repeat: repeat-x;`

For a refresher on how to add rules, refer to the **CSS Rules** section of this lesson (see page 114).

6. Add a rule for `table` elements that contains the following declarations:
 - `background-color: white;`
 - `border-collapse: collapse;`
 - `margin: auto;`
 - `min-width: 800px;`
7. Add a rule for `caption` elements that contains the following declaration:
 - `font-style: italic;`
8. Add a rule for `thead` elements that contains the following declarations:
 - `background-color: darkblue;`
 - `color: white;`

9. Add a rule for `td` and `th` elements that contains the following declarations:
 - `border: 1px solid orange;`
 - `margin: 0;`
 - `padding: .3em;`

For a refresher on adding a rule for multiple selectors, refer to **Grouping** within the **Selectors** section of this lesson (see page 119).

10. Add a rule for `td` elements that are in the `tfoot` that contains the following declarations:
 - `background-color: silver;`
 - `border-top: 2px solid black;`

- `font-weight: bold;`

For a refresher on adding a rule for an element that is contained within another element, refer to **Descendant Combinators** in the **Combinators** section of this lesson (see page 120).

11. Add a rule for elements with the “steroids-era” class that contains the following declaration:

- `font-style: italic;`

For a refresher on adding a rule for an element of a specific class, refer to the **Class Selectors** within the **Selectors** section of this lesson (see page 116).

12. Add a rule for the element with the “key” id that contains the following declarations:

- `background-color: lightgray;`
- `font-style: italic;`

For a refresher on adding a rule for an element of a specific id, refer to the **ID Selectors** within the **Selectors** section of this lesson (see page 117).

13. Add the following two rules to the bottom of `styles.css`:

```
tr td:nth-child(2) {  
    text-align: center;  
}  
  
.steroids-era td:first-child::after {  
    content: '*';  
}
```

Just add these last two rules exactly as written. Don’t worry about understanding the selectors yet. We will cover them in later lessons. For now, we just want you to see what CSS can do.

14. In `index-external-styles.html`:

- A. Assign a class of “steroids-era” to the table rows containing Barry Bonds and Alex Rodriguez.
- B. Assign an id of “key” to the table data cell in the `tfoot` that contains the text “*Steroids Era.”

15. Open `index-external-styles.html` in a browser. It should look like this:

Home Run Hitters

×

+

[ts/Webucator/CSS/CrashCourse/Solutions/index-embedded-styles.html](#)

All-time Home Run Records

Player	Home Runs	Team
Barry Bonds*	762	Giants
Hank Aaron	755	Braves
Babe Ruth	714	Yankees
Alex Rodriguez*	696	Yankees
Albert Pujols	679	Dodgers
Totals	3,606	N/A

*Steroids Era

Solution: CrashCourse/Solutions/index-external-styles.html

```
-----Lines 1 through 6 Omitted-----
7.  <link href="styles.css" rel="stylesheet">
8.  <title>Home Run Hitters</title>
9.  </head>
10. <body class="webucator">
11.   <table>
12.     <caption>All-time Home Run Records</caption>
13.     <thead>
14.       <tr>
15.         <th>Player</th>
16.         <th>Home Runs</th>
17.         <th>Team</th>
18.       </tr>
19.     </thead>
20.     <tbody>
21.       <tr class="steroids-era">
22.         <td>Barry Bonds</td>
23.         <td>762</td>
24.         <td>Giants</td>
25.       </tr>
-----Lines 26 through 35 Omitted-----
36.       <tr class="steroids-era">
37.         <td>Alex Rodriguez</td>
38.         <td>696</td>
39.         <td>Yankees</td>
40.       </tr>
-----Lines 41 through 52 Omitted-----
53.       <tr>
54.         <td colspan="3" id="key">*Steroids Era</td>
55.       </tr>
56.     </tbody>
57.   </table>
58. </body>
59. </html>
```

Solution: CrashCourse/Solutions/styles.css

```
1.  body {
2.      background-attachment: fixed;
3.      background-image: url(images/baseball.png);
4.      background-position: bottom;
5.      background-repeat: repeat-x;
6.  }
7.
8.  table {
9.      background-color: white;
10.     border-collapse: collapse;
11.     margin: auto;
12.     min-width: 800px;
13. }
14.
15. caption {
16.     font-style: italic;
17. }
18.
19. thead {
20.     background-color: darkblue;
21.     color: white;
22. }
23.
24. td,
25. th {
26.     border: 1px solid orange;
27.     margin: 0;
28.     padding: .3em;
29. }
30.
31. tfoot td {
32.     background-color: silver;
33.     border-top: 2px solid black;
34.     font-weight: bold;
35. }
36.
37. .steroids-era {
38.     font-style: italic;
39. }
40.
41. #key {
42.     background-color: lightgray;
43.     font-style: italic;
44. }
```

Evaluation
Copy

```
45.  
46. tr td:nth-child(2) {  
47.     text-align: center;  
48. }  
49.  
50. .steroids-era td:first-child::after {  
51.     content: '*';  
52. }
```

Exercise 9: Creating an Embedded Stylesheet

⌚ 5 to 10 minutes

In this exercise, you will replace your external stylesheet with an embedded stylesheet.

1. Open `CrashCourse/Exercises/index-external-styles.html` in your editor and save it as `index-embedded-styles.html`.
2. Remove the `<link>` tag, and add a `style` block.
3. Copy and paste all the CSS rules from `styles.css` to the `style` block in `index-embedded-styles.html`.
4. Open `index-embedded-styles.html` in a browser. It should look the same as `index-external-styles.html` did (see page 135).

Solution: CrashCourse/Solutions/index-embedded-styles.html

```
-----Lines 1 through 6 Omitted-----
7.  <title>Home Run Hitters</title>
8.  <style>
9.    body {
10.      background-attachment: fixed;
11.      background-image: url(images/baseball.png);
12.      background-position: bottom;
13.      background-repeat: repeat-x;
14.    }
15.
16.    table {
17.      background-color: white;
18.      border-collapse: collapse;
19.      margin: auto;
20.      min-width: 800px;
21.    }
22.
23.    caption {
24.      font-style: italic;
25.    }
26.
27.    thead {
28.      background-color: darkblue;
29.      color: white;
30.    }
31.
32.    td,
33.    th {
34.      border: 1px solid orange;
35.      margin: 0;
36.      padding: .3em;
37.    }
38.
39.    tfoot td {
40.      background-color: silver;
41.      border-top: 2px solid black;
42.      font-weight: bold;
43.    }
44.
45.    .steroids-era {
46.      font-style: italic;
47.    }
48.
49.    #key {
```

```
50.     background-color: lightgray;
51.     font-style: italic;
52. }
53.
54. tr td:nth-child(2) {
55.     text-align: center;
56. }
57.
58. .steroids-era td:first-child::after {
59.     content: '*';
60. }
61. </style>
62. </head>
-----Lines 63 through 112 Omitted-----
```



Exercise 10: Adding Inline Styles

⌚ 10 to 20 minutes

In this exercise, you will add some inline styles.

1. Open `CrashCourse/Exercises/index-external-styles.html` and save it as `index-inline-styles.html`.
2. Add inline styles to the table data cells containing the word “Giants”. The style should contain the following declarations:
 - `color: darkred;`
 - `font-style: italic;`
3. Add an inline style to the table data cell containing the word “Braves”. The style should contain the following declarations:
 - `color: red;`
 - `font-style: italic;`
4. Add an inline style to the table data cells containing the word “Yankees”. The style should contain the following declarations:
 - `color: navy;`
 - `font-style: italic;`
5. Add an inline style to the table data cells containing the word “Dodgers”. The style should contain the following declarations:
 - `color: dodgerblue;`
 - `font-style: italic;`
6. Open `index-inline-styles.html` in a browser. The last column should now look like this:

Team
<i>Giants</i>
<i>Braves</i>
<i>Yankees</i>
<i>Yankees</i>
<i>Dodgers</i>
N/A

Solution: CrashCourse/Solutions/index-inline-styles.html

```
-----Lines 1 through 19 Omitted-----
20.     <tbody>
21.         <tr class="steroids-era">
22.             <td>Barry Bonds</td>
23.             <td>762</td>
24.             <td style="color: darkred; font-style: italic;">Giants</td>
25.         </tr>
26.         <tr>
27.             <td>Hank Aaron</td>
28.             <td>755</td>
29.             <td style="color: red; font-style: italic;">Braves</td>
30.         </tr>
31.         <tr>
32.             <td>Babe Ruth</td>
33.             <td>714</td>
34.             <td style="color: navy; font-style: italic;">Yankees</td>
35.         </tr>
36.         <tr class="steroids-era">
37.             <td>Alex Rodriguez</td>
38.             <td>696</td>
39.             <td style="color: navy; font-style: italic;">Yankees</td>
40.         </tr>
41.         <tr>
42.             <td>Albert Pujols</td>
43.             <td>679</td>
44.             <td style="color: dodgerblue; font-style: italic;">Dodgers</td>
45.         </tr>
46.     </tbody>
-----Lines 47 through 59 Omitted-----
```



8.10. <div> and

The <div> and tags are used in conjunction with Cascading Style Sheets. By themselves, they do very little. In fact, the tag without CSS has no visual effect on its contents. The only visual effect of the <div> tag is to block off its contents, similar to putting a
 tag before and after a section on the page.

Like all tags, the <div> and tags can take the class, id, and style attributes. It is through these attributes that styles are applied to the elements. Unlike p (paragraph) elements, main (main

content) elements, header (header content) elements, etc., `div` elements and `span` elements do not inherently represent anything. `div` elements are generic content containers for *flow* content and `span` elements are generic containers of *phrasing* content.

Flow and Phrasing Content


Phrasing content is content that *can fit* inside a sentence: images, emphasized text, links (usually), etc. Flow content is content that *cannot fit* inside a sentence: headings, tables, sections, etc.

For more information on content categories, see https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Content_categories.

The following demo shows how you can apply styles to `div` and `span` elements:

Demo 8.7: CrashCourse/Demos/div-and-span.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="../../normalize.css" rel="stylesheet">
7.  <style>
8.      div {
9.          border-color: blue;
10.         border-style: groove;
11.         border-width: 2em;
12.         font-family: Verdana;
13.         font-size: 1em;
14.         padding: .5em;
15.     }
16.
17.     span {
18.         color: red;
19.         font-style: italic;
20.     }
21. </style>
22. <title>Div and Span</title>
23. </head>
24. <body class="webucator">
25. <div>
26.     <p>
27.         All of this text is in a div, and
28.         <span>this text is also in a span</span>.
29.     </p>
30. </div>
31. </body>
32. </html>
```



This page will render as follows:



Exercise 11: Styling div and span

🕒 10 to 20 minutes

In this exercise, you will add `class` and `id` attributes to `<div>` and `` tags on an already existing HTML page. Open `CrashCourse/Exercises/divs-and-spans.html` and `CrashCourse/Exercises/styles-divs-spans.css` in your editor. You will need to modify the HTML page based on the code in the CSS page. You will not need to modify the CSS page. Your goal is to make the page render as follows:



There are no step-by-step instructions. Review the rules in the external stylesheet (`styles-divs-spans.css`) and apply classes and ids as appropriate.

Solution: CrashCourse/Solutions/divs-and-spans.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="../normalize.css" rel="stylesheet">
7.  <link href="styles-divs-spans.css" rel="stylesheet">
8.  <title>Divs and Spans</title>
9.  </head>
10. <body class="webucator">
11. <div id="top-div">
12.   <span class="left-span">1</span>
13.   <span class="mid-span">2</span>
14.   <span class="right-span">3</span>
15. </div>
16. <div id="mid-div">
17.   <span class="left-span">4</span>
18.   <span class="mid-span">5</span>
19.   <span class="right-span">6</span>
20. </div>
21. <div id="bottom-div">
22.   <span class="left-span">7</span>
23.   <span class="mid-span">8</span>
24.   <span class="right-span">9</span>
25. </div>
26. </body>
27. </html>
```



8.11. Media Types

Styles can be defined for different media. For example, you may want to style a page one way for viewing with a browser and a different way for viewing in print. Some possible media types are:

Media List

- all
- print
- screen
- speech

To define the media type for an entire external or embedded stylesheet, the `media` attribute is added to the `<link>` or `<style>` tag, and assigned the value of a media type. If the `media` attribute is not included, the media type defaults to `all`.

```
<link href="stylesheet.css" rel="stylesheet" media="screen">

<style media="all">
  /* rules */
</style>
```

It is also possible to target multiple media types within one stylesheet using `@media`.

```
@media screen {
  /* rules */
}

@media print {
  /* rules */
}
```

`@media` is a powerful CSS tool. As you will see later, it can be used to target not just the type of media but also aspects of the user's device: screen width, screen height, orientation, etc. This is done with media queries. Media queries allow us to craft pages that are responsive, presenting different layouts for desktop computers, tablet devices, and smartphones.

The viewport meta tag

You may have noticed that all of our files have a line of code in the head that looks like this:

```
<meta name="viewport" content="width=device-width,initial-scale=1">
```

This is called the *viewport* meta tag and it is used to help pages adjust correctly for all devices. You should use it on all your web pages. For detailed information, see https://developer.mozilla.org/en-US/docs/Web/HTML/Viewport_meta_tag.



8.12. Units of Measurement

CSS allows for the use of many different units when specifying property values such as font size, border size, margins, etc. However, standards have been established for what units to use for different media types. In these lessons, except where specifically stated, you will be designing for screen.

Note that units of measurement and sizing will be covered throughout these lessons. In this section, we provide a general overview of the different units of measurement.

❖ 8.12.1. Absolute vs. Relative Units

Absolute units have a basis in the physical world; however, their actual size on the screen depends on the resolution and size of the user's device. When designing for screen, it is standard to use *pixels* for absolute units.

Relative units are relative to the size of other things (e.g., window size, font size, container element size, etc.). When designing for screen, it is standard to use *ems*, *rems*, and *percentages* for relative units.

❖ 8.12.2. Pixels

Pixels are the only recommended absolute unit to be used when designing for screen. A common practice is to use pixels to set *root* font sizes for different screen and window sizes. Root font sizes are font sizes set on the `html` or `body` element. This allows for designs using mostly *ems* and *rems* to adjust between different screen and window sizes because *ems* and *rems* are relative to font size.

❖ 8.12.3. Ems and Rems


You should use *ems* and *rems* for almost everything when building responsive designs for screens. The exception to this rule is height and width, which are more often defined in pixels (for absolute values) and percentages or viewport units (for relative values). An *em* is the size of the font size for the current element or the parent element. For example, if the font-size of the `html` element is set to 14px, then for all elements within that `html` element, 1em is 14px.

Rems are similar to *ems*, except for one key difference. The 'r' in *rem* stands for 'root' because 1rem is always equal to the root font size (the font size set for the `html` element) regardless of what element it is in. This means 1rem is consistent throughout the page, while 1em is not. Both *ems* and *rems* can be used to accomplish the same things.

The code below illustrates the difference between *ems* and *rems*:

Demo 8.8: CrashCourse/Demos/ems-vs-rem.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="../../normalize.css" rel="stylesheet">
7.  <style>
8.    /* Tighten vertical spacing of headings. */
9.    h1,
10.   h2,
11.   h3 {
12.     margin-bottom: .2rem;
13.     margin-top: .2rem;
14.   }
15. </style>
16. <title>Ems vs Rems</title>
17. </head>
18. <body class="webucator">
19.   <h1>Using em</h1>
20.   <h2 style="margin-left: 2em;">Subtitle 1</h2>
21.   <h3 style="margin-left: 2em;">Subtitle 2</h3>
22.
23.   <hr>
24.
25.   <h1>Using rem</h1>
26.   <h2 style="margin-left: 2rem;">Subtitle 1</h2>
27.   <h3 style="margin-left: 2rem;">Subtitle 2</h3>
28. </body>
29. </html>
```

A large, red, 3D-style watermark with the words "Evaluation" and "Copy" stacked diagonally is overlaid on the right side of the code block.

The code above will render the following:



Notice that “Subtitle 1” and “Subtitle 2” do not line up in the **Using em** section, but do line up in the **Using rem** section. This is because the ems are relative to the font sizes of their h2 and h3 elements, while the rem's are not:

- 2em = twice the size of the *inherited* font.
- 2rem = twice the size of the *base* font.

❖ 8.12.4. Percentages

Percentages are most often used to create flexible layouts that change to fit the size of the browser window. When specifying a value such as width or height in percentages, the percentage is relative to the size of the containing element. For font-size, percentages work the same as ems. 100% is equal to the inherited font size, so 100% is essentially 1em when specifying font size. It is most common to use percentages to specify values like width and height to create “fluid” boxes that grow and shrink to fit within the browser window.

❖ 8.12.5. Other Units

Inches (in), centimeters (cm), millimeters (mm), points (pt), and picas (pc) are all absolute units that are most often used when designing for print.

You might also run into viewport units, which are relative units based on the size of the viewport (the browser window). `vw` and `vh` specify sizes relative to the user's viewport width and height, respectively.

1. `vw` is 1/100th of the viewport's width.
2. `vh` is 1/100th of the viewport's height.
3. `vmax` specifies size relative to whichever dimension is larger: width or height.
4. `vmin` specifies size relative to whichever dimension is smaller: width or height.

Viewport units are used for designing for screen, but they are not as commonly used as other screen units, in part because the specification is not clear on how they should behave on mobile²⁸. There are newer units called *dynamic viewport units* coming, but major browsers do not yet support them²⁹.

If you would like to play with the different CSS units to see how they compare on a screen, open `CrashCourse/Demos/units-of-measurement.html` (shown below) in your browser:

28. <https://www.bram.us/2021/07/08/the-large-small-and-dynamic-viewports/>

29. <https://caniuse.com/?search=viewport%20units>

Units of Measurement

File | tor/CSS/CrashCourse/Demos/units-of-measurement.html

Units of Measurement

Unit Size:
Font Size:

Relative Units

Unit	Abbr	Length	To Change
Width of capital M	em	<div>M</div>	Change font size.
Height of lowercase x	ex	<div>x</div>	Change font size.
Width of the number 0	ch	<div>0</div>	Change font size.
Root em	rem	<div></div>	n/a
1/100 th of Width of Viewport	vw	<div></div>	Change width of browser window.
1/100 th of Height of Viewport	vh	<div></div>	Change height of browser window.
1/100 th of Viewport's Smaller Dimension	vmin	<div></div>	Change height or width of browser window.
1/100 th of Viewport's Larger Dimension	vmax	<div></div>	Change height or width of browser window.

Absolute Units

Unit	Abbr	Length
Pixels	px	<div></div>
Quarter millimeters	q	<div></div>
Millimeters	mm	<div></div>
Centimeters	cm	<div></div>
Picas	pc	<div></div>
Points	pt	<div></div>
Inches	in	<div></div>

Percentage

For even more detailed information on CSS values and units see https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/Values_and_units.



8.13. Inheritance

By default, all CSS properties are either **inherited** or **non-inherited** properties. This difference determines what happens when a property for an element goes unspecified.

If an inherited property for an element goes unspecified, then that element will inherit the value from its parent element. A common inherited property is `font-size`:

Demo 8.9: CrashCourse/Demos/inherited.html

```
1. <!DOCTYPE HTML>
2. <html>
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link href="../../normalize.css" rel="stylesheet">
7. <style>
8.   p {
9.     font-size: 20px;
10.  }
11. </style>
12. <title>Inherited Properties</title>
13. </head>
14. <body class="webucator">
15.   <p>This is a paragraph element with a
16.     <strong>strong element</strong> within it.</p>
17. </body>
18. </html>
```

The above code will render the following:

This is a paragraph element with a **strong element** within it.

Notice that the `strong` element has the same font size as the `p` element, even though `font-size` was not specified for it.

If a non-inherited property for an element goes unspecified, then that element will get the initial (default) value of that property. A common non-inherited property is `border`:

Demo 8.10: CrashCourse/Demos/non-inherited.html

```
1. <!DOCTYPE HTML>
2. <html>
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width,initial-scale=1">
6. <link href="../normalize.css" rel="stylesheet">
7. <style>
8.   p {
9.     border: 0.2rem solid black;
10.  }
11. </style>
12. <title>Non-Inherited Properties</title>
13. </head>
14. <body class="webucator">
15.   <p>This is a paragraph element with a
16.     <strong>strong element</strong> within it.</p>
17. </body>
18. </html>
```

The above code will render the following:

This is a paragraph element with a **strong element** within it.

Notice that the `strong` element does not also have its own border. This is because the initial value for `border` is `none` and `border` is **not** an inherited property.

❖ 8.13.1. The inherit Value

You can force a property to inherit the value of its parent by setting its value to `inherit`.

The `inherit` property is most often used to override other rules, as illustrated in the following demo:

Demo 8.11: CrashCourse/Demos/inherit-styles.css

```
1.  h2 {  
2.      color: blue;  
3.  }  
4.  
5.  article h2 {  
6.      color: inherit;  
7.  }  
8.  
9.  #article-red {  
10.     color: red;  
11. }  
12.  
13. #article-green {  
14.     color: green;  
15. }  
16.  
17. #article-purple {  
18.     color: purple;  
19. }
```

Evaluation
Copy

Demo 8.12: CrashCourse/Demos/inherit.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <h1>inherit</h1>
12.   <header>
13.     <h2>This is the Header</h2>
14.     <p>This is a paragraph.</p>
15.   </header>
16.   <article id="article-red">
17.     <h2>Article Red</h2>
18.     <p>This is a paragraph.</p>
19.   </article>
20.   <article id="article-green">
21.     <h2>Article Green</h2>
22.     <p>This is a paragraph.</p>
23.   </article>
24.   <article id="article-purple">
25.     <h2>Article Purple</h2>
26.     <p>This is a paragraph.</p>
27.   </article>
28.   <footer>
29.     <h2>This Is the Footer</h2>
30.     <p>This is a paragraph.</p>
31.   </footer>
32. </body>
33. </html>
```

The above code will render the following:³⁰

30. If you are reading this in black and white, be sure to open the page in your browser.



Notice that this rule says that all h2 elements should be blue:

```
h2 {  
  color: blue;  
}
```

This rule would normally trump inheritance. In other words, by default the h2 elements would inherit the color of their parent element, but the rule above overrides that, specifying that all h2 elements should be blue.

But in the result, the h2 elements in the `article` elements are **not** blue. Why?

Because this more specific rule says that h2 elements within `article` elements should inherit the `color` property from their parent elements:

```
article h2 {  
    color: inherit;  
}
```

Evaluation
Copy

Conclusion

Cascading Style Sheets provide a far better way of formatting HTML pages than the traditional use of HTML tags. In this lesson, you have developed a foundation for creating and applying CSS rules.

LESSON 9

CSS Fonts

Topics Covered

- ☒ font-family
- ☒ @font-face.
- ☒ font-size
- ☒ font-style
- ☒ font-variant
- ☒ font-weight
- ☒ line-height
- ☒ Shorthand properties.
- ☒ font

Evaluation
Copy

Introduction

In this lesson, you will learn to use CSS to modify font properties. You will also learn to work with CSS shorthand properties.



9.1. font-family

The `font-family` property is used in CSS to specify the font applied to an element. You can specify by font name or font category.

❖ 9.1.1. Specifying by Font Name

When you specify a font by font name, the browser will look for the named font on the end user's computer. If it finds it, the text will be displayed in that font. For example, the following rule would make the font of all `<p>` tags Arial:

```
p {  
  font-family: Arial;  
}
```

If the Arial font were not found on the end user's computer, the browser would display a default font. If you are concerned that the font name you want to use might not be found on a user's computer, you can provide a list of options.

```
p {  
  font-family: Arial, Helvetica;  
}
```

In this case, the browser will first look for Arial. If it doesn't find Arial, it will then look for Helvetica.

❖ 9.1.2. Specifying Font by Category

When you specify a font by category, the browser will use the font the user's computer specifies for that category. For example, for monospace, the computer might specify Courier. Some common font family categories are listed below:

- cursive
- fantasy
- monospace
- sans-serif
- serif

To be extra safe, designers often specify a couple specific options followed by a font family category, like so:

```
p {  
  font-family: Arial, Helvetica, sans-serif;  
}
```

This way, if neither Arial nor Helvetica is found, the browser at least knows to use some sans-serif font.

The following code samples illustrate how `font-family` works:

Demo 9.1: Fonts/Demos/font-family-styles.css

```
1.  #cursive-div {
2.      font-family: 'Cool Linked Font', cursive;
3.  }
4.
5.  #fantasy-div {
6.      font-family: 'Cool Linked Font', fantasy;
7.  }
8.
9.  #monospace-div {
10.     font-family: 'Courier New', monospace;
11. }
12.
13. #sans-serif-div {
14.     font-family: Tahoma, Verdana, Arial, sans-serif;
15. }
16.
17. #serif-div {
18.     font-family: Times, 'Times New Roman', Georgia, serif;
19. }
```

Notice that `#cursive-div` and `#fantasy-div` both have a ‘Cool Linked Font’ listed before their generic family names. Cursive and fantasy fonts are not as popular or as widely supported as the other three, so it is common to load your own fonts for these two font families.

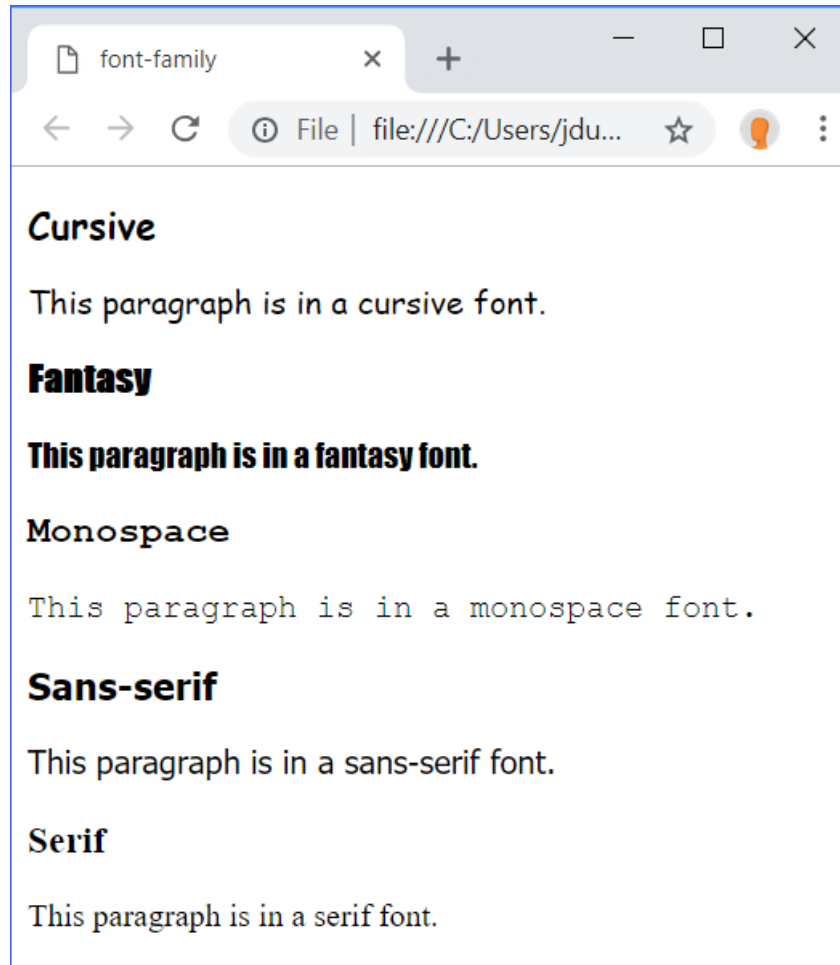
Also notice that fonts that have multi-word names (e.g., Times New Roman) should be contained in either single or double quotes.

Here is an HTML page that uses the stylesheet shown above:

Demo 9.2: Fonts/Demos/font-family.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <div id="cursive-div">
12.     <h3>Cursive</h3>
13.     <p>This paragraph is in a cursive font.</p>
14.   </div>
15.   <div id="fantasy-div">
16.     <h3>Fantasy</h3>
17.     <p>This paragraph is in a fantasy font.</p>
18.   </div>
19.   <div id="monospace-div">
20.     <h3>Monospace</h3>
21.     <p>This paragraph is in a monospace font.</p>
22.   </div>
23.   <div id="sans-serif-div">
24.     <h3>Sans-serif</h3>
25.     <p>This paragraph is in a sans-serif font.</p>
26.   </div>
27.   <div id="serif-div">
28.     <h3>Serif</h3>
29.     <p>This paragraph is in a serif font.</p>
30.   </div>
31. </body>
32. </html>
```

The code above will render as follows:



The web-safe fonts are those fonts that are most commonly installed on computers and are therefore safe to use in your CSS. You can see the full list of web-safe fonts on cssfontstack.com³¹.



9.2. @font-face

The CSS `@font-face` rule enables the loading of a font file, and thus offers us the ability to use any font, not just those our users happen to have loaded on their computer or device. The `@font-face` rule is defined in the W3C's CSS Fonts Module Level 3³² specification. As the W3C specification states:

31. <https://www.cssfontstack.com/>

32. <https://www.w3.org/TR/css-fonts-3/>

The `@font-face` rule allows for linking to fonts that are automatically fetched and activated when needed. This allows authors to select a font that closely matches the design goals for a given page rather than limiting the font choice to a set of fonts available on a given platform. A set of font descriptors define the location of a font resource, either locally or externally, along with the style characteristics of an individual face. Multiple `@font-face` rules can be used to construct font families with a variety of faces. Using CSS font matching rules, a user agent can selectively load only those faces that are needed for a given piece of text.

In the CSS, you define your own `font-family` and associate it with a font file:³³

```
@font-face {  
  font-family: Gentium;  
  src: url('Gentium.ttf');  
}
```

A subsequent CSS rule could style a given element with the newly defined `font-family`:

```
div.newsite {  
  font-family: Gentium;  
}
```

Browsers will download the specified font file and will use it to render the designated content.

Different browsers support different font file formats. To address these differences, the specification for the `@font-face` rule allows for a series of font files, with format hints to aid browsers in selecting the appropriate file for the font:

```
@font-face {  
  font-family: bodytext;  
  src: url(ideal-sans-serif.woff) format("woff"),  
       url(basic-sans-serif.ttf) format("opentype");  
}
```

33. We will cover how to get font files soon.

Here's a list of the formats supported by the W3C specification:³⁴

1. **Embedded OpenType**

- **Format string:** embedded-opentype
- **Extension(s):** .eot
- Only supported in Internet Explorer.

2. **OpenType**

- **Format string:** opentype
- **Extension(s):** .ttf, .otf
- Well supported, but in Internet Explorer it only works with fonts set to installable.

3. **SVG Font**

- **Format string:** svg
- **Extension(s):** .svg, .svgz
- Not well supported.

4. **TrueType**

- **Format string:** truetype
- **Extension(s):** .ttf
- Well supported, but in Internet Explorer it only works with fonts set to installable.

5. **WOFF (Web Open Font Format)**

- **Format string:** woff
- **Extension(s):** .woff
- Well supported.

6. **WOFF 2.0**

- **Format string:** woff2
- **Extension(s):** .woff2
- Supported well in most recent browsers, but not supported in Internet Explorer or older browsers.

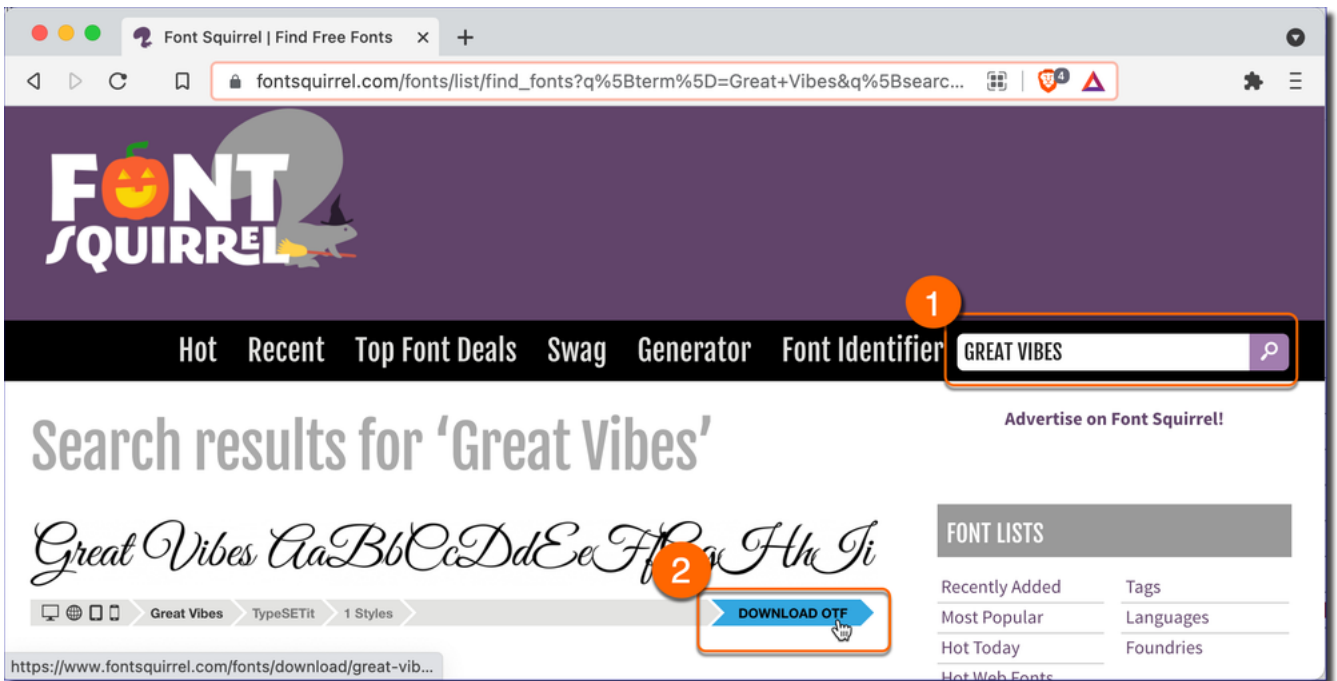
34. <https://www.w3.org/TR/2018/REC-css-fonts-3-20180920/#fontformats>

In the @font-face rule, you can specify additional properties, such as font-style and font-weight, which we will cover soon.

❖ 9.2.1. Getting Fonts

Let's look at a typical example, in which you will download the font “Great Vibes” from Font Squirrel³⁵, a popular resource for free fonts and then convert it into several different file font formats:

1. Search <https://www.fontsquirrel.com> for “Great Vibes”:

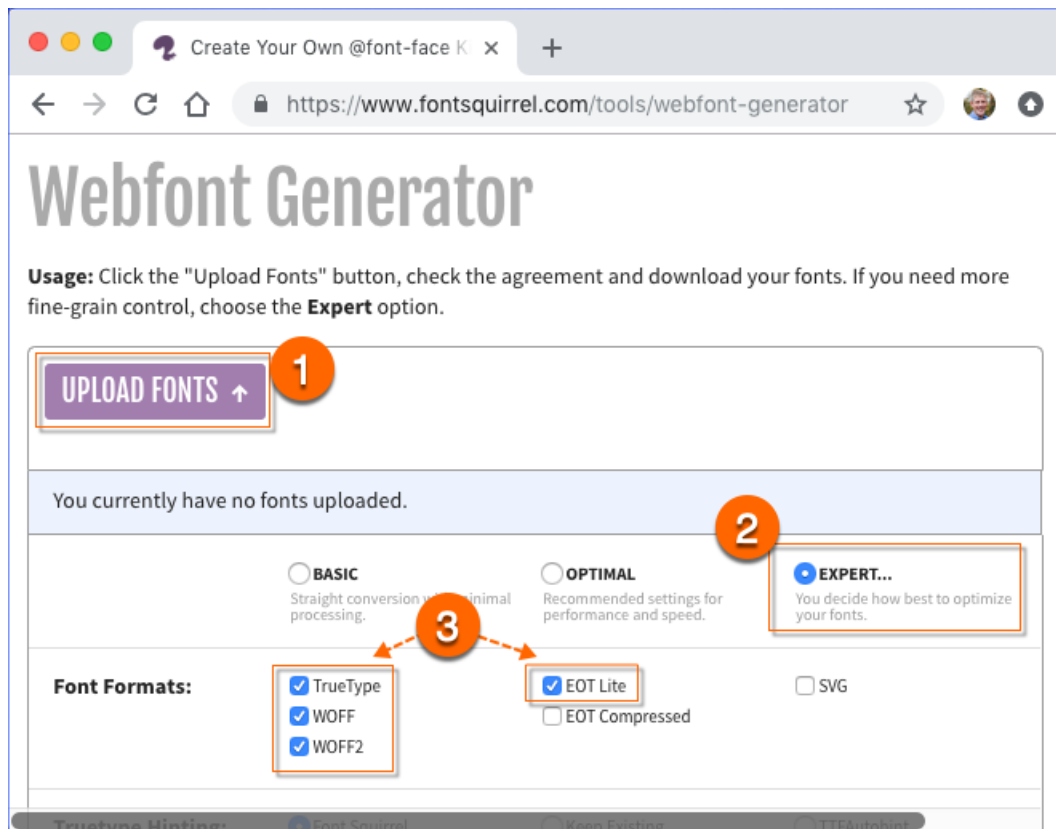


Click the **DOWNLOAD OTF** button to download the zip file containing the font. You can save it wherever you like.

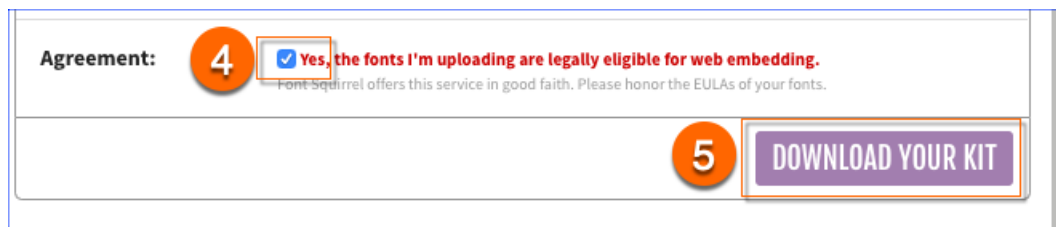
2. Unzip the file. You will see a GreatVibes-Regular.otf file.
3. To convert the GreatVibes-Regular.otf to other file types, use the Font Squirrel Webfont Generator.³⁶
4. Upload the OpenType file you just downloaded and choose four font formats to convert it to. Select the “Expert” option as the other options do not allow for the specific selection of font formats:

35. <https://www.fontsquirrel.com/>

36. <https://www.fontsquirrel.com/tools/webfont-generator>





5. At the bottom of the generator, check the agreement box and click the **DOWNLOAD YOUR KIT** button:



You can save it wherever you like.




6. Unzip the file. Among its contents, you will see:

 greatvibes-regular-demo.html – a demo HTML file using the new font.

 stylesheet.css – A stylesheet demonstrating how to create the @font-face rule:

```
@font-face {
  font-family: 'great_vibesregular';
  src: url('greatvibes-regular-webfont.eot');
  src: url('greatvibes-regular-webfont.eot?#iefix') format('embedded-opentype'),
        url('greatvibes-regular-webfont.woff2') format('woff2'),
        url('greatvibes-regular-webfont.woff') format('woff');
  font-weight: normal;
  font-style: normal;
}
```

- Three font files:

-  greatvibes-regular-webfont.eot
-  greatvibes-regular-webfont.woff
-  greatvibes-regular-webfont.woff2

7. Open greatvibes-regular-demo.html to see the demo, which illustrates the use of “Great Vibes” in different sizes, shows a sample page that uses only the “Great Vibes” font, and provides charts showing special characters and glyphs.


Internet Explorer Bug

In the generated stylesheet, notice the ?#iefix appended to the .eot file. This tricks versions of Internet Explorer to get around a known bug for not loading multiple versions of the font file.

We made a few edits to the generated stylesheet and attached it to a simple HTML page as you can see in the samples below:

Demo 9.3: Fonts/Demos/font-face-styles.css

```
1.  @font-face {
2.      font-family: 'GreatVibes';
3.      src: url('fonts/greatvibes-regular-webfont.eot');
4.      src: url('fonts/greatvibes-regular-webfont.eot?#iefix')
5.          format('embedded-opentype'),
6.          url('fonts/greatvibes-regular-webfont.woff2')
7.          format('woff2'),
8.          url('fonts/greatvibes-regular-webfont.woff')
9.          format('woff'),
10.         url('fonts/greatvibes-regular-webfont.ttf')
11.         format('truetype');
12.      font-weight: normal;
13.      font-style: normal;
14.  }
15.
16.  p {
17.      font-family: GreatVibes;
18.  }
```



We changed the font name from what the generated stylesheet suggested, and we adjusted the paths for the urls. In the Fonts/Demos folder, we moved all of our “Great Vibes” font files into a folder called fonts, and we deleted the unnecessary generated files (e.g., the demo file, the original stylesheet, etc.).

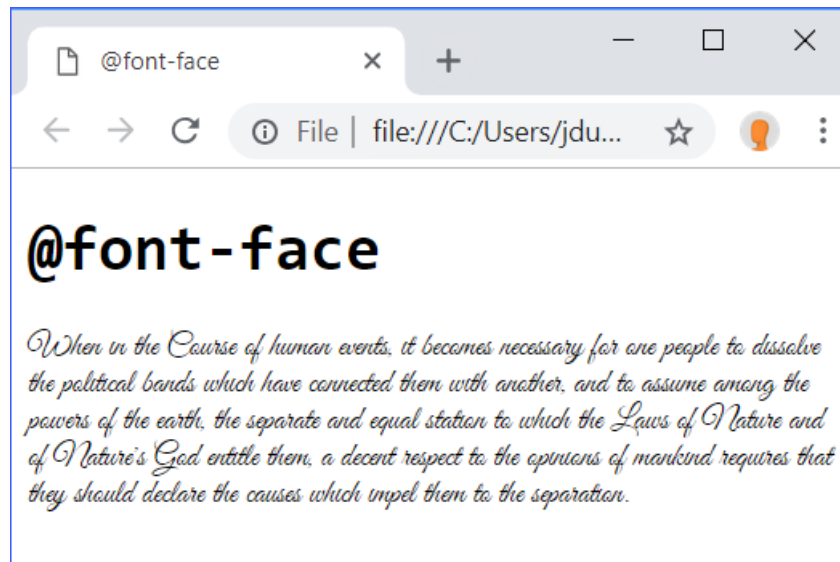
Finally, after the @font-face rule, we used the “Great Vibes” font to style the first paragraph of the U.S. Declaration of Independence.

Here’s the HTML:

Demo 9.4: Fonts/Demos/font-face.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <h1><code>@font-face</code></h1>
12.   <p>When in the Course of human events, it becomes necessary
13.     for one people to dissolve the political bands which have
14.     connected them with another, and to assume among the powers
15.     of the earth, the separate and equal station to which the
16.     Laws of Nature and of Nature's God entitle them, a decent
17.     respect to the opinions of mankind requires that they should
18.     declare the causes which impel them to the separation.</p>
19. </body>
20. </html>
```

And here is the resulting page:



9.3. font-size

font-size is an unexpectedly complicated CSS property. It is difficult to quickly grasp the different font-size units, how they relate to each other, and how they change depending on the font family. It will require some practice and experimentation to get used to.

❖ 9.3.1. Relative font-size Terms

In addition to all the units of measurement we discussed in the Crash Course lesson (see page 154), font size can be defined using the following terms:

- `xx-large`
- `x-large`
- `large`
- `medium`
- `small`
- `x-small`
- `xx-small`
- `smaller`
- `larger`

Evaluation
Copy

The terms `xx-small` to `xx-large` are absolute-size keywords relative to the user's default font size (medium). The terms `smaller` and `larger` are relative-size keywords, meaning that they change the font size of an element relative to its parent element's font size. The following examples illustrate these terms:

Demo 9.5: Fonts/Demos/font-size-styles.css

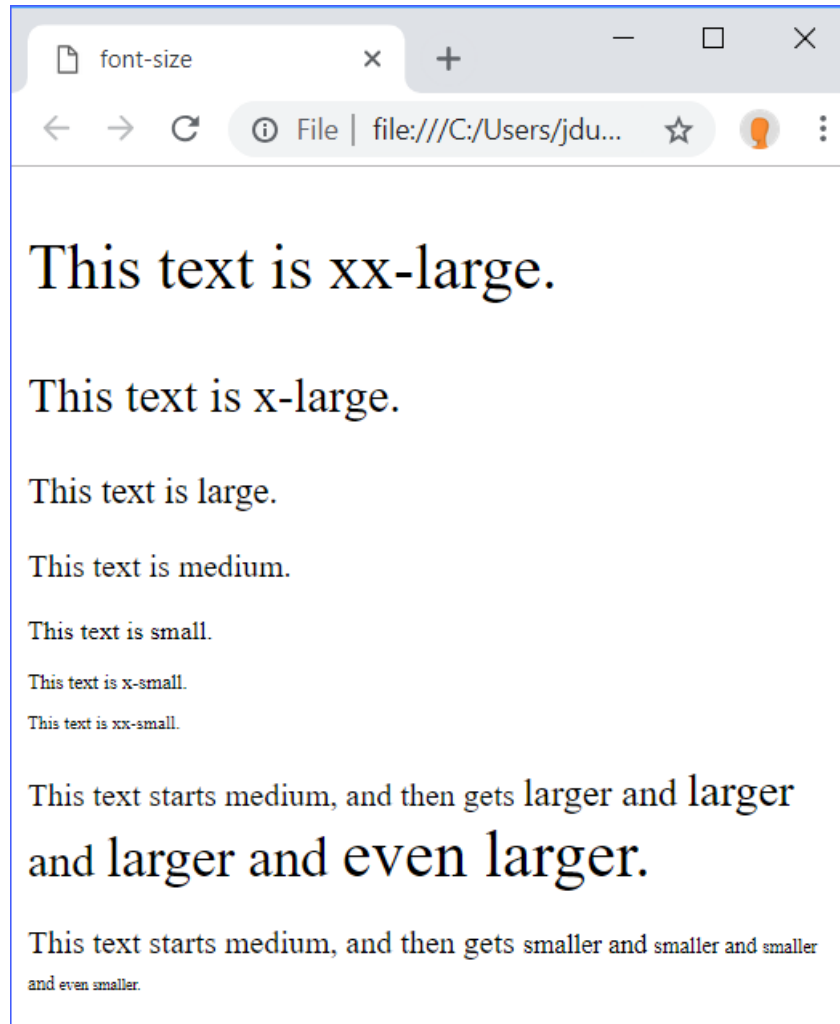
```
1.  .biggest {
2.    font-size: xx-large;
3.  }
4.
5.  .second-biggest {
6.    font-size: x-large;
7.  }
8.
9.  .third-biggest {
10.   font-size: large;
11. }
12.
13. .medium {
14.   font-size: medium;
15. }
16.
17. .third-smallest {
18.   font-size: small;
19. }
20.
21. .second-smallest {
22.   font-size: x-small;
23. }
24.
25. .smallest {
26.   font-size: xx-small;
27. }
28.
29. .larger {
30.   font-size: larger;
31. }
32.
33. .smaller {
34.   font-size: smaller;
35. }
```

Evaluation
Copy

Demo 9.6: Fonts/Demos/font-size.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <p class="biggest">This text is xx-large.</p>
12.   <p class="second-biggest">This text is x-large.</p>
13.   <p class="third-biggest">This text is large.</p>
14.   <p class="medium">This text is medium.</p>
15.   <p class="third-smallest">This text is small.</p>
16.   <p class="second-smallest">This text is x-small.</p>
17.   <p class="smallest">This text is xx-small.</p>
18.   <div id="compare-div" class="medium">
19.     <p>This text starts medium, and then gets
20.       <span class="larger">larger and
21.         <span class="larger">larger and
22.           <span class="larger">larger and
23.             <span class="larger">even larger.
24.               </span>
25.             </span>
26.           </span>
27.         </span>
28.       </p>
29.     <p>This text starts medium, and then gets
30.       <span class="smaller">smaller and
31.         <span class="smaller">smaller and
32.           <span class="smaller">smaller and
33.             <span class="smaller">even smaller.
34.               </span>
35.             </span>
36.           </span>
37.         </span>
38.       </p>
39.   </div>
40. </body>
41. </html>
```

The code above will output the following:



Visualizing Ems and Rems

For a visual representation of fonts using ems and rems, open `CrashCourse/Demos/units-of-measurement.html` in your browser.

❖ 9.3.2. Best Practices

Most experts agree that font size should be defined in relative units (e.g., `em`, `rem`, or `%`) or in terms (e.g., `large`, `small`, etc.). This is because absolute font sizes can make pages inaccessible to people who have difficulty seeing. In most browsers, a user can change both the “zoom” of the page and, separately, the default font size. While “zooming” will increase the size of all elements (font included)

on the page, changing the default browser font size won't have any effect on fonts whose size in CSS is specified using absolute, rather than relative, units.



9.4. font-style

Currently, the only use for `font-style` is to italicize (and unitalicize) text. The values are listed below:

- `normal`
- `italic`
- `oblique`

However, `italic` and `oblique` are displayed in the same way. Since `italic` has better support, you should use it. For more on `font-style`, see <https://developer.mozilla.org/en-US/docs/Web/CSS/font-style>.



9.5. font-variant

The most common and well-supported use of `font-variant` is to turn lowercase letters into small caps. The values are listed below:

- `normal`
- `small-caps`

The screenshot below shows an `h1` tag with `small-caps`:³⁷

³⁷. The file used for this screenshot is `Fonts/Demos/font-variant.html`.



`font-variant` is actually a shorthand property (more on shorthand properties soon) for several longhand properties, but those are rarely used and poorly supported, so you shouldn't worry about them. For more details, see <https://developer.mozilla.org/en-US/docs/Web/CSS/font-variant>.

*Evaluation
Copy*

9.6. font-weight

The weight of a font determines how thick (or bold) it is. Possible values are:

- Any number between 1 and 1000
- **bold**
- **bolder**
- **lighter**
- **normal**

The numeric values are used for fonts that can have many different degrees of boldness. Most fonts, however, are either **bold** or **normal** (not bold). **bold** is the same as 700 and **normal** is the same as 400. **bolder** and **lighter** work similarly to larger and smaller for font-size.

Here is a stylesheet with rules setting font-weight for elements on the page:

Demo 9.7: Fonts/Demos/font-weight-styles.css

```
1.  html {
2.      font-family:'Segoe UI', 'Open Sans',
3.      'Helvetica Neue', sans-serif;
4.  }
5.  #div1 {
6.      font-weight: normal;
7.  }
8.  #div2 {
9.      font-weight: bold;
10. }
11. #div3 {
12.     font-weight: 400;
13. }
14. #div4 {
15.     font-weight: 1000;
16. }
17. .bolder {
18.     font-weight: bolder;
19. }
20. .lighter {
21.     font-weight: lighter;
22. }
23. #li1 {
24.     font-weight: 100;
25. }
26. #li2 {
27.     font-weight: 400;
28. }
29. #li3 {
30.     font-weight: 600;
31. }
32. #li4 {
33.     font-weight: 700;
34. }
35. #li5 {
36.     font-weight: 900;
37. }
```

Evaluation
Copy

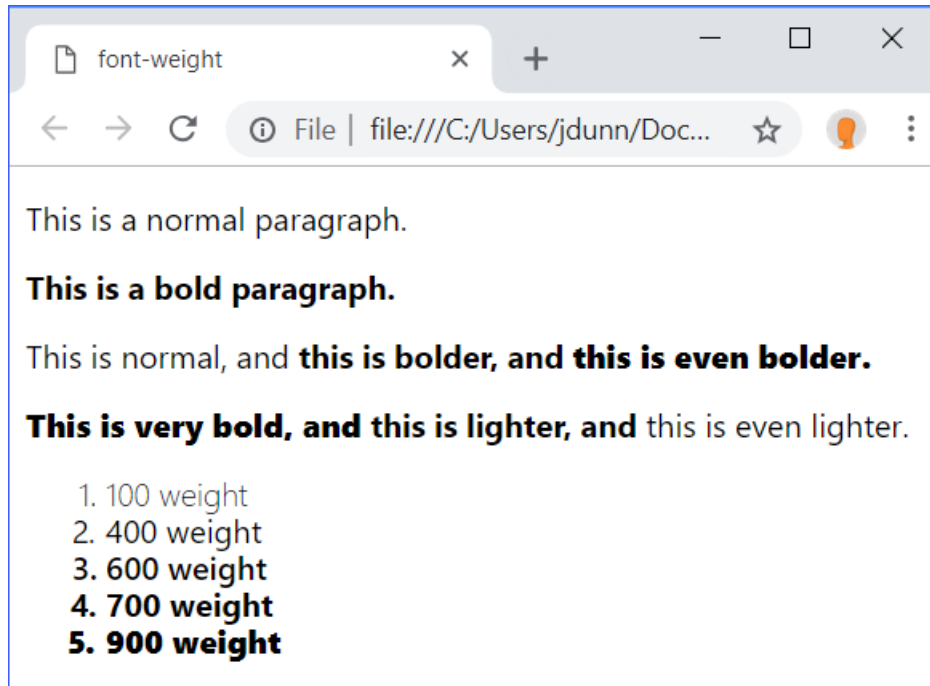
We used the Segoe UI font because it has precise control over its weight. If we were to use a font that didn't have this control, like Times, then each element would only render either bold or normal.

And here is an HTML page that uses this stylesheet:

Demo 9.8: Fonts/Demos/font-weight.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <div id="div1">
12.     <p>This is a normal paragraph.</p>
13.   </div>
14.   <div id="div2">
15.     <p>This is a bold paragraph.</p>
16.   </div>
17.   <div id="div3">
18.     <p>This is normal, and
19.       <span class="bolder">this is bolder, and
20.         <span class="bolder">this is even bolder.
21.           </span>
22.         </span>
23.       </p>
24.   </div>
25.   <div id="div4">
26.     <p>This is very bold, and
27.       <span class="lighter">this is lighter, and
28.         <span class="lighter">this is even lighter.
29.           </span>
30.         </span>
31.       </p>
32.   </div>
33.   <ol>
34.     <li id="li1">100 weight</li>
35.     <li id="li2">400 weight</li>
36.     <li id="li3">600 weight</li>
37.     <li id="li4">700 weight</li>
38.     <li id="li5">900 weight</li>
39.   </ol>
40. </body>
41. </html>
```

The code above will render the following:



font-weight Values

In early versions of font-weight, only intervals of 100 (100 to 900) were accepted, so if you use a more exact number value, make sure both the font you're using and your target browsers support that value. For more on font-weight, see <https://developer.mozilla.org/en-US/docs/Web/CSS/font-weight>.



9.7. line-height

line-height determines the amount of vertical space used for lines, most commonly in text.

Although there are several options for line-height values, the best option is to use a unit-less number. This number is relative to the font-size of the text. For example, if you set line-height to 1, there will be no space between the text on a line and the text on lines before and after that line.

The default line-height value is approximately 1.2. You can set that specifically using:

```
line-height: 1.2;
```

Or you can set it using:

```
line-height: normal;
```

The latter will set the `line-height` to the exact default value, which is dependent on the browser and the font.

To add more space between lines, set `line-height` to a value higher than 1.2 and to make lines tighter, set it to a value less than 1.2.

Other Values of line-height

While `line-height` can also take a length with units (e.g., `em` or `px`), we recommend you stick with a unit-less number.

The following samples shows some different line heights:

Demo 9.9: Fonts/Demos/line-height-styles.css

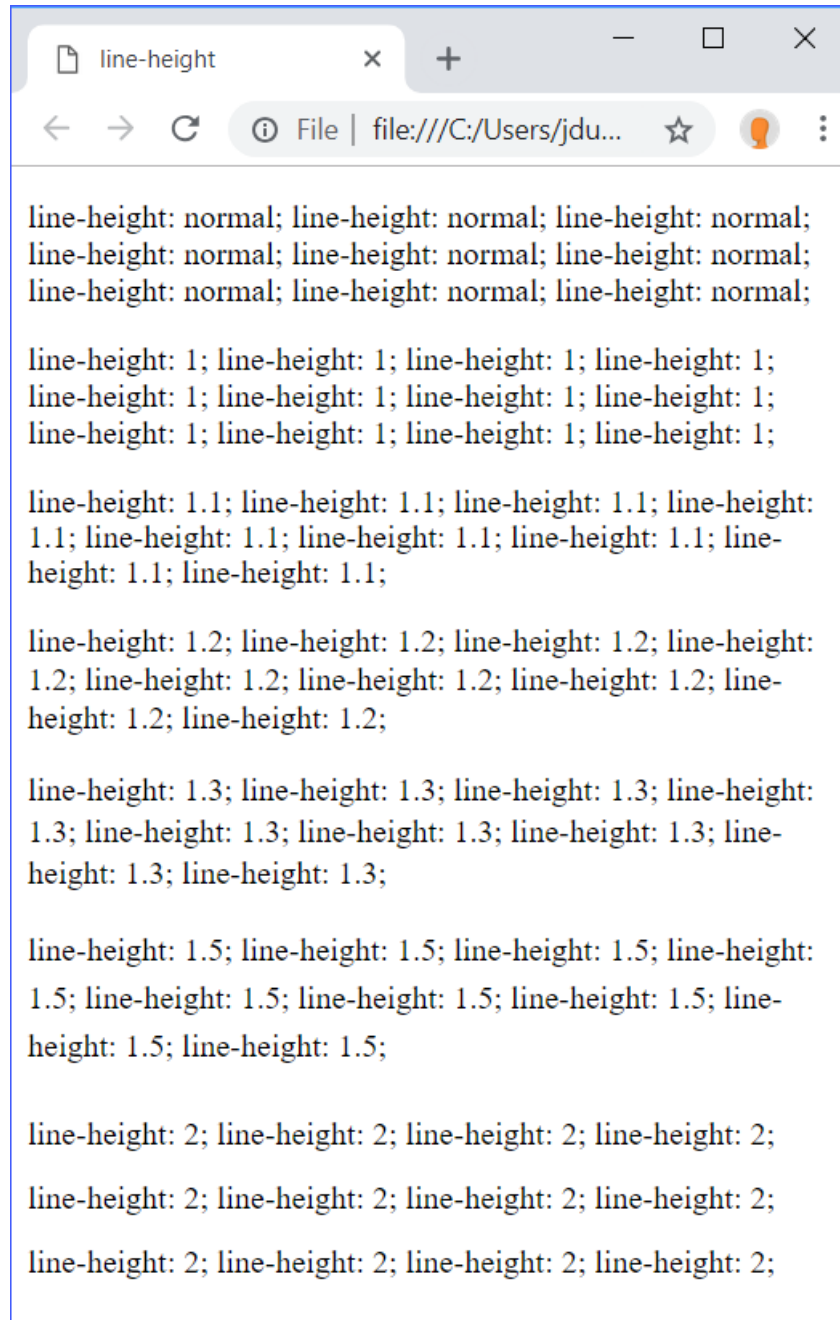
```
1.  html {
2.    font-size: 16px;
3.  }
4.
5.  #normal {
6.    line-height: normal;
7.  }
8.
9.  #p1 {
10.   line-height: 1;
11. }
12.
13. #p2 {
14.   line-height: 1.1;
15. }
16.
17. #p3 {
18.   line-height: 1.2;
19. }
20.
21. #p4 {
22.   line-height: 1.3;
23. }
24.
25. #p5 {
26.   line-height: 1.5;
27. }
28.
29. #p6 {
30.   line-height: 2;
31. }
```

Evaluation
Copy

Demo 9.10: Fonts/Demos/line-height.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <p id="normal">
12.     line-height: normal; line-height: normal; line-height: normal;
13.     line-height: normal; line-height: normal; line-height: normal;
14.     line-height: normal; line-height: normal; line-height: normal;
15.   </p>
16.   <p id="p1">
17.     line-height: 1; line-height: 1; line-height: 1; line-height: 1;
18.     line-height: 1; line-height: 1; line-height: 1; line-height: 1;
19.     line-height: 1; line-height: 1; line-height: 1; line-height: 1;
20.   </p>
21.   <p id="p2">
22.     line-height: 1.1; line-height: 1.1; line-height: 1.1;
23.     line-height: 1.1; line-height: 1.1; line-height: 1.1;
24.     line-height: 1.1; line-height: 1.1; line-height: 1.1;
25.   </p>
26.   <p id="p3">
27.     line-height: 1.2; line-height: 1.2; line-height: 1.2;
28.     line-height: 1.2; line-height: 1.2; line-height: 1.2;
29.     line-height: 1.2; line-height: 1.2; line-height: 1.2;
30.   </p>
31.   <p id="p4">
32.     line-height: 1.3; line-height: 1.3; line-height: 1.3;
33.     line-height: 1.3; line-height: 1.3; line-height: 1.3;
34.     line-height: 1.3; line-height: 1.3; line-height: 1.3;
35.   </p>
36.   <p id="p5">
37.     line-height: 1.5; line-height: 1.5; line-height: 1.5;
38.     line-height: 1.5; line-height: 1.5; line-height: 1.5;
39.     line-height: 1.5; line-height: 1.5; line-height: 1.5;
40.   </p>
41.   <p id="p6">
42.     line-height: 2; line-height: 2; line-height: 2; line-height: 2;
43.     line-height: 2; line-height: 2; line-height: 2; line-height: 2;
44.     line-height: 2; line-height: 2; line-height: 2; line-height: 2;
45.   </p>
46. </body>
47. </html>
```

The code above will render the following:



For more on line-height, see <https://developer.mozilla.org/en-US/docs/Web/CSS/line-height>.



9.8. font

`font` is a shorthand property that encompasses the properties we have discussed in this lesson.

Shorthand Properties

Shorthand properties set multiple CSS properties at once, allowing for more concise, more readable stylesheets that save time and space. Using shorthand properties instead of longhand properties also has the benefit of reducing file size, which increases download speed.

An important thing to note about shorthand properties is that any unspecified values will be set to their default values.

The syntax for `font` is shown below. The highlighted properties are required:

```
font: font-style  
      font-variant  
      font-weight  
      font-stretch  
      font-size/line-height  
      font-family;
```

Evaluation
Copy

Some notes:

1. `font-stretch`, while well supported by browsers, only works with a small number of fonts.³⁸
2. `font-style`, `font-variant`, `font-weight`, and `font-stretch` are optional and may be written in any order before `font-size`.
3. `line-height` is also optional, but it must be written immediately after `font-size` separated by a forward slash (/).
4. If one of the optional values is not included, it will be set to the default value, overriding any rules with lower precedence.

The code below illustrates the use of `font`:

³⁸. See <https://developer.mozilla.org/en-US/docs/Web/CSS/font-stretch>.

Demo 9.11: Fonts/Demos/font-styles.css

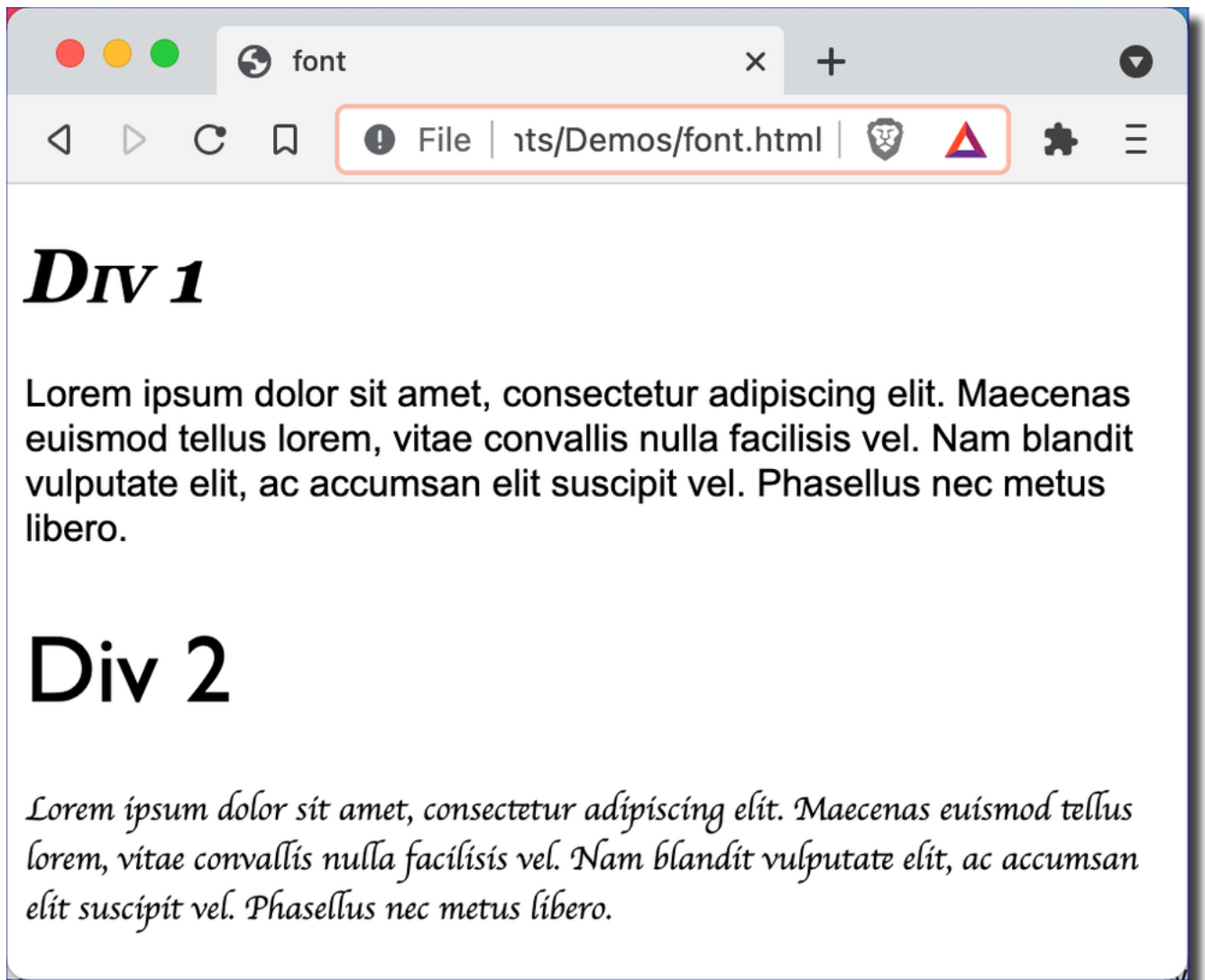
```
1.  #div1 h1 {
2.      font: italic small-caps bold 2em Georgia, Times,
3.      'Times New Roman', sans-serif;
4.  }
5.
6.  #div1 p {
7.      font: 1em/1.2 Arial, Helvetica, sans-serif;
8.  }
9.
10. #div2 h1 {
11.     font-style: italic; /* gets overridden */
12.     font-variant: small-caps; /* gets overridden */
13.     font: 2.5em 'Gill Sans', serif;
14. }
15.
16. #div2 p {
17.     font: 0.9em/1.5 cursive;
18. }
```

Notice that the `font-style` and `font-variant` declarations written in the `#div2 h1` rule will not be applied because they were later overridden by defaults in the `font` declaration, even though those defaults are not explicitly stated.

Demo 9.12: Fonts/Demos/font.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <div id="div1">
12.     <h1>Div 1</h1>
13.     <p>
14.       Lorem ipsum dolor sit amet, consectetur adipiscing elit.
15.       Maecenas euismod tellus lorem, vitae convallis nulla
16.       facilisis vel. Nam blandit vulputate elit, ac accumsan
17.       elit suscipit vel. Phasellus nec metus libero.
18.     </p>
19.   </div>
20.   <div id="div2">
21.     <h1>Div 2</h1>
22.     <p>
23.       Lorem ipsum dolor sit amet, consectetur adipiscing elit.
24.       Maecenas euismod tellus lorem, vitae convallis nulla
25.       facilisis vel. Nam blandit vulputate elit, ac accumsan
26.       elit suscipit vel. Phasellus nec metus libero.
27.     </p>
28.   </div>
29. </body>
30. </html>
```

The code above will render the following:



For more on font, see <https://developer.mozilla.org/en-US/docs/Web/CSS/font>.



Exercise 12: Styling Fonts

⌚ 25 to 40 minutes

In this exercise, you will modify an HTML page by applying the font properties you just learned.

1. Open `Fonts/Exercises/index.html` in your editor. You will see the home page for a website called Runners Home.
2. Create an external stylesheet called `styles.css` and link to it from `index.html`.
3. Using your new knowledge of font properties, add some styles to this page. The object of this exercise is to practice using the font properties covered in this lesson. You are also welcome to download a font and use `@font-face`.
4. When you are done, open `index.html` in your browser to see the results. You are welcome to go back to the code and continue to work.

You can design it however you like, or you can try to make it look something like this:

Runners Home

- [Home](#)
- [Resources](#)
 - [Calculator](#)
 - [Running Log](#)
 - [Running Terms](#)
 - [Links](#)
- [Running Advice](#)
- [Races](#)
- [Register](#)

Hello, Stranger!

Welcome to Runners Home

BEST RUNNING TIPS

There are good ways to train and bad ways to train. To great the most out of your runs... [Read more...](#)

BEST HEALTH TIPS

In addition to keeping up with your physical training, you must be sure to take care of your general health... [Read more...](#)

[More advice articles...](#)

Purpose

Runners Home is dedicated to providing you with:

1. [the most up-to-date information on running races.](#)
2. [the best resources for runners.](#)

Newsletter

Email:

Be the first to hear about our great offers.
Sign up for our newsletter today!


Disclosure: This is not a real website.

© 2022 Runners Home. All rights reserved.

info@runners-home.com

Solution: Fonts/Solutions/styles.css

```
1.  html {
2.      font-size: 16px;
3.  }
4.
5.  body {
6.      font: 1rem/1.3 Verdana, Geneva, Tahoma, sans-serif;
7.  }
8.
9.  #welcome {
10.     font-size: larger;
11. }
12.
13. article h2 {
14.     font-variant: small-caps;
15. }
16.
17. article p {
18.     line-height: 2;
19. }
20.
21. aside label {
22.     font-weight: bold;
23. }
24.
25. footer {
26.     font-size: smaller;
27.     font-style: italic;
28. }
```



Conclusion

In this lesson, you have learned to use CSS font properties.

LESSON 10

Color and Opacity

Topics Covered

☒ color

☒ opacity

Introduction

In this lesson, you will learn to add color and opacity to your HTML pages.



10.1. About Color and Opacity

CSS makes it really easy to add color to your HTML pages. Almost every element can have color added to it, and CSS provides multiple methods to do so. For a comprehensive list of things in CSS that can have color, see Mozilla's list of Things that can have color³⁹.

The opacity of an element is its level of opaqueness: an element with zero opacity is fully transparent. Everything that can be colored, can also have its opacity changed, and opacity can additionally affect images, which colors do not affect. Opacity and color go hand in hand, and there are several ways to apply color and opacity at the same time.



10.2. Color and Opacity Values

There are two main color models used to create colors in CSS. The first is the RGB color model, which combines **red**, **green**, and **blue** in different amounts to create a wide range of colors. The second is the HSL color model, which creates colors based on **hue**, **lightness**, and **saturation**. RGB is more popular and more commonly used.

³⁹. https://developer.mozilla.org/en-US/docs/Web/HTML/Applying_color#Things_that_can_have_color

Picking RGB and HSL Colors

Guessing RGB and HSL colors by their notations, which are explained below, is extremely difficult, and thankfully, unnecessary. There are plenty of excellent tools for selecting RGB and HSL colors from color palettes, such as Mozilla's Color picker tool⁴⁰.

Opacity is most commonly expressed as a percentage from 0% (invisible) to 100% (opaque), and/or a number from 0.0 (invisible) to 1.0 (opaque).

❖ 10.2.1. Color Keywords

CSS recognizes over a hundred color keywords as acceptable color values. Examples include standard colors you would expect, such as blue, green, red, black, silver, and white; and some more fun color names, such as blanchedalmond, firebrick, floralwhite, and lawngreen. For a full list of accepted color keywords see Mozilla's List of color keywords⁴¹

❖ 10.2.2. RGB Hexadecimal Notation

RGB hexadecimal notation is the most commonly used type of color value. Hexadecimal digits are used to represent the amounts of each color component (red, green, and blue) ranging from 0 (00) to 255 (ff). The syntax for hexadecimal notation is below:

```
#rrggbb /* longhand */  
#rgb /* shorthand */
```

Hexadecimal Numbers

The numbering system we are all used to is base 10. To make that work, we have ten single-digit characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

The hexadecimal system uses base 16. So, there are sixteen single-digit characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f.

In base 10, 90 is 10 greater than 80. In base 16, 90 is 16 greater than 80, and a0 is 16 greater than 90. You count it like this:

⁴⁰. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Colors/Color_picker_tool

⁴¹. https://developer.mozilla.org/en-US/docs/Web/CSS/color_value#Color_keywords

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
20	21	22	23	24	25	26	27	28	29	2a	2b	2c	2d	2e	2f
30	31	32	33	34	35	36	37	38	39	3a	3b	3c	3d	3e	3f
40	41	42	43	44	45	46	47	48	49	4a	4b	4c	4d	4e	4f
50	51	52	53	54	55	56	57	58	59	5a	5b	5c	5d	5e	5f
60	61	62	63	64	65	66	67	68	69	6a	6b	6c	6d	6e	6f
70	71	72	73	74	75	76	77	78	79	7a	7b	7c	7d	7e	7f
80	81	82	83	84	85	86	87	88	89	8a	8b	8c	8d	8e	8f
90	91	92	93	94	95	96	97	98	99	9a	9b	9c	9d	9e	9f
a0	a1	a2	a3	a4	a5	a6	a7	a8	a9	aa	ab	ac	ad	ae	af
b0	b1	b2	b3	b4	b5	b6	b7	b8	b9	ba	bb	bc	bd	be	bf
c0	c1	c2	c3	c4	c5	c6	c7	c8	c9	ca	cb	cc	cd	ce	cf
d0	d1	d2	d3	d4	d5	d6	d7	d8	d9	da	db	dc	dd	de	df
e0	e1	e2	e3	e4	e5	e6	e7	e8	e9	ea	eb	ec	ed	ee	ef
f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	fa	fb	fc	fd	fe	ff

In shorthand hexadecimal notation, the six digits are cut down to three, and each digit represents two of itself. For example, #f03 would be #ff0033. Both of those notations mean that the color is made from ff of red, 00 of green, and 33 of blue. The result is sort of a rose color.

❖ 10.2.3. RGB Functional Notation

RGB colors can also be expressed in `rgb()` functional notation, which takes three parameters (red, green, and blue). These can either be integers from 0 to 255 or percentages from 0% to 100%. The syntax for RGB functional notation is:

```
rgb(red, green, blue)
```

❖ 10.2.4. HSL Functional Notation

HSL colors are expressed using the `hsl()` function notation. The `hsl()` function notation takes three parameters:

1. **Hue** - An angle of the color circle in degrees (deg), radians (rad), gradians (grad), or turns (turn). If units are omitted, deg are assumed.

A. red = 0° or 360°

B. green = 120°

C. blue = 240°

2. **Saturation** - A percentage where 100% is completely saturated (full color) and 0% is a shade of gray.
3. **Lightness** - A percentage where 100% is white, 50% is “normal” lightness, and 0% is black.

The syntax for `hsl()` function notation is:

```
hsl(hue, saturation, lightness)
```

`rgba()`, `hsla()`, and `#rgba`

Both RGB and HSL functional notations have ways to specify opacity (alpha) at the same time as color. Along with their respective normal notations (`rgb()` and `hsl()`), they also have alpha notations (`rgba()` and `hsla()`) that accept a fourth parameter: `alpha`, which is expressed either as a percentage from 0% to 100% or a number between 0.0 and 1.0. These are both well-supported in modern browsers.

In modern browsers (i.e., not IE), the `rgb()` and `rgba()` notations are interchangeable, meaning that `rgb()` can also take the `alpha` parameter.

You can also specify opacity using hexadecimal notation by adding the hexadecimal value for opacity at the end. For example: `#ff0033aa`.



10.3. color

`color` is the CSS property used to apply color to the *foreground* of an element. The most common example of a “foreground” is just plain old text. See the samples below:

Demo 10.1: ColorAndOpacity/Demos/color-styles.css

```
1.  #keyword {
2.      color: red;
3.  }
4.
5.  #rgb-longhand {
6.      color: #ff0000;
7.  }
8.
9.  #rgb-shorthand {
10.     color: #f00;
11. }
12.
13. #rgb-function-numbers {
14.     color: rgb(255, 0, 0);
15. }
16.
17. #rgb-function-percentage {
18.     color: rgb(100%, 0%, 0%);
19. }
20.
21. #rgba {
22.     color: rgba(255, 0, 0, 0.5);
23. }
24.
25. #hsl-function {
26.     color: hsl(0, 100%, 50%);
27. }
28.
29. #hsla {
30.     color: hsla(0, 100%, 50%, 50%);
31. }
```

Evaluation
Copy

Demo 10.2: ColorAndOpacity/Demos/color-demo.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <h1>color</h1>
12.   <div id="keyword">color: red;</div>
13.   <div id="rgb-longhand">color: #ff0000;</div>
14.   <div id="rgb-shorthand">color: #f00;</div>
15.   <div id="rgb-function-numbers">color: rgb(255, 0, 0);</div>
16.   <div id="rgba">color: rgba(255, 0, 0, 0.5);</div>
17.   <div id="rgb-function-percentage">color: rgb(100%, 0%, 0%);</div>
18.   <div id="hsl-function">color: hsl(0, 100%, 50%);</div>
19.   <div id="hsla">color: hsla(0, 100%, 50%, 50%);</div>
20. </body>
21. </html>
```

The code above will render the following:⁴²



42. If you are reading this in black and white, be sure to open the page in your browser.

10.4. opacity

opacity is the CSS property used to set the transparency of an entire element. See the demo below:

Demo 10.3: ColorAndOpacity/Demos/opacity-styles.css

```
1.  #my-div {
2.    /* To make the changes in opacity
3.    easier to see, we have added a yellow
4.    background color and made the text red */
5.    background-color: yellow;
6.    color: red;
7.  }
8.
9.  .invisible {
10.   opacity: 0;
11.  }
12.
13.  .light {
14.   opacity: .25;
15.  }
16.
17.  .medium {
18.   opacity: .5;
19.  }
20.
21.  .heavy {
22.   opacity: .75;
23.  }
24.
25.  .opaque {
26.   opacity: 1;
27.  }
```

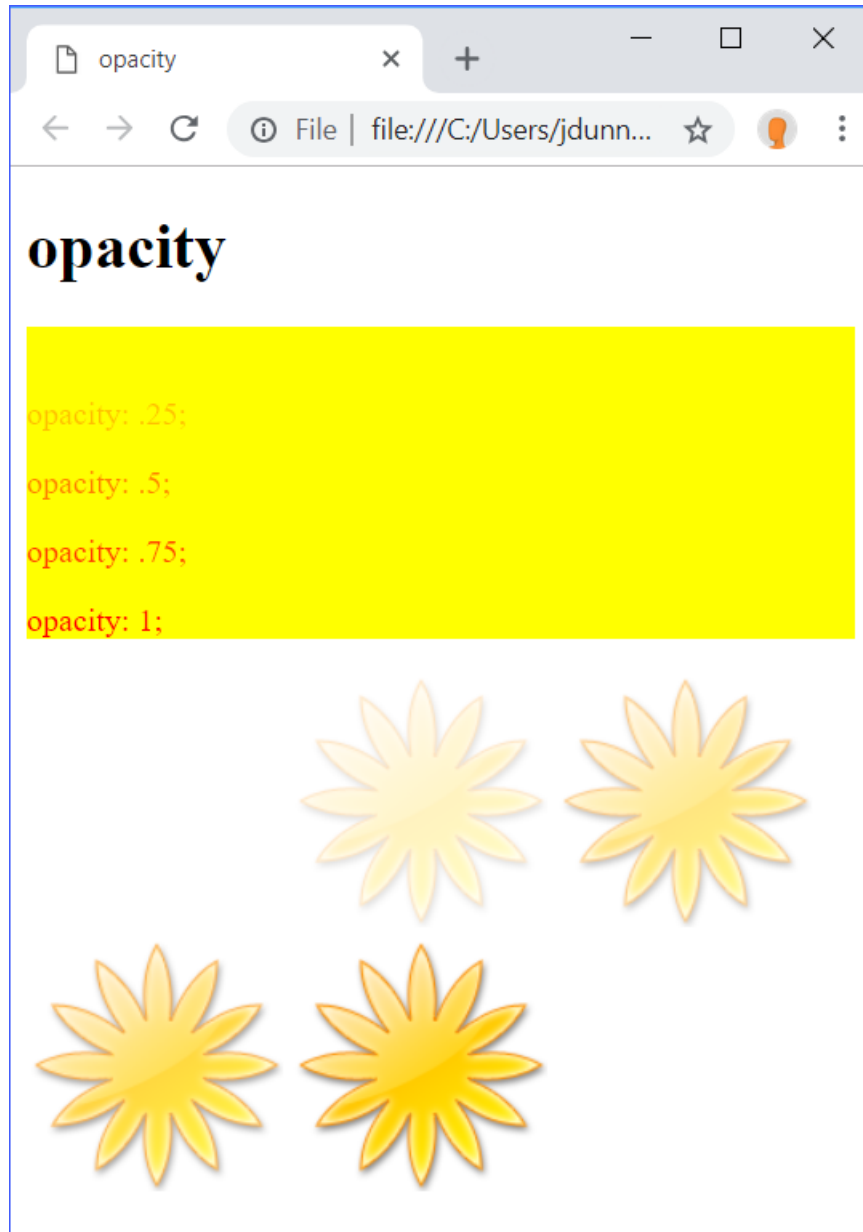
Evaluation
Copy

In this demo, opacity was also used to change the transparency of images. This is something that the color property cannot do as images are not affected by the color property.

Demo 10.4: ColorAndOpacity/Demos/opacity-demo.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <h1>opacity</h1>
12.   <div id="my-div">
13.     <p class="invisible">opacity: 0;</p>
14.     <p class="light">opacity: .25;</p>
15.     <p class="medium">opacity: .5;</p>
16.     <p class="heavy">opacity: .75;</p>
17.     <p class="opaque">opacity: 1;</p>
18.   </div>
19.   
20.   
21.   
22.   
23.   
24. </body>
25. </html>
```

The code above will render the following:



Exercise 13: Adding Color and Opacity to Text

⌚ 25 to 40 minutes

In this exercise, you will add color and opacity properties to an HTML file containing three children's stories.

1. Open Exercises/stories.html in your editor.
2. Create a CSS file called stories-styles.css and link to it from stories.html.
3. Using CSS, edit the color and opacity of the different elements in stories.html. The object of this exercise is to practice using the color and opacity properties.
4. When you are done, open stories.html in your browser. You are welcome to go back to the code and keep working.

You can design it however you like, or you can try to make it look something like: ⁴³

^{43.} The images used in this section are in the public domain (https://commons.wikimedia.org/wiki/Public_domain):

- https://commons.wikimedia.org/wiki/File:Cinderella_and_the_Fairy_Godmother.jpg
- https://commons.wikimedia.org/wiki/File:Boys_and_Girls_of_Bookland_Alice_in_Wonderland.jpg
- https://commons.wikimedia.org/wiki/File:Joshua_Reynolds_-_Cupid_as_Link_Boy.jpg


Stories

← → ↻ 🏠 🔍 'CSS-101_print/ClassFiles/ColorAndOpacity/Solutions/stories.html' 📧 📺 📱 📄 📌 📂 📁 📅 📆 📇 📈 📉 📊 📋 📌 📍 📎 📏 📐 📑 📒 📓 📔 📕 📖 📗 📙 📚 📛 📜 📝 📞 📟 📠 📡 📢 📣 📤 📥 📦 📧 📨 📩 📪 📫 📬 📭 📮 📯 📰 📱 📲 📳 📴 📵 📶 📷 📸 📹 📺 📻 📼 📽 📾 📿 📰 📱 📲 📳 📴 📵 📶 📷 📸 📹 📺 📻 📼 📽 📾 📿

Stories

- [A Mad Tea-Party](#)
- [Cinderella](#)
- [The Naughty Boy](#)

A MAD TEA-PARTY - Lewis Carroll



There was a table set out under a tree in front of the house, and the March Hare and the Hatter were having tea at it: a Dormouse was sitting between them, fast asleep, and the other two were using it as a cushion, resting their elbows on it, and talking over its head. "Very uncomfortable for the Dormouse," thought Alice; "only, as it's asleep, I suppose it doesn't mind." [Continue reading](#)

[Back to top](#)

Solution: ColorAndOpacity/Solutions/stories-styles.css

```
1.  a {
2.    color: red;
3.  }
4.
5.  a[href='#top'],
6.  h1 {
7.    color: limegreen;
8.  }
9.
10. nav a {
11.   color: hsl(300, 20%, 50%);
12. }
13.
14. h2 {
15.   font-size: 1.2em;
16.   font-style: italic;
17.   font-weight: normal;
18. }
19.
20. #cinderella h2 {
21.   color: hsl(56, 100%, 45%);
22. }
23.
24. #alice h2 {
25.   color: #f60;
26. }
27.
28. #naughtyboy h2 {
29.   color: #e96df2;
30. }
31.
32. p {
33.   color: rgb(51, 102, 102);
34.   font-family: Cambria, Cochin, Georgia, Times, serif;
35.   font-size: .9em;
36. }
37.
38. img {
39.   opacity: .5;
40. }
```

Conclusion

In this lesson, you have learned to work with color and opacity in CSS.

Evaluation
Copy

LESSON 11

CSS Text

Topics Covered

- ☒ letter-spacing
- ☒ text-align
- ☒ text-decoration
- ☒ text-indent
- ☒ text-shadow
- ☒ text-transform
- ☒ white-space
- ☒ word-break
- ☒ word-spacing

Evaluation
Copy

Introduction

In this lesson, you will learn to use CSS properties for formatting text.



11.1. letter-spacing

The `letter-spacing` property is used to specify the amount of space between letters. The amount indicated is in addition to the default spacing. The property either takes the keyword `normal`, which is the default defined by the current font and/or browser, or a length value. See the following example:

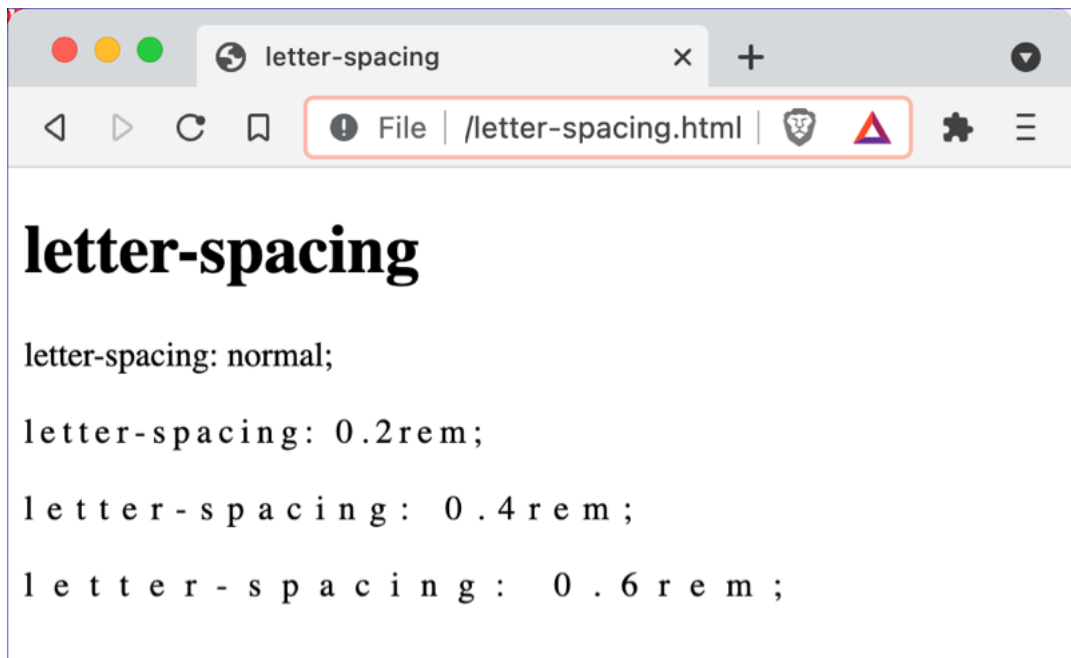
Demo 11.1: CssText/Demos/letter-spacing-styles.css

```
1.  #normal {
2.      letter-spacing: normal;
3.  }
4.
5.  #wide {
6.      letter-spacing: 0.2rem;
7.  }
8.
9.  #wider {
10.     letter-spacing: 0.4rem;
11. }
12.
13. #widest {
14.     letter-spacing: 0.6rem;
15. }
```

Demo 11.2: CssText/Demos/letter-spacing.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.     <h1>letter-spacing</h1>
12.     <div id="normal">letter-spacing: normal;</div>
13.     <br>
14.     <div id="wide">letter-spacing: 0.2rem;</div>
15.     <br>
16.     <div id="wider">letter-spacing: 0.4rem;</div>
17.     <br>
18.     <div id="widest">letter-spacing: 0.6rem;</div>
19. </body>
20. </html>
```

The code above will render the following:



ENCLOSURE



11.2. text-align

The `text-align` property is used to specify how inline content should be aligned horizontally within a block. The values are listed below:

- left
- right
- center
- justify

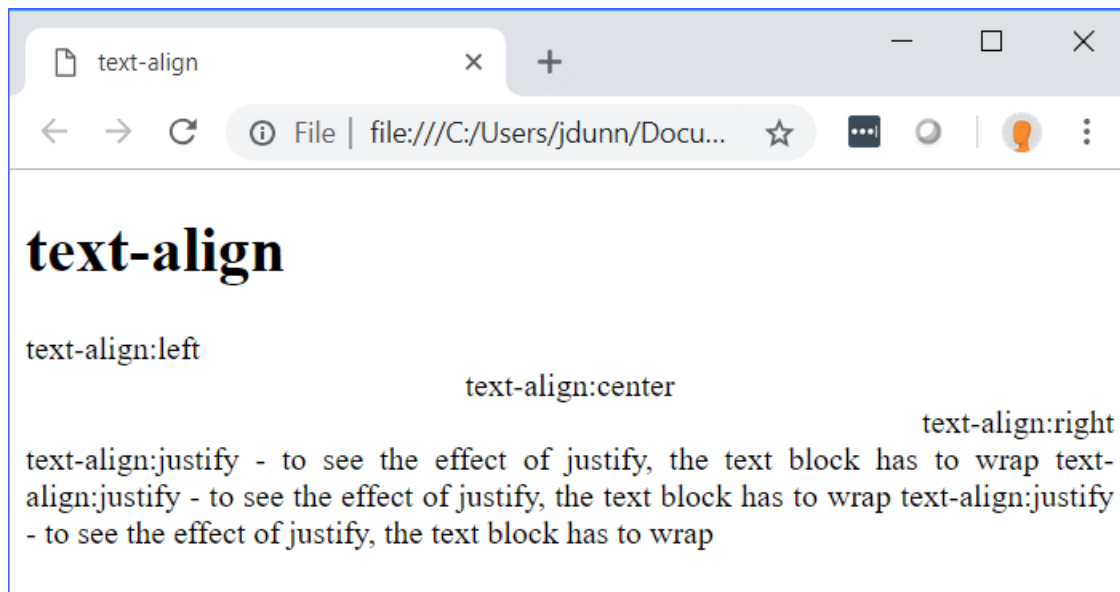
Demo 11.3: CssText/Demos/text-align-styles.css

```
1.  .left {
2.      text-align: left;
3.  }
4.
5.  .center {
6.      text-align: center;
7.  }
8.
9.  .right {
10.     text-align: right;
11. }
12.
13. .justify {
14.     text-align: justify;
15. }
```

Demo 11.4: CssText/Demos/text-align.html

```
-----Lines 1 through 8 Omitted-----
9.  <body class="webucator">
10. <h1>text-align</h1>
11. <div class="left">text-align:left</div>
12. <div class="center">text-align:center</div>
13. <div class="right">text-align:right</div>
14. <div class="justify">
15.     text-align:justify - to see the effect of justify,
16.         the text block has to wrap
17.     text-align:justify - to see the effect of justify,
18.         the text block has to wrap
19.     text-align:justify - to see the effect of justify,
20.         the text block has to wrap
21. </div>
22. </body>
23. </html>
```

The code above will render the following:



11.3. text-decoration

Evaluation
Copy

The `text-decoration` property is used to add effects to text, such as underlines and line-throughs. The values are listed below:

- `none`
- `underline`
- `overline`
- `line-through`

The `none` value of the `text-decoration` property can be used to remove the underline from links, as shown below:

```
a {  
  text-decoration: none;  
}
```

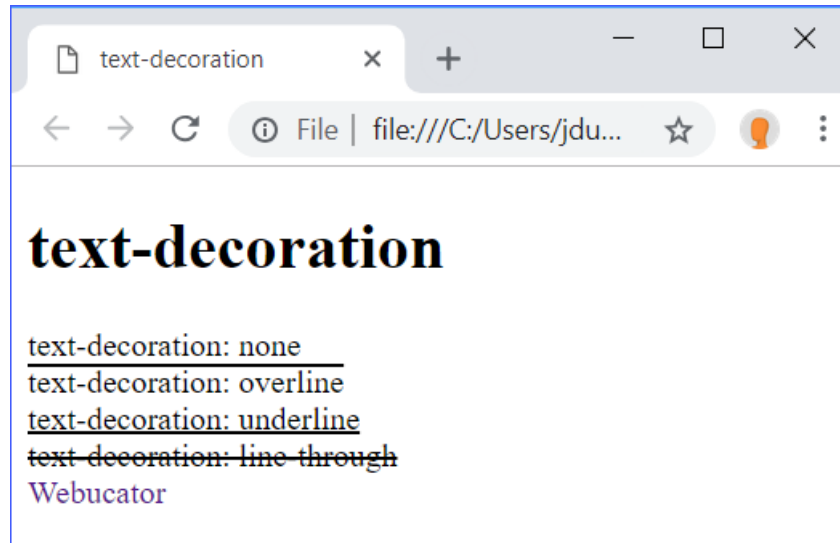
Demo 11.5: CssText/Demos/text-decoration-styles.css

```
1.  .none {
2.      text-decoration: none;
3.  }
4.
5.  .overline {
6.      text-decoration: overline;
7.  }
8.
9.  .underline {
10.     text-decoration: underline;
11.  }
12.
13.  .line-through {
14.     text-decoration: line-through;
15.  }
```

Demo 11.6: CssText/Demos/text-decoration.html

```
-----Lines 1 through 8 Omitted-----
9.  <body class="webucator">
10.     <h1>text-decoration</h1>
11.     <div class="none">text-decoration: none</div>
12.     <div class="overline">text-decoration: overline</div>
13.     <div class="underline">text-decoration: underline</div>
14.     <div class="line-through">text-decoration: line-through</div>
15.     <div><a href="https://www.webucator.com" class="none">
16.         Webucator</a>
17.     </div>
18. </body>
19. </html>
```

The code above will render the following:



Note how we have removed the underline from the link using `text-decoration: none`.

`text-decoration` is actually a shorthand property for the following longhand properties:

1. `text-decoration-line`: Values include `none`, `underline`, `overline`, and `line-through`.
2. `text-decoration-color`: Any valid color.
3. `text-decoration-style`: Values include `solid`, `double`, `dotted`, `dashed`, and `wavy`.
4. `text-decoration-thickness`: Values can be a length (e.g., `3px`), a percentage (e.g., `10%`), or a keyword (e.g., `auto` or `from-font`).

For more details on `text-decoration`, see <https://developer.mozilla.org/en-US/docs/Web/CSS/text-decoration>.



11.4. text-indent

The `text-indent` property is used to indent (or outdent) the first line of a block of text. The value can be specified in number of units or in percentage of the width of the containing block.

The following code sample shows the effects of `text-indent`:

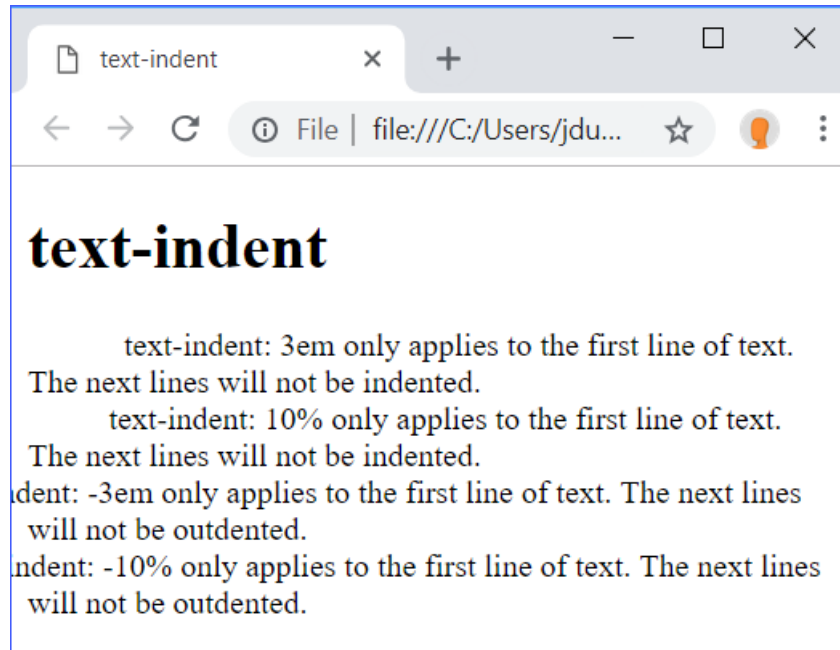
Demo 11.7: CssText/Demos/text-indent-styles.css

```
1.  .indent-length {
2.      text-indent: 3em;
3.  }
4.
5.  .indent-percentage {
6.      text-indent: 10%;
7.  }
8.
9.  .outdent-length {
10.     text-indent: -3em;
11. }
12.
13. .outdent-percentage {
14.     text-indent: -10%;
15. }
```

Demo 11.8: CssText/Demos/text-indent.html

```
-----Lines 1 through 8 Omitted-----
9.  <body class="webucator">
10.     <h1>text-indent</h1>
11.     <div class="indent-length">
12.         text-indent: 3em only applies to the first line of text.
13.         The next lines will not be indented.
14.     </div>
15.     <div class="indent-percentage">
16.         text-indent: 10% only applies to the first line of text.
17.         The next lines will not be indented.
18.     </div>
19.     <div class="outdent-length">
20.         text-indent: -3em only applies to the first line of text.
21.         The next lines will not be outdented.
22.     </div>
23.     <div class="outdent-percentage">
24.         text-indent: -10% only applies to the first line of text.
25.         The next lines will not be outdented.
26.     </div>
27. </body>
28. </html>
```

The code above will render the following:



Note how the outdenting moves some of the content out of the viewport. Normally, you would add padding on the left to prevent this from happening. We will learn how to do that in a later lesson.

11.5. text-shadow

The `text-shadow` property is used to add shadow to text. It can take a comma-delimited list of shadows, each described by a combination of x and y offsets, blur radius, and color. The syntax is shown below:

```
text-shadow: x-offset y-offset blur-radius color /* shadow1 */,
             x-offset y-offset blur-radius color /* shadow2 */,
             x-offset y-offset blur-radius color /* shadow3 */
```

The values are explained below:

1. `x-offset` and `y-offset` – Required length values. `x-offset` must come immediately before `y-offset`. `x-offset` specifies the horizontal displacement (positive values to the right and negative values to the left). `y-offset` specifies the vertical displacement (positive values below and negative values above). If both values are zero, the shadow will be directly behind the text.

2. **blur-radius** – An optional length value that specifies the size of the blur effect. If included, this value must come after the two offset values.
3. **color** – An optional color value that has to be either the first value included or the last value. If not included, the browser picks the color, so for consistency across browsers, it is a good idea to specify the shadow's color.

Demo 11.9: CssText/Demos/text-shadow-styles.css

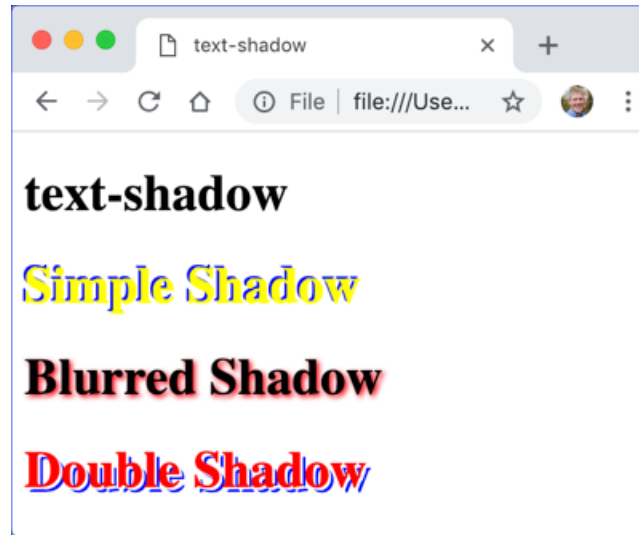
```
1.  #simple-shadow {
2.    color: yellow;
3.    text-shadow: -.1rem -.1rem blue;
4.  }
5.
6.  #blurred-shadow {
7.    text-shadow: .1rem .1rem .25rem red;
8.  }
9.
10. #double-shadow {
11.   color: red;
12.   text-shadow: .1rem .1rem white,
13.               .2rem .2rem blue;
14. }
```

Demo 11.10: CssText/Demos/text-shadow.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <h1>text-shadow</h1>
12.   <h1 id="simple-shadow">Simple Shadow</h1>
13.   <h1 id="blurred-shadow">Blurred Shadow</h1>
14.   <h1 id="double-shadow">Double Shadow</h1>
15. </body>
16. </html>
```

The code above will render the following:⁴⁴

44. If you are reading this in black and white, be sure to open the page in your browser to see the color effects.



11.6. text-transform

The `text-transform` property is used to change the capitalization of text. The most common values are listed below:

- none
- capitalize
- uppercase
- lowercase

The following code sample shows the effects of `text-transform`:

Demo 11.11: CssText/Demos/text-transform-styles.css

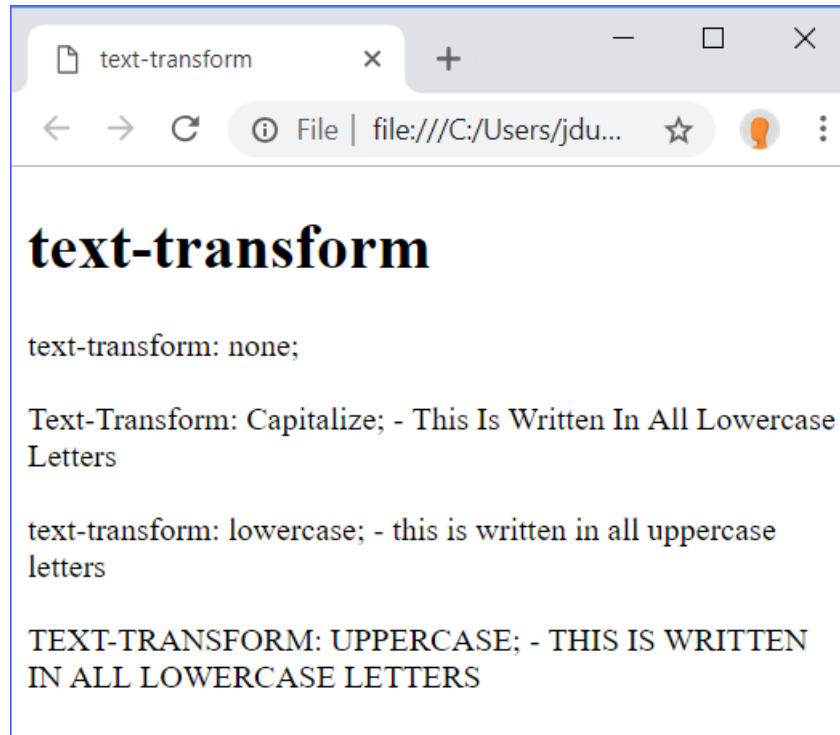
```
1.  #none {
2.      text-transform: none;
3.  }
4.
5.  #caps {
6.      text-transform: capitalize;
7.  }
8.
9.  #lower {
10.     text-transform: lowercase;
11. }
12.
13. #upper {
14.     text-transform: uppercase;
15. }
```

Demo 11.12: CssText/Demos/text-transform.html

-----Lines 1 through 9 Omitted-----

```
10. <body class="webucator">
11.     <h1>text-transform</h1>
12.     <div id="none">text-transform: none;</div><br>
13.     <div id="caps">text-transform: capitalize;
14.         - this is written in all lowercase letters
15.     </div><br>
16.     <div id="lower">text-transform: lowercase;
17.         - THIS IS WRITTEN IN ALL UPPERCASE LETTERS
18.     </div><br>
19.     <div id="upper">text-transform: uppercase;
20.         - this is written in all lowercase letters
21.     </div>
22. </body>
23. </html>
```

The code above will render the following:



Notice in the last two examples that the lowercase letters have been transformed to uppercase, and the uppercase letters have been transformed to lowercase.

Evaluation
Copy

*

11.7. white-space

The `white-space` property determines how sequences of whitespace are displayed. Below we list the most common values of the `white-space` property and their effects:

1. `normal`
 - Collapses adjacent spaces and tabs.
 - Collapses line breaks.
 - Wraps to fit containing box.
2. `nowrap`
 - Collapses adjacent spaces and tabs.
 - Collapses line breaks.
 - Does not wrap to fit containing box.

3. `pre`
 - Does not collapse adjacent spaces and tabs.
 - Does not collapse line breaks.
 - Does not wrap to fit containing box.
4. `pre-line`
 - Collapses adjacent spaces and tabs.
 - Does not collapse line breaks.
 - Wraps to fit containing box.
5. `pre-wrap`
 - Does not collapse adjacent spaces and tabs.
 - Does not collapse line breaks.
 - Wraps to fit containing box.

The following code sample shows the effects of `white-space`:

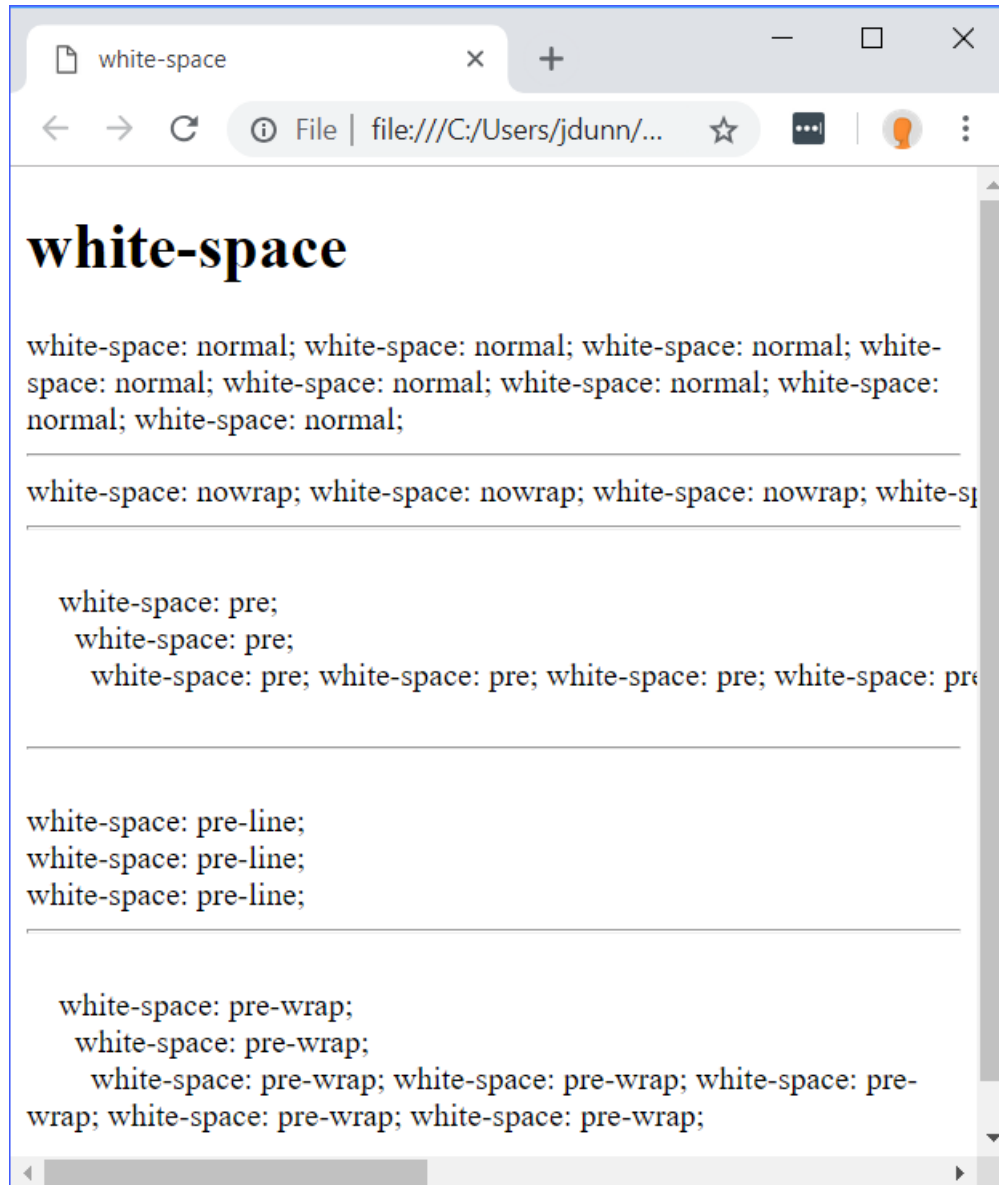
Demo 11.13: `CssText/Demos/white-space-styles.css`

```
1.  #normal {
2.      white-space: normal;
3.  }
4.
5.  #nowrap {
6.      white-space: nowrap;
7.  }
8.  #pre {
9.      white-space: pre;
10. }
11.
12. #pre-line {
13.     white-space: pre-line;
14. }
15.
16. #pre-wrap {
17.     white-space: pre-wrap;
18. }
```

Demo 11.14: CssText/Demos/white-space.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <h1>white-space</h1>
12.   <div id="normal">
13.     white-space: normal;
14.     white-space: normal;
15.     white-space: normal; white-space: normal; white-space: normal; white-space:
        normal;
16.     white-space: normal; white-space: normal;
17.   </div>
18.   <hr>
19.   <div id="nowrap">
20.     white-space: nowrap;
21.     white-space: nowrap;
22.     white-space: nowrap; white-space: nowrap; white-space: nowrap; white-space:
        nowrap;
23.     white-space: nowrap; white-space: nowrap;
24.   </div>
25.   <hr>
26.   <div id="pre">
27.     white-space: pre;
28.     white-space: pre;
29.     white-space: pre; white-space: pre; white-space: pre; white-space: pre;
        white-space: pre;
30.   </div>
31.   <hr>
32.   <div id="pre-line">
33.     white-space: pre-line;
34.     white-space: pre-line;
35.     white-space: pre-line;
36.   </div>
37.   <hr>
38.   <div id="pre-wrap">
39.     white-space: pre-wrap;
40.     white-space: pre-wrap;
41.     white-space: pre-wrap; white-space: pre-wrap; white-space: pre-wrap;
        white-space: pre-wrap; white-space: pre-wrap;
42.   </div>
43. </body>
44. </html>
```

The code above will render the following:



Notice that the user would need to scroll right to see the content that we have forced not to wrap.



11.8. word-break

The `word-break` property specifies where it is permissible to have line breaks when text would otherwise overflow its content box. The most common values are explained below:

1. `normal` -- default line breaks (at the end of words or at hyphens)
2. `break-all` -- line breaks can happen between any two characters

The following example shows the effects of word-break:

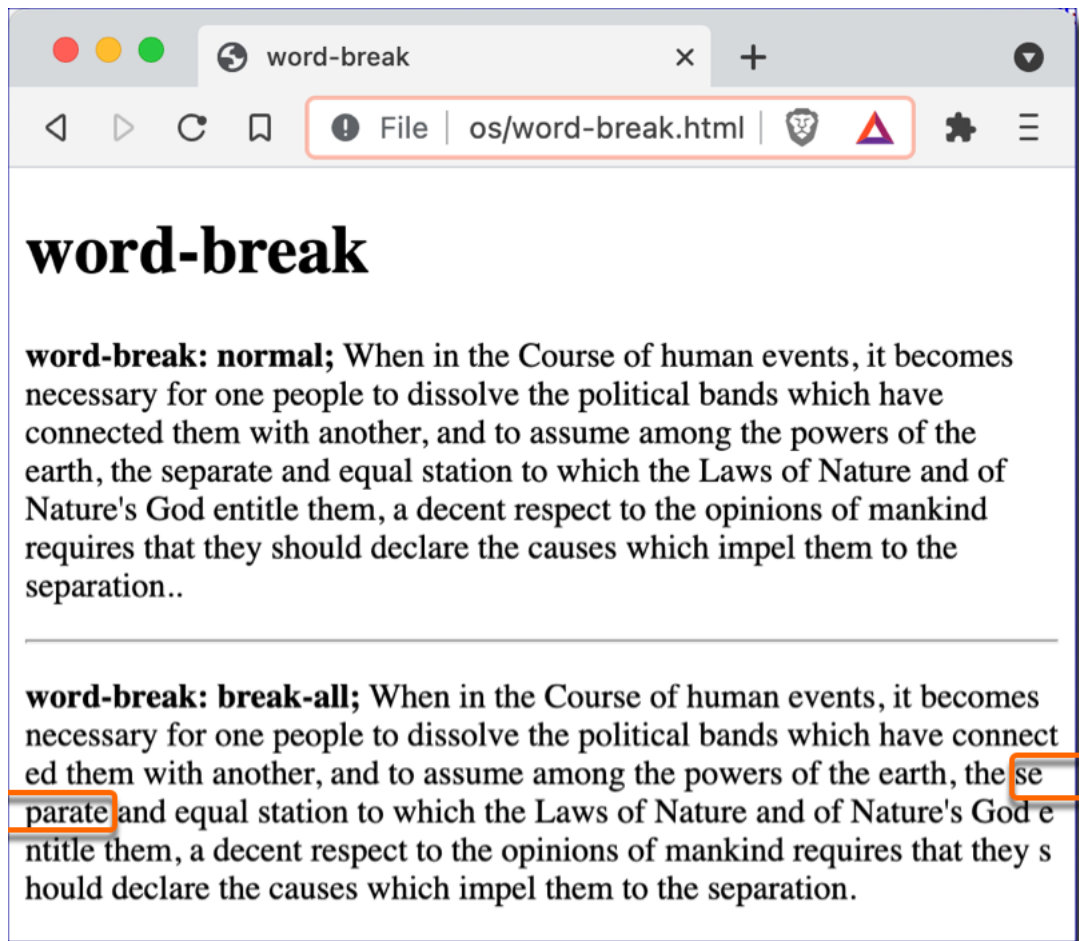
Demo 11.15: CssText/Demos/word-break-styles.css

```
1.  #div1 {
2.      word-break: normal;
3.  }
4.
5.  #div2 {
6.      word-break: break-all;
7.  }
```

Demo 11.16: CssText/Demos/word-break.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.   <h1>word-break</h1>
12.   <div id="div1">
13.     <p><strong>word-break: normal;</strong> When in the Course of human
14.       events, it becomes necessary for one people to dissolve the
15.       political bands which have connected them with another, and to
16.       assume among the powers of the earth, the separate and equal station
17.       to which the Laws of Nature and of Nature's God entitle them, a
18.       decent respect to the opinions of mankind requires that they should
19.       declare the causes which impel them to the separation.</p>
20.   </div>
21.   <hr>
22.   <div id="div2">
23.     <p><strong>word-break: break-all;</strong> When in the Course of
24.       human events, it becomes necessary for one people to dissolve the
25.       political bands which have connected them with another, and to
26.       assume among the powers of the earth, the separate and equal station
27.       to which the Laws of Nature and of Nature's God entitle them, a
28.       decent respect to the opinions of mankind requires that they should
29.       declare the causes which impel them to the separation.</p>
30.   </div>
31. </body>
32. </html>
```

The code above will render the following:



Notice that in the second rendering of the paragraph, some of the wrapping splits in the middle of a word (e.g., “separate”).



11.9. word-spacing

The word-spacing property is used to specify the amount of space between words. The amount indicated is in addition to the default spacing. The property either takes the keyword `normal`, which is the default defined by the current font and/or browser, or a length value. See the following example:

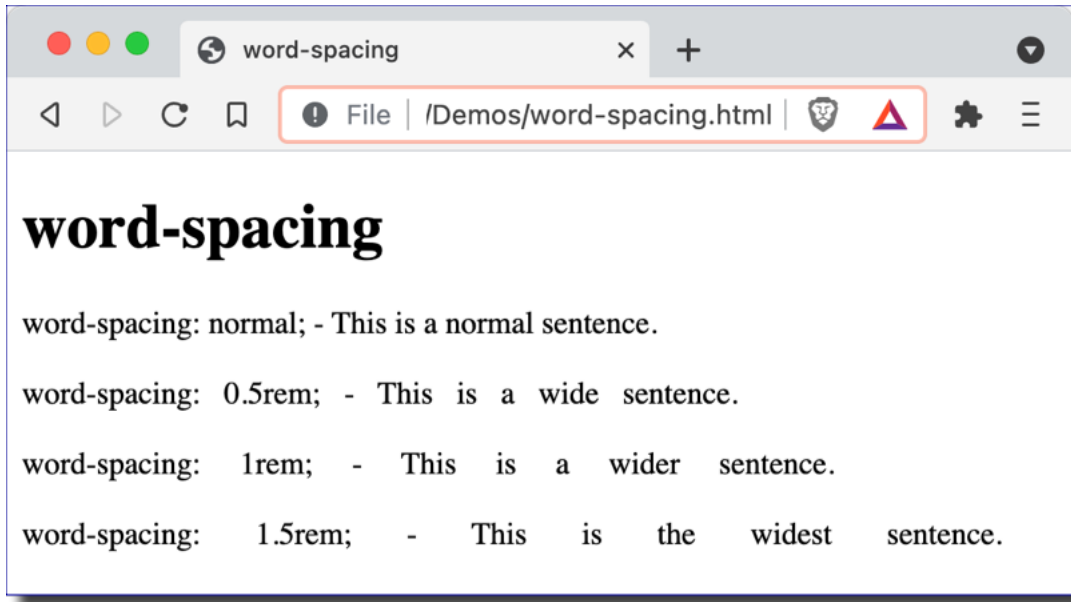
Demo 11.17: CssText/Demos/word-spacing-styles.css

```
1.  #normal {
2.      word-spacing: normal;
3.  }
4.
5.  #wide {
6.      word-spacing: 0.5rem;
7.  }
8.
9.  #wider {
10.     word-spacing: 1rem;
11. }
12.
13. #widest {
14.     word-spacing: 1.5rem;
15. }
```

Demo 11.18: CssText/Demos/word-spacing.html

```
-----Lines 1 through 9 Omitted-----
10. <body class="webucator">
11.     <h1>word-spacing</h1>
12.     <div id="normal">word-spacing: normal;
13.         - This is a normal sentence.
14.     </div><br>
15.     <div id="wide">word-spacing: 0.5rem;
16.         - This is a wide sentence.
17.     </div><br>
18.     <div id="wider">word-spacing: 1rem;
19.         - This is a wider sentence.
20.     </div><br>
21.     <div id="widest">word-spacing: 1.5rem;
22.         - This is the widest sentence.
23.     </div>
24. </body>
25. </html>
```

The code above will render the following:





Exercise 14: Text Properties

⌚ 25 to 40 minutes

In this exercise, you will continue to work on the Runners Home page that you started in the Fonts lesson (see page 194).

1. Open `CssText/Exercises/index.html` in your editor. You will see the Runners Home home page.
2. Open `CssText/Exercises/styles.css`. This is the stylesheet from the solution to the Fonts exercise. Feel free to continue working from your own stylesheet.
3. Using your new knowledge of text properties, add some styles to this page. The object of this exercise is to practice using the text properties covered in this lesson. Feel free to use color and opacity and any other properties you have learned as well.
4. When you are done, open `index.html` in your browser to see the results. You are welcome to go back to the code and continue to work.

You can design it however you like, or you can try to make it look something like:

- [Home](#)
- [Races](#)
- [Resources](#)
- [Calculator](#)
- [Running Log](#)
- [My Account](#)
- [Log out](#)

Hello, Stranger!

Welcome to Runners Home™



Runners Home™ is dedicated to providing you with:

1. [the most up-to-date information on running races.](#)
2. [the best resources for runners.](#)

BEST RUNNING TIPS

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam et gravida sapien, facilisis condimentum arcu. Morbi eget dui iaculis, porttitor eros et, tincidunt erat. Sed sollicitudin, mauris vitae semper consequat, purus velit aliquet mi, sit amet posuere eros tortor eget neque.

Phasellus hendrerit justo... [Click here to read more!](#)

BEST HEALTH TIPS

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam et gravida sapien, facilisis condimentum arcu. Morbi eget dui iaculis, porttitor eros et, tincidunt erat. Sed sollicitudin, mauris vitae semper consequat, purus velit aliquet mi, sit amet posuere eros tortor eget neque.

Phasellus hendrerit justo... [Click here to read more!](#)

Newsletter

Email:

Be the first to hear about our great offers.
Sign up for our newsletter today!

© 2022 Runners Home. All rights reserved. For questions, send email to info@runners-home.com.

Solution: CssText/Solutions/styles.css

```
1.  html {
2.    font-size: 16px;
3.  }
4.
5.  body {
6.    font: 1rem/1.3 Verdana, Geneva, Tahoma, sans-serif;
7.  }
8.
9.  nav a {
10.   text-decoration: none;
11. }
12.
13. #welcome {
14.   font-size: larger;
15. }
16.
17. #welcome h1 {
18.   color: rgb(8, 5, 211);
19.   text-shadow: 0.2rem 0.2rem 0.1rem rgb(211, 200, 238);
20. }
21.
22. article {
23.   text-align: center;
24. }
25.
26. article h2 {
27.   font-variant: small-caps;
28.   letter-spacing: 0.1em;
29.   word-spacing: 0.2em;
30. }
31.
32. article p {
33.   line-height: 2;
34.   text-align: left;
35.   text-indent: 3rem;
36. }
37.
38. .read-more {
39.   text-decoration: none;
40. }
41.
42. aside label {
43.   font-weight: bold;
44. }
```

Evaluation
Copy


```
45.  
46. #newsletter {  
47.     text-align: right;  
48. }  
49.  
50. footer {  
51.     font-size: smaller;  
52.     font-style: italic;  
53. }
```

Evaluation
Copy

Conclusion

In this lesson, you have learned to use CSS text properties.

LESSON 12

JavaScript Basics

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.



12.1. 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 Corporation⁴⁵. 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.

❖ 12.1.1. What is ECMAScript?

ECMAScript, sometimes abbreviated as “ES”, is a scripting language specification maintained and trademarked by Ecma International (<http://www.ecma-international.org/memento/in>

⁴⁵. <https://en.wikipedia.org/wiki/JavaScript#Trademark>

dex.html), 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/>

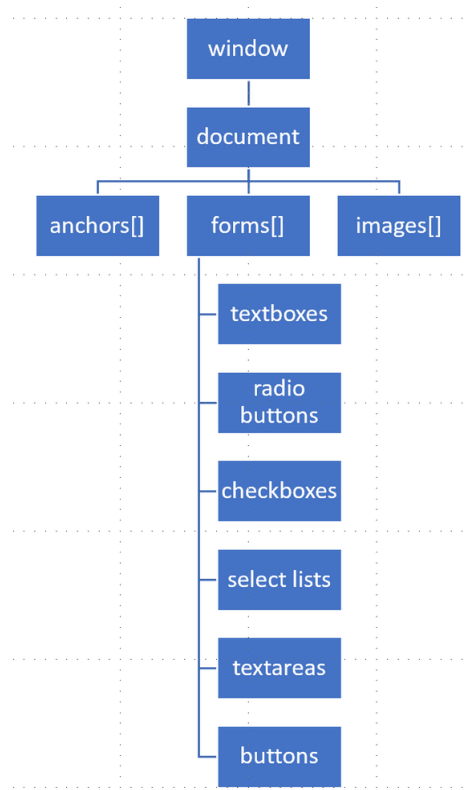
As we mentioned above, JavaScript – the scripting language you are learning here and whose code is run by the browsers you (or others) use to visit the pages you build – is an implementation of ECMAScript.

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.



12.2. 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.



12.3. JavaScript Syntax

❖ 12.3.1. 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.  
*/
```



12.4. Accessing Elements

❖ 12.4.1. Dot Notation

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.

❖ 12.4.2. 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

12.5. 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.

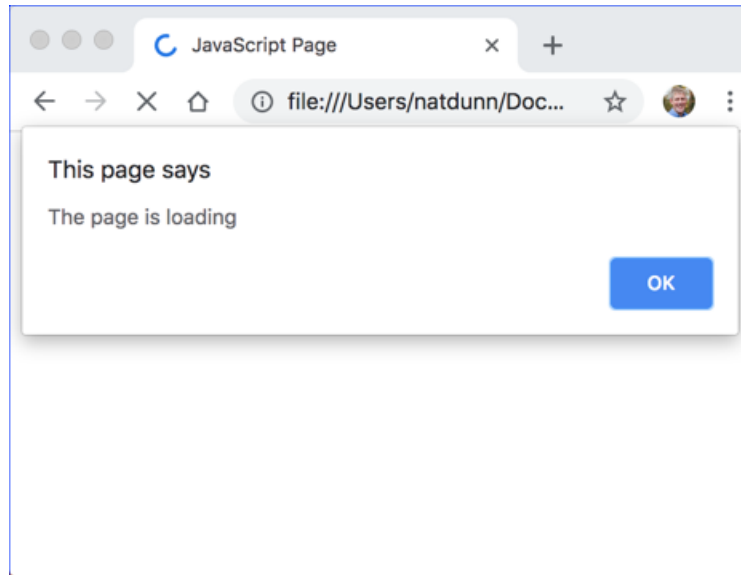
Demo 12.1: JavaScriptBasics/Demos/javascript.html

```
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>
```

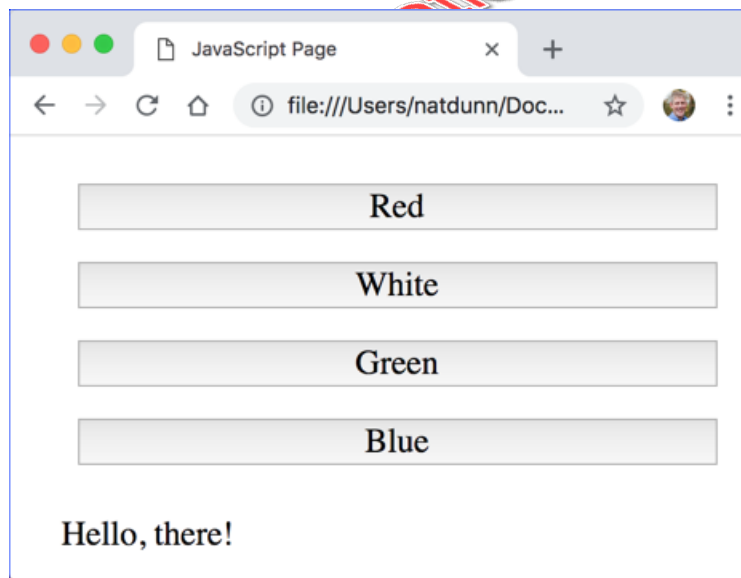
Demo 12.2: JavaScriptBasics/Demos/script.js

```
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:

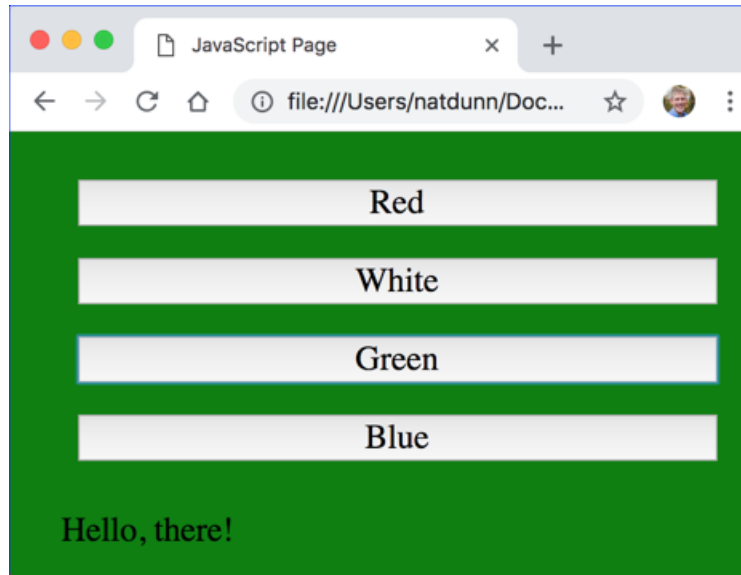


2. Click the **OK** button. The page will finish loading and will appear as follows:



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()`.



12.6. 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.

❖ 12.6.1. 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.

❖ 12.6.2. 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 15: 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 13

Variables, Arrays, and Operators

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.



13.1. 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.⁴⁶ See the Mozilla documentation⁴⁷ for details.



13.2. 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 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

TypeScript⁴⁸ 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

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

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

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

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.

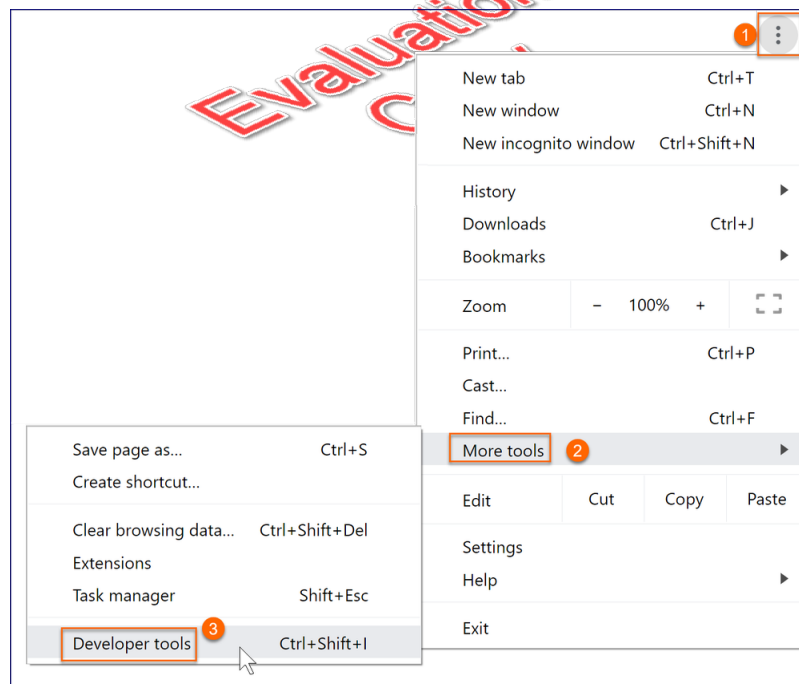


13.3. Google Chrome DevTools

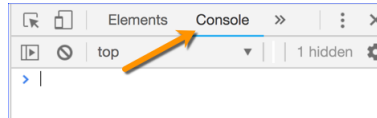
Google Chrome DevTools is a set of tools to help web developers. We will use the Chrome DevTools Console to illustrate JavaScript's dynamic typing.

To open the Chrome DevTools Console:

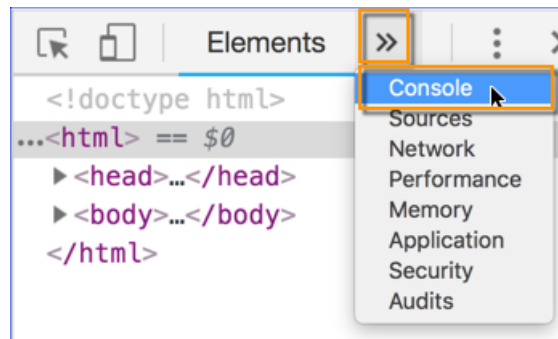
1. Click the three-vertical-dot icon in the upper right of Google Chrome.
2. Select **More Tools**.
3. Select **Developer Tools**.



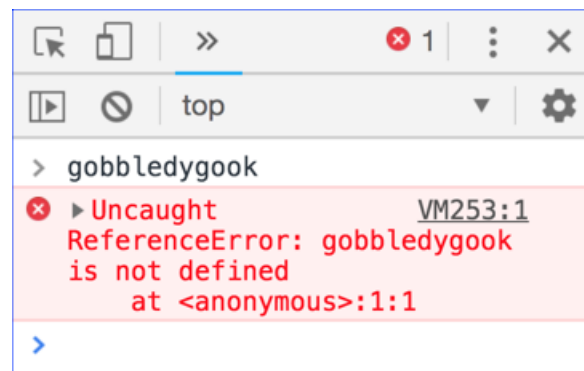
4. The tools will usually be docked on the right or bottom of your screen. Make sure that the Console is selected:



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

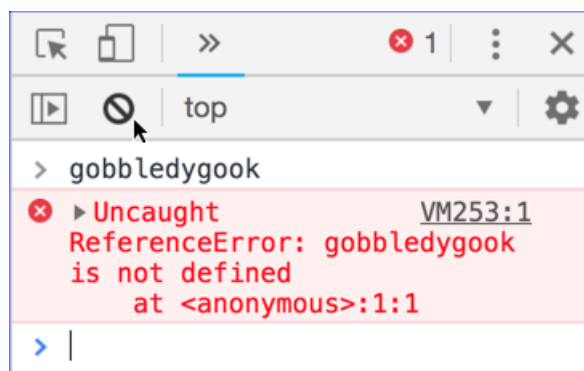


5. Now type “gobbledygook” in the Console and press **Enter**:

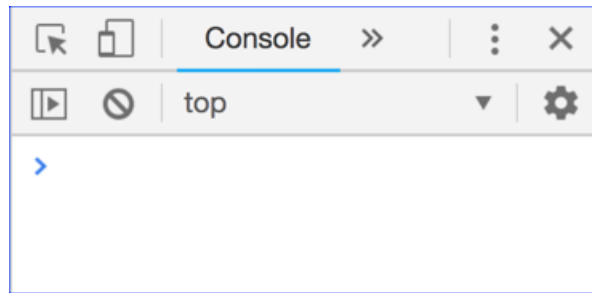


The word “gobbledygook” doesn’t mean anything in JavaScript and we have not defined a variable named “gobbledygook”, so we get an error.

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.

- 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.

- 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.

- 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.

❖ 13.3.1. Variable Naming

- Variable names must begin with a letter, underscore (`_`), or dollar sign (`$`).
- Variable names cannot contain spaces or special characters (other than the underscore and dollar sign).
- Variable names can contain numbers (but not as the first character).
- Variable names are case sensitive.

5. You cannot use keywords (e.g., window or function) as variable names.



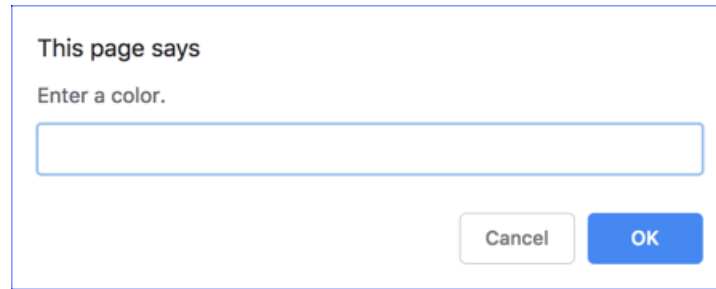
13.4. Storing User-Entered Data

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 13.1: 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`.⁴⁹ 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:

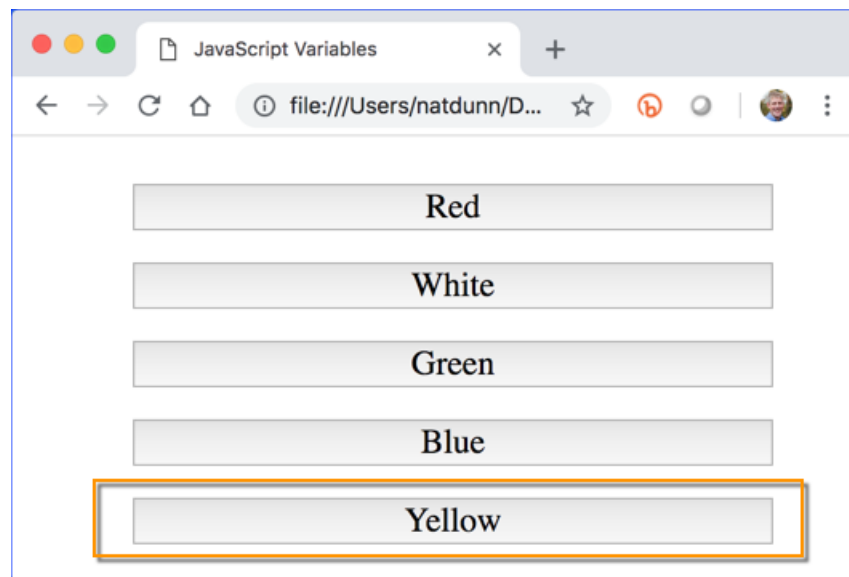
⁴⁹. In JavaScript, `null` is a datatype with only one value: `null`. It represents a value that we don't know or that is missing.

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 16: 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 16.1: `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>
```



13.5. 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 scope⁵⁰, 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

13.6. 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.⁵¹

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

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

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

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

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

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

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;
```

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 13.2: 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 17: 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?

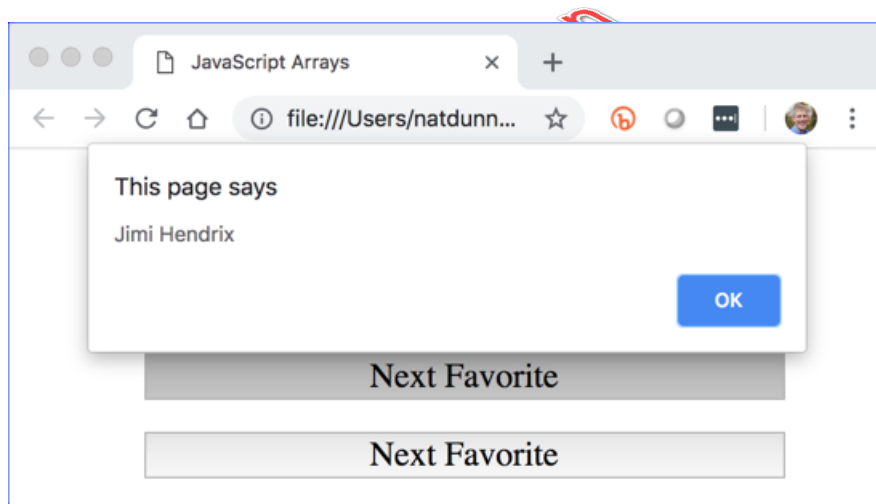
CancelOK

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 17.1: 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>
```



13.7. 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];
```

❖ 13.7.1. 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



13.8. 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.



13.9. 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
	<pre>let greeting = "Hello " + firstname;</pre>
+=	One step concatenation and assignment
	<pre>let greeting = "Hello "; greeting += firstname;</pre>

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!"

```



13.10. 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




13.11. 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.

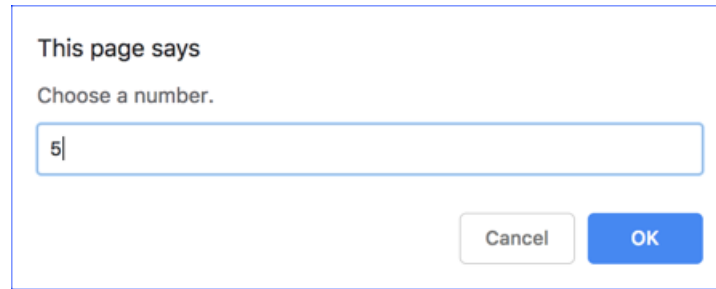
Demo 13.3: VariablesArraysOperators/Demos/operators.html

```
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:

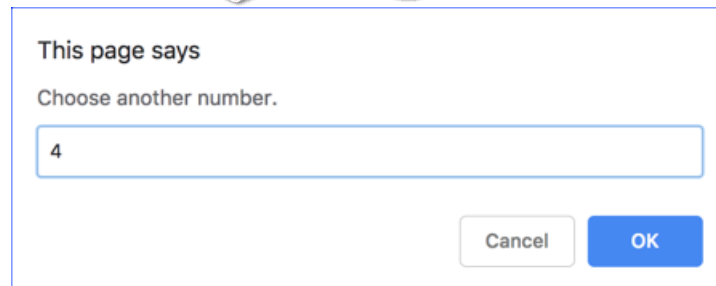


This page says

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

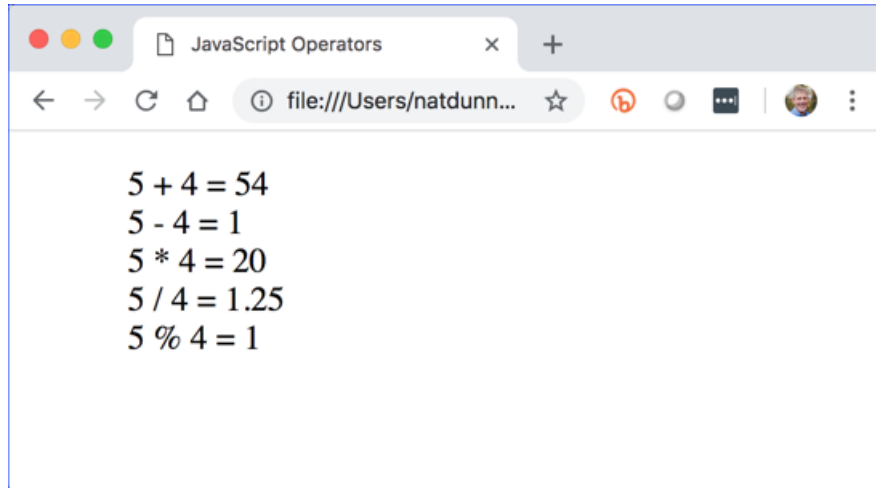
You chose 4

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.



13.12. The Default Operator

Default Operator

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

The following code sample shows how the default operator works:

Demo 13.4: 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?

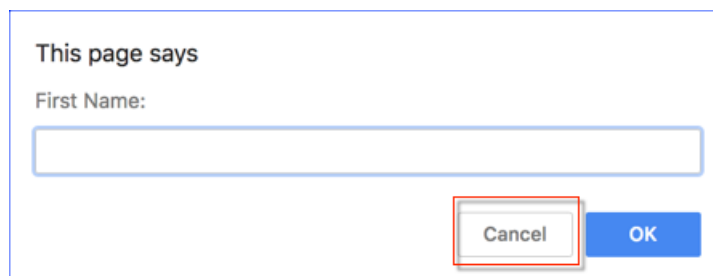
The default operator allows you to make sure that your variable contains a non-null value, so that you can perform operations on the variable with no errors. To illustrate, do the following in the Chrome DevTools Console:

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 18: 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?

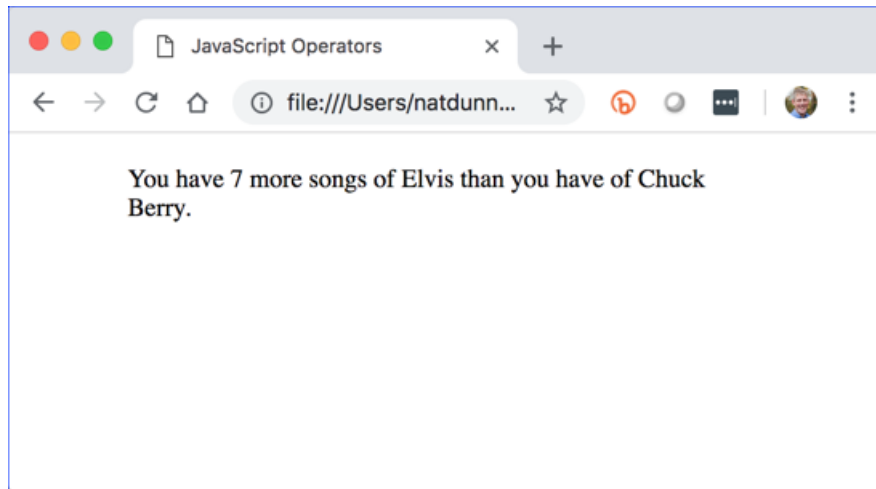
Cancel OK

This page says

How many Chuck Berry songs do you have?

Cancel OK

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:



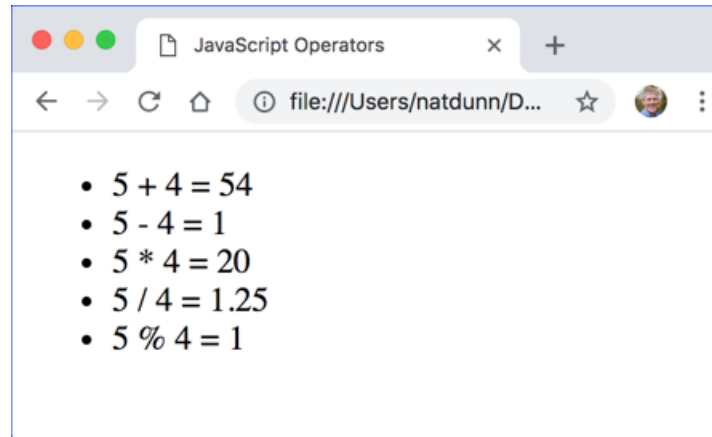
4. Test your solution in a browser.

Exercise Code 18.1: 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 14

JavaScript Functions

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.



14.1. 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.

❖ 14.1.1. 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
```

❖ 14.1.2. `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
```


❖ 14.1.3. `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 19: 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 19.1:

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?

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?

Cancel OK

And their current age:

This page says

How old are you now?

Cancel OK

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

JavaScript Built-in Functions x Nat

file:///... ☆

Nat, you have used computers for 36 years.

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.



14.2. 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.

❖ 14.2.1. 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 14.1: 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.

❖ 14.2.2. 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 14.2: 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 14.3:

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 20: 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:

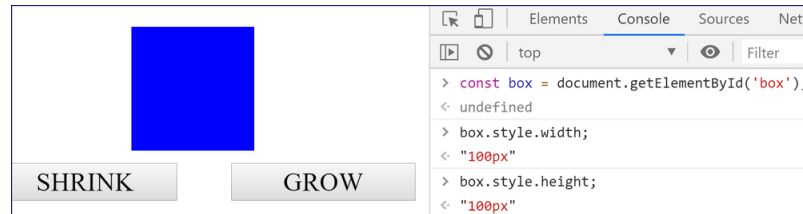
- A. Declare a constant `box` that holds the “box” `div`. You will do this using `document.getElementById()`, which is a method for accessing elements on the page by their id value:

```
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.



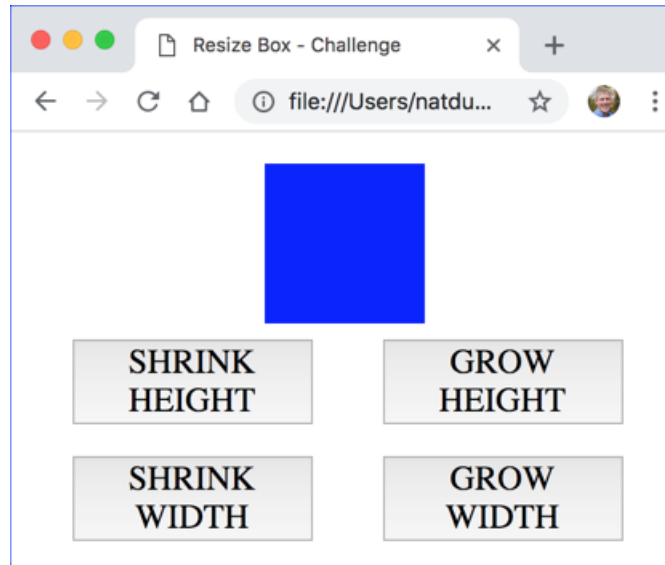
- C. Just as you did for width, declare a constant `h` that holds the current height of the box.
- D. Declare variables `wNew` and `hNew` that contain the new width and height values. Note that you will need to add the value of `change` to the current values of `w` and `h`, but before doing so, you will 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 20.1: 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>
```



14.3. Returning Values from Functions

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

Demo 14.4: 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 15

Built-In JavaScript Objects

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.

15.1. 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.
	<code>myStr.length; // returns 9</code>

Try the following out in the Chrome DevTools Console:

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

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.



15.2. 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
Math.PI	The value of Pi (π)
	Math.PI; //3.141592653589793
Math.SQRT2	Square root of 2.
	Math.SQRT2; //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
<code>Math.abs(number)</code>	Absolute value of number.
	<code>Math.abs(-12); // returns 12</code>
<code>Math.ceil(number)</code>	number rounded up.
	<code>Math.ceil(5.4); // returns 6</code>
<code>Math.floor(number)</code>	number rounded down.
	<code>Math.floor(5.6); // returns 5</code>
<code>Math.max(numbers)</code>	Highest Number in numbers.
	<code>Math.max(2, 5, 9, 3); // returns 9</code>
<code>Math.min(numbers)</code>	Lowest Number in numbers.
	<code>Math.min(2, 5, 9, 3); // returns 2</code>
<code>Math.pow(number, power)</code>	number to the power of power.
	<code>Math.pow(2, 5); // returns 32</code>
<code>Math.round(number)</code>	Rounded number.
	<code>Math.round(2.5); // returns 3</code>
<code>Math.random()</code>	Random number between 0 and 1.
	<code>Math.random(); // Returns random number from 0 to 1</code>

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.



15.3. Date

Evaluation Copy

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

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 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.



15.4. 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 15.1: 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 21: 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 16

Conditionals and Loops

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.



16.1. Conditionals

There are two types of conditionals in JavaScript:

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

❖ 16.1.1. 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

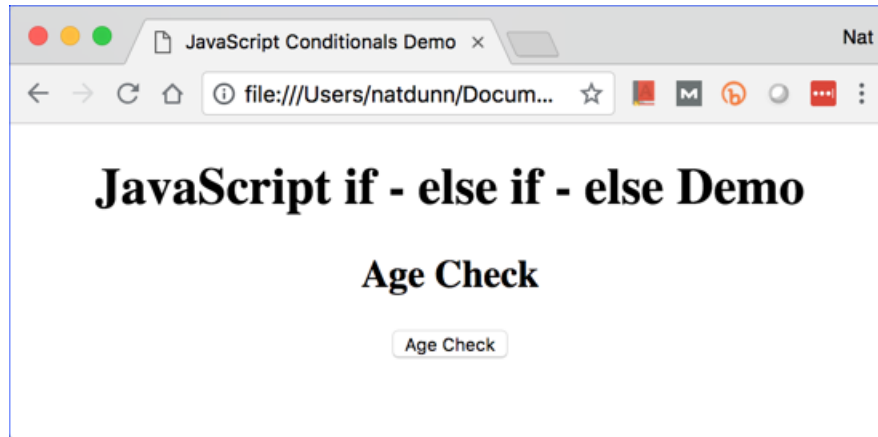
Operator	Description	Example
&&	and	(a == b && c != d)
	or	(a == b c != d)
!	not	!(a == b c == d)

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

Demo 16.1: 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:

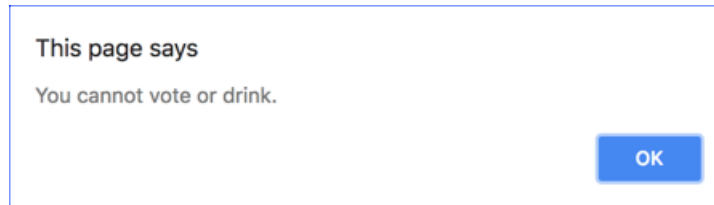


When the user clicks the **Age Check** button, the following prompt pops up:

A prompt box with a light gray background. It has a title 'This page says' in bold. 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 box with a light gray background. It has a title 'This page says' in bold. Below the title is the text 'You can vote and drink!'. At the bottom right is a blue button labeled 'OK'.An alert box with a light gray background. It has a title 'This page says' in bold. Below the title is the text 'You can vote, but can't drink.'. At the bottom right is a blue button labeled 'OK'.



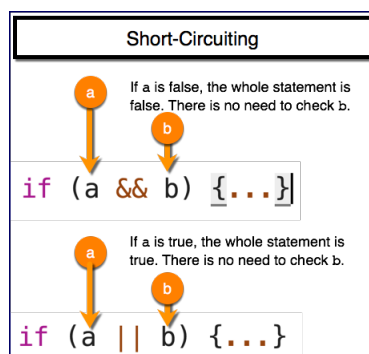
Compound Conditions

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");  
}
```

16.2. 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 16.2: ConditionalsAndLoops/Demos/password-check-broken.html

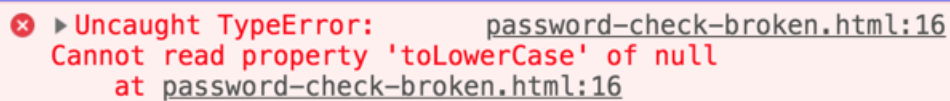
```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <script>
9.      const userPass = prompt("Password:", ""); //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>
```

Evaluation
Copy

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 16.3: 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`.



16.3. 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*.

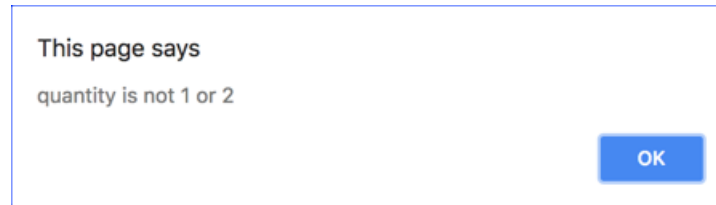
Take a look at the following example:

Demo 16.4: 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 16.5: 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 16.6: 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);
```

Evaluation
Copy

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.



16.4. 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 16.7: 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.



16.5. 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 22: 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-----
```



16.6. Loops

There are several types of loops in JavaScript:

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



16.7. while and do...while Loops

❖ 16.7.1. while Loop Syntax

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

Evaluation
Copy

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
```

❖ 16.7.2. 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:

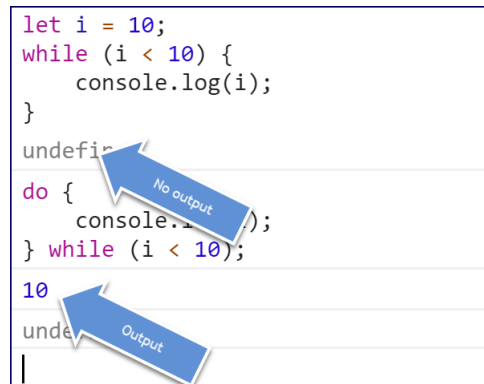
```
let i=0;
do {
  console.log(i);
  i++; // changing value of i
} while (i < 5);
```

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
|
```



16.8. for Loops

❖ 16.8.1. 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
```

❖ 16.8.2. 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);  
}
```

Evaluation
Copy

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
```

❖ 16.8.3. 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.



16.9. 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();
```

Evaluation
Copy

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 23: 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-----
```

16.10. 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 17

Event Handlers and Listeners

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.



17.1. 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.

❖ 17.1.1. 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 17.1: 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 24: 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 24.1: 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>
```



17.2. 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 17.2: 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 17.3: 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 17.4: 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`.



17.3. 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 17.5: 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.



17.4. 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 deprecated⁵² 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.

Demo 17.6: 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>
```

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

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();
  }
});
```

Evaluation
Copy

innerHTML

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



Exercise 25: Adding Event Listeners

🕒 15 to 25 minutes

You will start with the following code:

Exercise Code 25.1: 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



17.5. 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 17.7: 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.



17.6. 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 17.8: 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 17.9: 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 26: 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 26.1: 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 18

The HTML Document Object Model

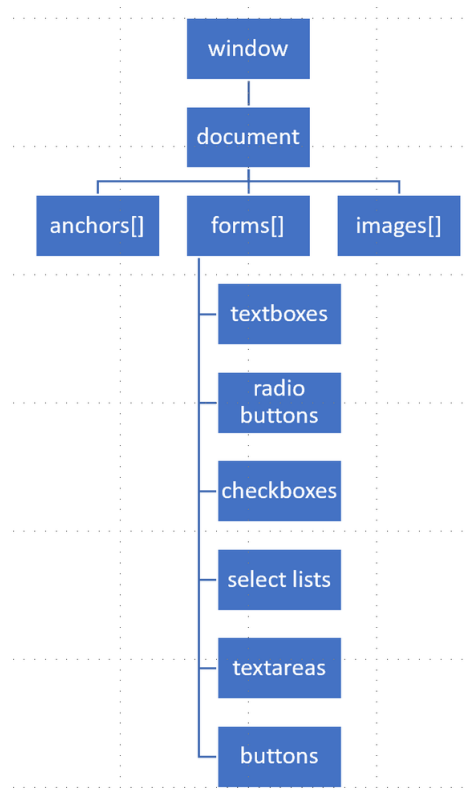
Topics Covered

- ☒ 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.



18.1. 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.

❖ 18.1.1. 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;  
}
```

❖ 18.1.2. 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.

❖ 18.1.3. 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.

❖ 18.1.4. 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">...
```

❖ 18.1.5. 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>
```

❖ 18.1.6. 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.



18.2. 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.



18.3. 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>.



18.4. 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.
- `getElementsByClassName(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`.

❖ 18.4.1. `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 18.1: 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:



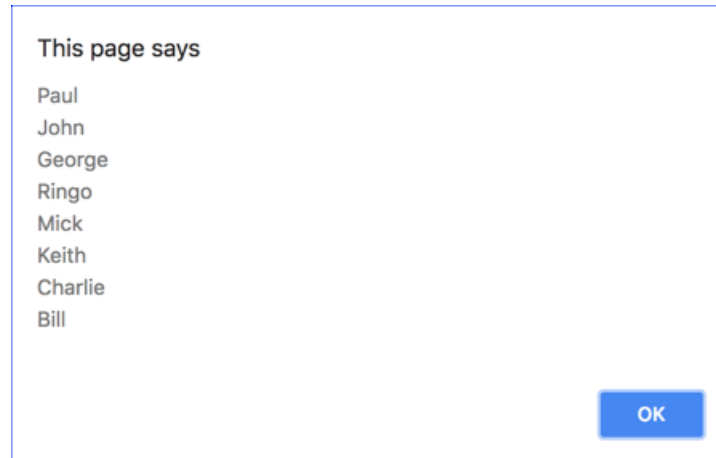
❖ 18.4.2. `getElementsByName()`

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 18.2: [HTMLDOM/Demos/get-elements-by-tag-name.html](#)

```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.  window.addEventListener('load', function() {
10.    const elems = document.getElementsByTagName('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:



❖ 18.4.3. `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');
```

❖ 18.4.4. `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:

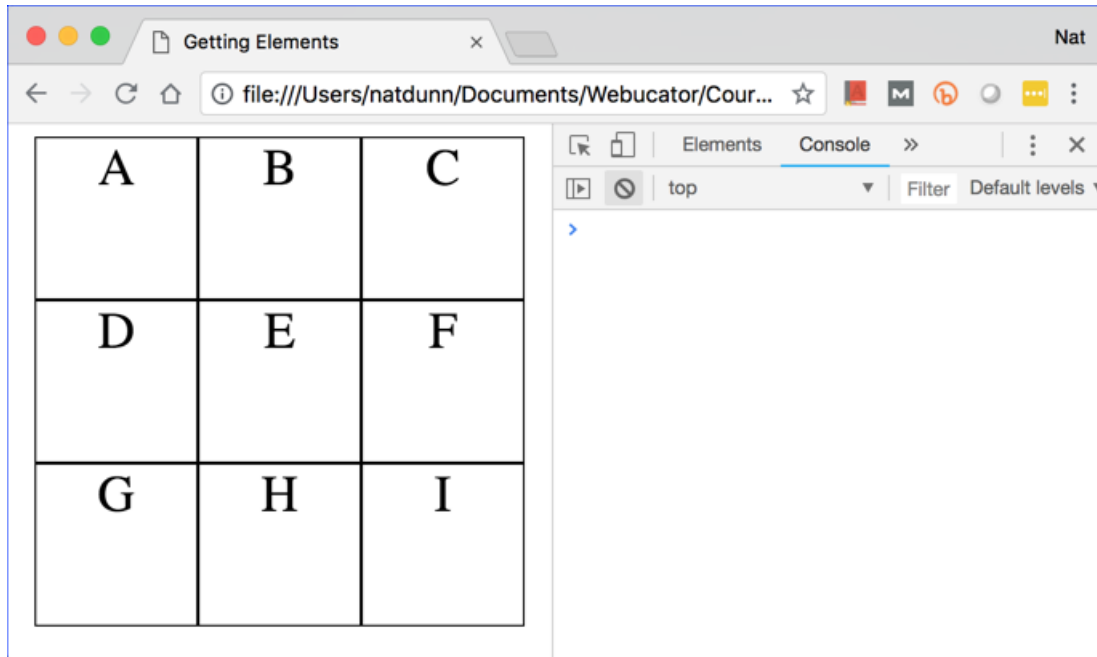
```
const firstLinkInTd = document.querySelectorAll('td>a')[0];  
const firstLinkInTd = document.querySelector('td>a');
```

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

Demo 18.3: 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

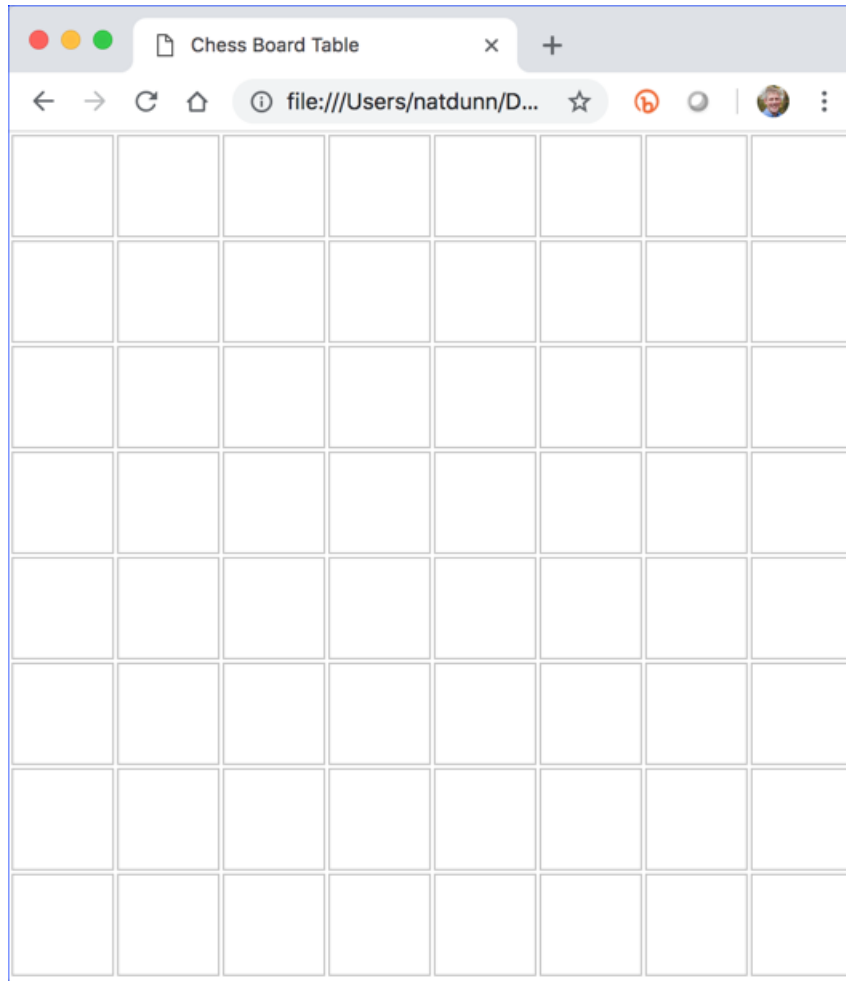
```
> const squares = document.getElementsByClassName('col');
  for (let i=0; i < squares.length; i++) {
    squares[i].innerHTML = i+1;
  }
< 9
> |
```

Exercise 27: Accessing Elements

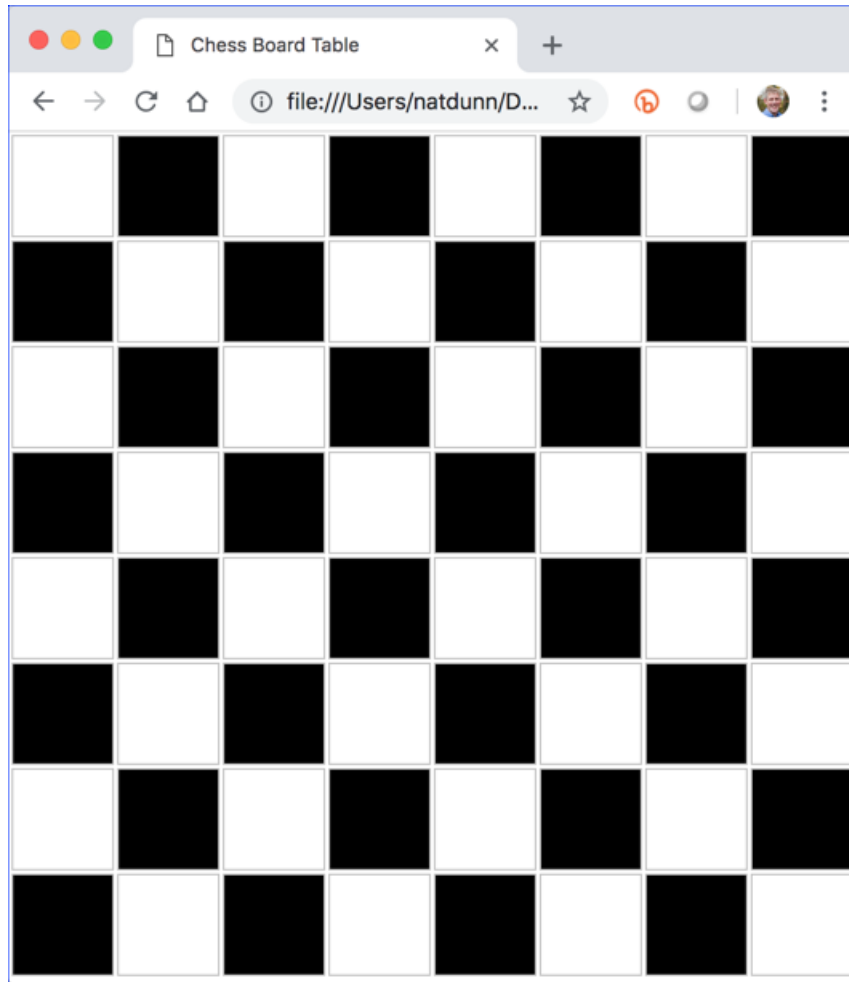
 10 to 15 minutes

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-----
```

Code Explanation

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



18.5. 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
```

❖ 18.5.1. 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
```

❖ 18.5.2. 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 18.4: HTMLDOM/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 HTMLDOM/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"]`;

```
> document.forms[0];
< ▶ <form name="form-a">...</form>
> document['form-a'];
< ▶ <form name="form-a">...</form>
```

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

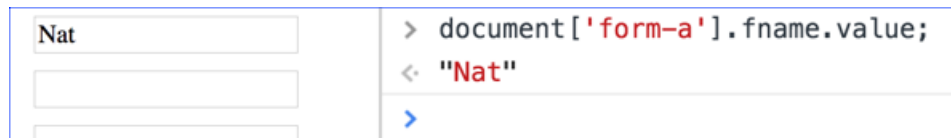
```
> document.form-a;
✖ ▶ Uncaught ReferenceError: a is not defined
   at <anonymous>:1:15
```

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:

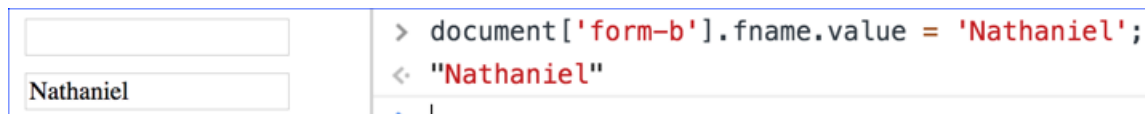
```
> const form = document['form-a'];  
< undefined  
> form.fname;  
< <input name="fname">  
> form['fname'];  
< <input name="fname">
```

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:



```
> document['form-a'].fname.value;  
< "Nat"
```

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



```
> document['form-b'].fname.value = 'Nathaniel';  
< "Nathaniel"
```



18.6. 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. ⁵³

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:

53. 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 18.5: 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 28: Working with Hierarchical Elements

Evaluation
Copy

⌚ 10 to 15 minutes

You will start with the code shown below:

Exercise Code 28.1: HTMLDOM/Exercises/elem-hierarchy.html

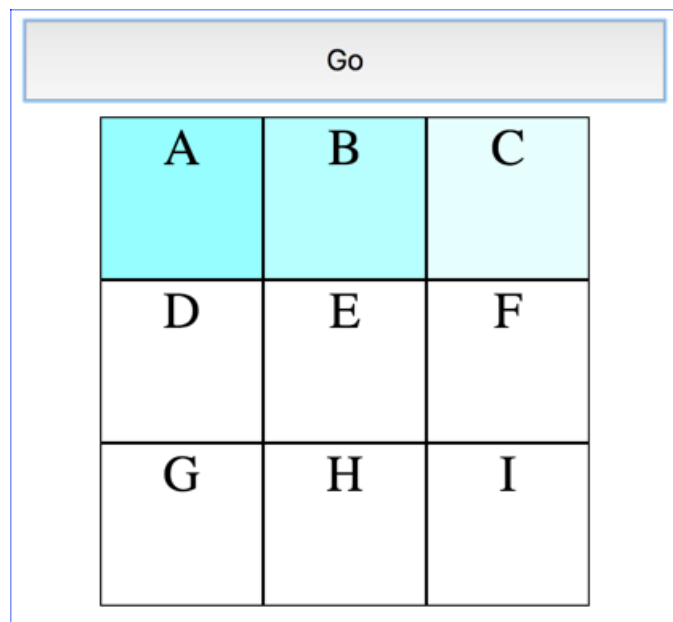
```
-----Lines 1 through 7 Omitted-----
8.  <script>
9.    function create() {
10.      const board = document.getElementById('board');
11.      const topRow = board.firstChild;
12.      const trLeftCol = topRow.firstChild;
13.      trLeftCol.style.backgroundColor='rgba(0, 255, 255, .5)';
14.      const 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.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.      const middleRow = topRow.nextElementSibling;
20.      const mrLeftCol = middleRow.firstChild;
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.firstChild;
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-----
```



18.7. 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:

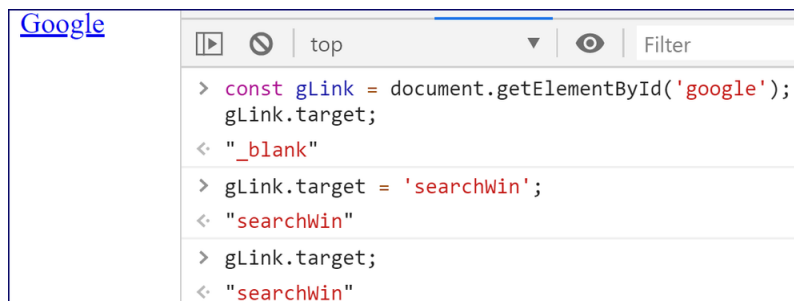
```
const gLink = document.getElementById('google');
console.log(gLink.target);
```

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 (<code>true/false</code>) 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 <code>null</code> 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.



18.8. 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:

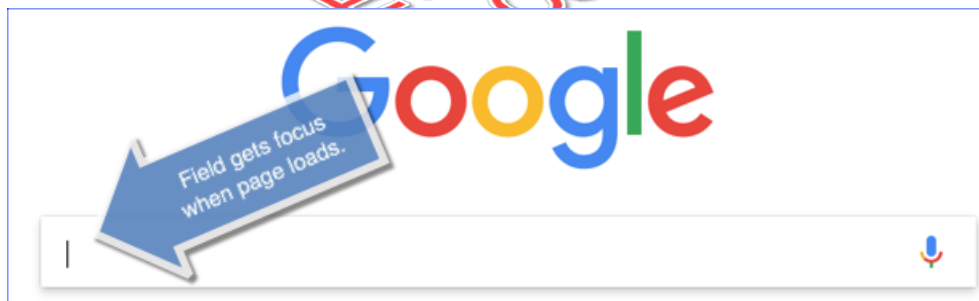
Methods for Inserting and Removing Nodes

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.



18.9. 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 18.6: 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.



18.10. 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="634 468 672 520" type="button" value="-"/>
Bread	<input data-bbox="634 520 672 575" type="button" value="-"/>

Common Items

Milk	<input data-bbox="964 468 1008 520" type="button" value="+"/>
Eggs	<input data-bbox="964 520 1008 575" type="button" value="+"/>
Bread	<input data-bbox="964 575 1008 630" type="button" value="+"/>
Chicken	<input data-bbox="964 630 1008 684" type="button" value="+"/>
Tomatoes	<input data-bbox="964 684 1008 737" 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 29: 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 30: 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 31: 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:

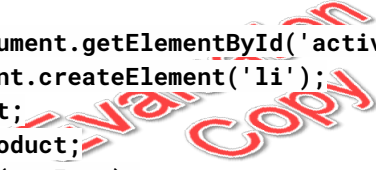
```
function addToList(product) {  
  log(product + ' added.')
```

```
}  
  
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 32: 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 `HTMLODOM/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 33: 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 `HTMLODOM/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 34: 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-----
```



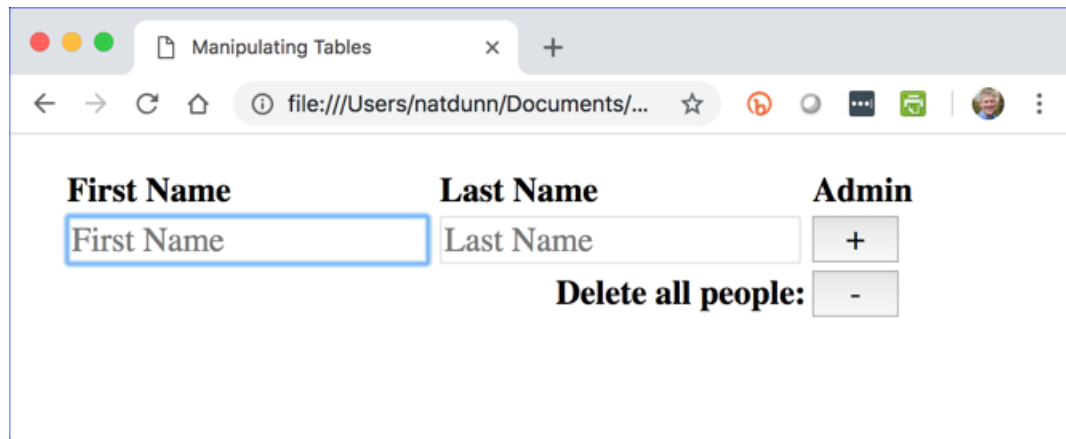
18.11. Manipulating Tables

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

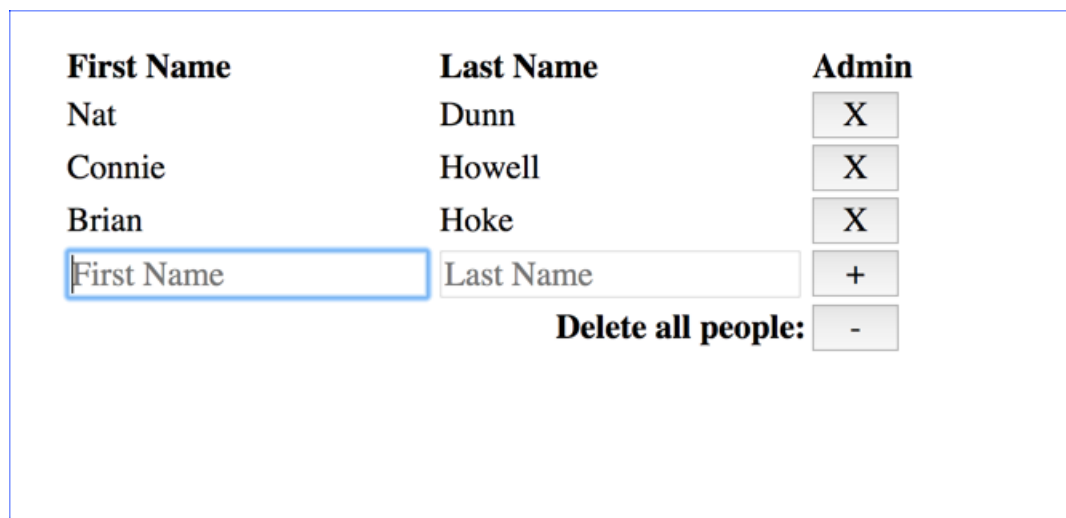
cells: `insertCell()` and `deleteCell()`. The following example shows how these objects can be used to dynamically create HTML tables.

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 18.7: 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;
  }

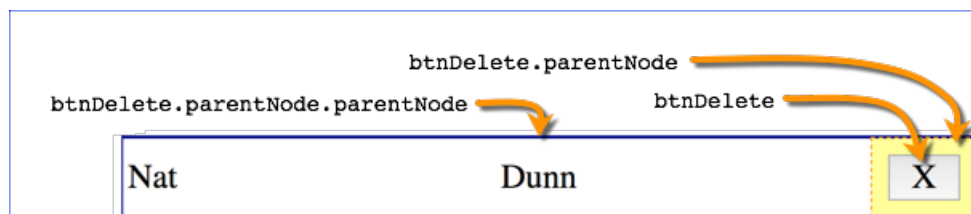
  // Insert a final cell with a Delete button
  newCell = newRow.insertCell();
  const btnDelete = document.createElement('button');
  btnDelete.innerHTML = 'X';
  btnDelete.addEventListener('click', function(e) {
    btnDelete.parentNode.parentNode.remove();
  });
  newCell.appendChild(btnDelete);
}
```

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 19

HTML Forms

Topics Covered

- ☒ How HTML forms work.
- ☒ The post and get methods.
- ☒ Form elements.
- ☒ Labels.

Introduction

In this lesson, you will learn to work with HTML forms.

19.1. How HTML Forms Work

HTML forms are used for submitting data back to a script on the server for data processing. When a form is submitted, the data in the form fields is passed to the server as name-value pairs. Server-side scripts, which can be written in several different languages, are used to process the incoming data and return a new HTML page to the browser. The page returned to the browser could be anything from a “Thank you for registering” message to a list of search results generated from a database query.

The form processing occurs in the following sequence:

1. The user fills out the form and submits the form data using a “submit” button.
2. The data is sent to the web server.
3. A script on the web server processes the form, possibly interacting with the file system, one or more databases, a mail server, or any number of other applications.
4. The script generates an HTML page, which the server returns to the client for display.



19.2. The form Element

HTML forms are created using the `<form>` tag, which takes two main attributes: `action` and `method`.

The `action` specifies the URL of the page that processes the form. The `method` attribute has two possible values: `post` and `get`. Here is an example of a form element:

```
<form method="post" action="process-form.cfm">
  <!--form fields go here-->
</form>
```

❖ 19.2.1. Get vs. Post

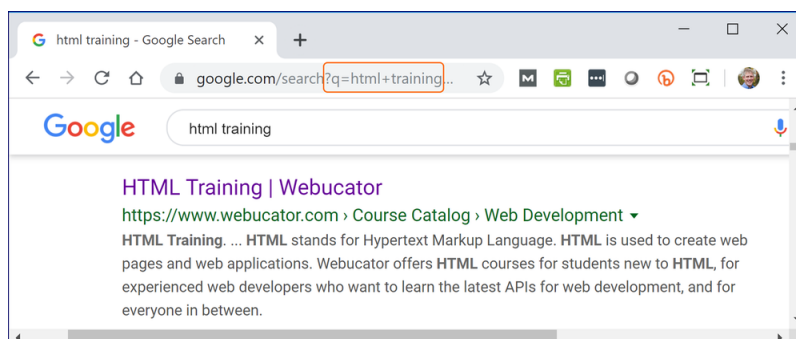
The value of the `method` attribute determines how the form data will be passed to the server.

get

When using the `get` method, which is the default, form data is sent to the server in the URL as a *query string*. The query string is appended to the website address starting with a question mark (?) and followed by name-value pairs delimited (separated) by an ampersand (&). A URL with a query string might look like this:

```
https://www.example.com?firstname=Nat&lastname=Dunn
```

The `get` method is commonly used by search engines, because it allows the resulting page to be bookmarked. For example, Google uses the `get` method. You can tell by looking at the location bar after doing a search:



post

When `post` is used, the name-value pairs are not sent as part of the query string. Instead, they are sent behind the scenes. This has the advantage of keeping the values hidden from anyone looking over the user's shoulder. Two other advantages of the `post` method are:

1. It allows for much more data to be submitted (i.e., larger forms).
2. It allows for files to be uploaded to the server.⁵⁴

Use Post for Most Forms

As a general rule, you should use `post` unless you want the user to be able to bookmark or share (e.g., via email) the resulting web page.



19.3. Form Elements

This section describes the different form elements that can be used to input data into a form. As you will see, many of these elements, but not all, are created with the `<input>` tag.

❖ 19.3.1. id and name Attributes

Form fields (also called controls) take both the `name` attribute and the `id` attribute. They are used for different purposes:

- The `name` attribute is used to hold the value of the field when data is sent to the server.
- The `id` attribute is used by the browser to identify a specific element.

❖ 19.3.2. Text Fields

Text fields are created with the `<input>` tag with the `type` attribute set to `"text"`. They are used for single lines of text:

Username:

⁵⁴. Files can be uploaded to the server via the `file` input type. The tag syntax is: `<input type="file" name="filename">`.

The code to create the input element shown above is:

```
Username: <input type="text" name="username" id="username">
```

As text is the default type for input elements, if the type attribute is absent, the input type will be text. So, the above code can also be written:

```
Username: <input name="username" id="username">
```

❖ 19.3.3. Labels

Form element labels should be put in <label> tags. Labels can be associated with form elements using two methods:

1. Using the for attribute of the <label> to point to the id attribute of the form element.
2. Wrapping the form element in the <label> tag.

Method 1

```
<label for="username">Username:</label>  
<input type="text" name="username" id="username">
```

Method 2

```
<label>  
  Username:  
  <input type="text" name="username" id="username">  
</label>
```

We will mostly use the first method.

❖ 19.3.4. Text-like Input Types

There are many input types that are similar to the text type:

1. password
2. date
3. time

4. `datetime-local`
5. `month`
6. `week`
7. `color`
8. `email`
9. `tel`
10. `url`
11. `search`
12. `number`

Not all of these types are supported by all browsers. When a browser does not support a certain type, it will fall back to using a standard `text` type, so you can technically use all of these types today. However, **we recommend not using the date and time types until there is more consistent browser support**. We will explain why soon.

Common Attributes for Text-like Inputs

1. `type` – Input type (e.g., `text`, `tel`, etc.). Applies to all text-like inputs.
2. `name` – Variable name used to send data to server. Applies to all text-like inputs.
3. `id` – Variable name used to identify field in the browser. Applies to all text-like inputs.
4. `value` – Initial value in the field. Applies to all text-like inputs.
5. `size` – Approximate number of characters visible in the field. Applies to `text`, `search`, `tel`, `url`, `email`, and `password`.
6. `minlength` – Minimum number of characters that must be entered. Applies to `text`, `search`, `tel`, `url`, `email`, and `password`.
7. `maxlength` – Maximum number of characters that can be entered. Applies to `text`, `search`, `tel`, `url`, `email`, and `password`.
8. `placeholder` – A hint indicating what should be entered in the field. Applies to all text-like inputs.
9. `pattern` – A regular expression expressing a valid value for the field. Applies to `text`, `search`, `tel`, `url`, `email`, and `password`.

10. **required** – When present, the user must fill in a value before submitting the form. Applies to all text-like inputs.
11. **autofocus** – Instructs the browser to place focus on that field allowing the user to begin typing as soon as the page loads. Applies to all text-like inputs.
12. **autocomplete** – Used to override the browser’s or form element’s autocomplete behavior on a field-by-field basis. When used, it is usually set to “off”.⁵⁵ Applies to all text-like inputs, except password.

Be Careful with Autofocusing

Autofocusing on a form element can cause problems for people using screen readers. For sighted people, it’s generally okay if we provide one focus point for the keyboard (i.e., autofocus) and another one for the eyes (e.g., instructions for filling out the form), but for people using screen readers, there is only one focus point. So be careful not to skip over important contextual content when directing focus to a form field using autofocus.

❖ 19.3.5. placeholder Attribute

The `placeholder` attribute is used to add placeholder text to the form field. The placeholder text will go away as soon as the user begins typing in the field. The following code illustrates:

Demo 19.1: Forms/Demos/placeholder.html

```
-----Lines 1 through 9 Omitted-----
10.    <label for="fullname">Name: </label>
11.    <input type="text" name="fullname" id="fullname"
12.        placeholder="Enter full name">
13.    <input type="submit">
-----Lines 14 through 16 Omitted-----
```

Here’s what it looks like in the browser:

Name:

55. See <https://html.spec.whatwg.org/multipage/form-control-infrastructure.html#autofill>.

❖ 19.3.6. pattern Attribute

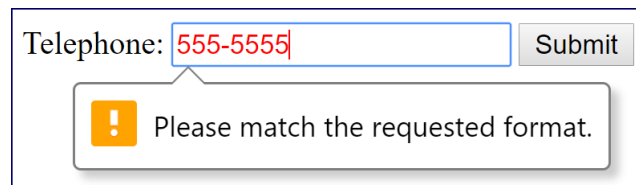
The `pattern` attribute is used to force a specific pattern (via a regular expression⁵⁶) within a form field. You can use a `placeholder` to give an example of a valid entry:

Demo 19.2: Forms/Demos/pattern.html

```
-----Lines 1 through 14 Omitted-----
15.    <label for="telephone">Telephone: </label>
16.    <input type="tel" name="telephone" id="telephone"
17.        placeholder="(555) 555-5555"
18.        pattern="^\(?([2-9]\d\d)\)?[ \-\. ]?([2-9]\d\d)[ \-\. ]?(\d{4})$">
-----Lines 19 through 22 Omitted-----
```

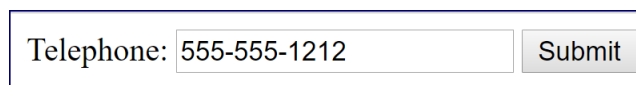
Open this file in the browser and enter data in the field. It should remain red⁵⁷ until the value is a valid 10-digit U.S.-style phone number. It will allow for parentheses around the area code and for dashes, spaces, and dots as separators. It will also allow for no separators. Note the first digit may not be a 0 or a 1.

When the user tries to submit with an invalid pattern, an error will appear:



The screenshot shows a web form with a label "Telephone:" followed by a text input field containing "555-5555" in red text. To the right of the input field is a "Submit" button. Below the input field, a yellow warning icon is displayed next to the message "Please match the requested format."

When the user enters a valid pattern, the field will indicate that it is valid (e.g., by changing the font color from red to black):



The screenshot shows the same web form as before, but the text in the input field is now "555-555-1212" in black text. The "Submit" button remains to the right.

❖ 19.3.7. Password Fields

Password fields are similar to text fields. They are coded as follows:

^{56.} Regular expressions are used by many programming languages for pattern matching. The syntax is complex, but super powerful.
^{57.} Note that we use CSS to make the invalid content red. By default, you won't know it's invalid until you submit.

```
<label for="pw">Password:</label>
<input type="password" name="pw" id="pw" size="10" maxlength="12">
```

The only difference between a password field and a text field is that the value entered in a password field is disguised so that onlookers cannot see it:

Password:

Not Really Secure

Note that there is no additional security provided by password fields beyond obscuring the text you enter in the field. Passwords are not sent to the server any differently than other fields.

❖ 19.3.8. Date and Time Fields

The date, time and datetime-local field controls in Google Chrome are beautiful, both in their presentation and in their control of the data entry. When these controls are used, you can count on the data being in the formats shown in the following table:

Date and Time Data Formats

Input Type	Data Format	Example(s)
date	yyyy-mm-dd	1991-06-12
time	hh:mm	03:05, 15:05
datetime-local	yyyy-mm-ddThh:mm	1991-06-12T03:05
month	yyyy-mm	1776-07
week	yyyy-Www	2028-W02

Knowing the format of incoming data makes it much easier to process the data on the server side. Unfortunately, users of browsers (most notably Safari) that do not support these field types are presented with basic text fields instead of the more modern date and time controls. Such users are not likely to enter data in the formats shown in the table above. That means that you have to write code to validate the user-entered data. While this is possible, it would still be confusing to the user. As such, for now, you are better off taking other approaches to collecting date and time values. One simple approach is to use a combination of number inputs and radio buttons. We will learn about those soon. In the meantime, here is a screenshot of what the form might look like in the browser:

Birth Date: / /
 Birth Time: : ☐ AM ☐ PM

While this isn't as pretty as the date and time controls used by some browsers, for now, it provides a safer way of collecting accurate data from the user.

❖ 19.3.9. Number Fields

Browsers present up and down buttons (*spinboxes*) to scroll through numbers, and they also allow you to use the up and down arrows on the keyboard:

Number:

The step Attribute

By default, numbers increment by 1 and any non-integer (e.g., a decimal like 0.5) is considered invalid. However, you can change the increment using the `step` attribute:

```
<input id="amount" name="amount" type="number" step=".01">
```

With `step` set to ".01", valid numbers can have a decimal point followed by one or two digits.

The min and max Attributes

You can control the range of possible values using the `min` and `max` attributes:

```
<input id="amount" name="amount" type="number" step=".01" min="0" max="100">
```

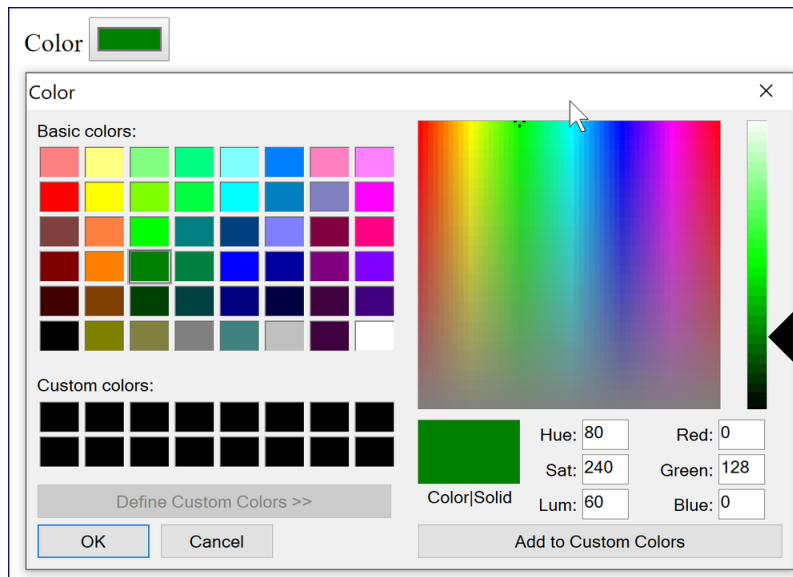
With the code above, numbers below 0 and above 100 will be invalid.

❖ 19.3.10. Color Fields

Most browsers will present a color picker when the user focuses on a color field:

```
<input id="color" name="color" type="color">
```

This will appear different in different browsers. The following screenshot shows how it appears in Google Chrome:



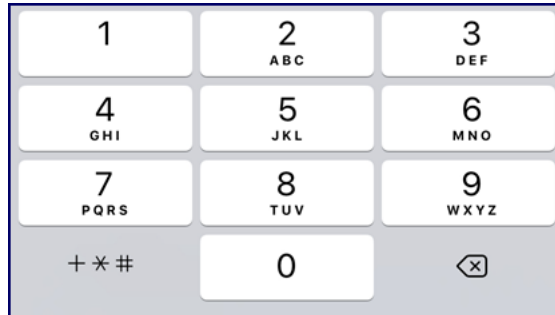
❖ 19.3.11. Tel, URL, and Email Fields

tel

```
<input id="telephone" name="telephone" type="tel">
```

On desktop browsers, you don't really gain anything by using the `tel` input type. As telephone numbers can come in all different formats, there are no constraints on what can be entered here. You could, however, add your own custom validation to all telephone inputs using the `pattern` attribute as demonstrated earlier.

Also, user agents are free to provide a different/better means for filling out input fields based on their type. For example, the iPhone provides a more appropriate interface (presenting the user with the phone keypad) for filling out fields of the `tel` type:



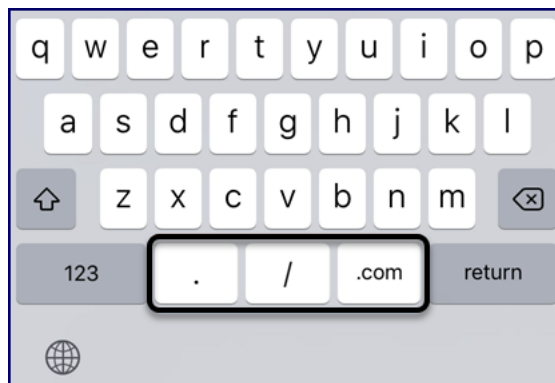
url and email

```
<input id="url" name="url" type="url">  
<input id="email" name="email" type="email">
```

Browsers provide validation for `url` and `email` fields to make sure the user enters valid data.

Also, as with `type="tel"`, user agents are free to provide a different/better means for entering URLs and email addresses.

For example, for `url` types, the iPhone provides keys for `.`, `/` and `.com` and does not provide a **Space** key, as spaces are not allowed in URLs:



For emails, the iPhone provides `@` and `.` keys:

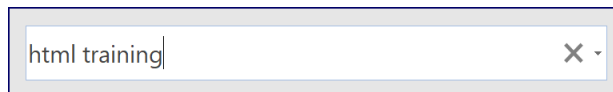


Interestingly enough, the iPhone also provides a “**space**” key for emails. This is because email input types can include a **multiple** attribute, which, when included, allows users to enter multiple emails delimited by spaces. If the iPhone were a little smarter, it would only include the **Space** key when the **multiple** attribute was present.

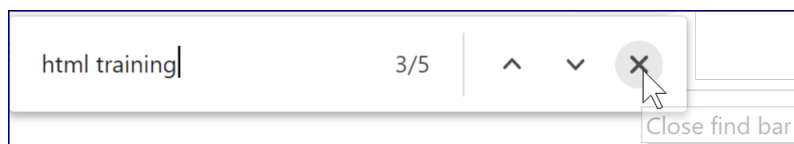
❖ 19.3.12. Search Fields

```
<input id="search" name="search" type="search">
```

Most input fields are meant to be filled out only one time and then submitted for processing. But a search box is a bit different. For example, consider Microsoft Word’s search box:



Notice the **x** used for clearing the box. If you look at search boxes in other applications, you’ll notice many of them also provide a simple way to clear the text. Modern web browsers take the same approach that Word does. Here’s Chrome’s search box:



Note that the **x** doesn’t show up until you have entered some text into the field.

❖ 19.3.13. Hidden Fields

Hidden fields are created with the `input` element with the `type` attribute set to `hidden`. They are used to pass name-value pairs to the server without displaying them to the user. Hidden fields are often used to identify the product being ordered on an e-commerce page. For example:

```
<input type="hidden" name="product-id" id="product-id" value="42">
```

Beware of Hackers

Although the user can't change the value of an input field via the form, savvy users can change the value of any field, including input fields, using the browser's developer tools, so you must always include some sort of server-side validation, meaning you must have code on the server that verifies that the data coming in from the form is valid and safe.



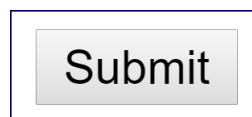
19.4. Buttons

Evaluation
Copy

Submit and reset buttons can both be created with the `<input>` tag.

❖ 19.4.1. Submit Button

A simple submit button looks like this:



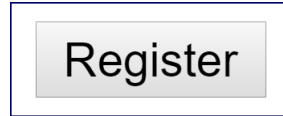
Sample code for a submit button:

```
<input type="submit" name="submitbtn" id="submitbtn">
```

Different browsers format buttons in different ways and some have different default text for submit buttons. Use the `value` attribute to explicitly set the text of the button:

```
<input type="submit" name="submitbtn" id="submitbtn" value="Register">
```

Now the button will appear as follows:



When a form has a submit button, it can be submitted either by clicking the button or by pressing the **Enter** key when an input element has focus.

When a submit button is clicked, the name and value of that button are sent to the server (as a name-value pair). This can be useful in the event that a form has multiple submit buttons as the processing page can be set to behave differently depending on which button is clicked to submit the form.

❖ 19.4.2. Reset Button

A reset button is used to set all the form fields back to their original values. A reset button looks like this:



Most browsers use “Reset” as the default text. While this can be changed with the value attribute, it is generally a better practice to leave the default value unchanged, or even to explicitly set it to “Reset” as users are likely familiar with a standard **Reset** button.

Sample code for a reset button:

```
<input type="reset" name="resetbtn" id="resetbtn" value="Reset">
```

❖ 19.4.3. Button Buttons

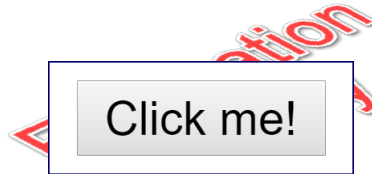
Buttons can also be created using the <button> tag with an optional type attribute, which defaults to “submit” if not present. Other possible values for type are “reset” and “button”.

`<button type="submit">` and `<button type="reset">` can be used interchangeably with `<input type="submit">` and `<input type="reset">`. The text between the opening and closing `<button>` tags shows up on the button.

Sample code for a button button:

```
<button type="button" id="mycustombtn">Click me!</button>
```

This will appear as follows:



Button Buttons and JavaScript

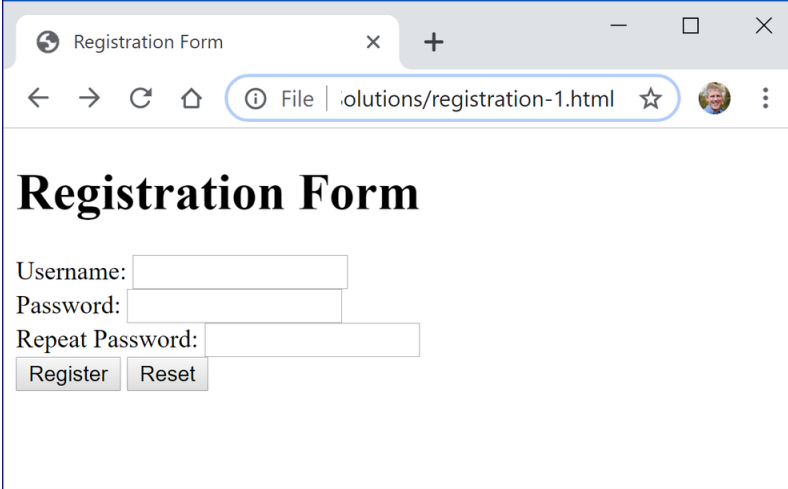
Button buttons are often used in conjunction with JavaScript to add custom behaviors to a web page. Open `forms/Demos/button-buttons.html` in your browser to see some fun things you can do with buttons and JavaScript.

Exercise 35: Creating a Registration Form

🕒 20 to 40 minutes

In this exercise, you will begin to create a registration form.

The form should appear as follows:



Registration Form

Username:

Password:

Repeat Password:

1. Open Forms/Exercises/registration.html for editing.
2. Add a form element to the page.
 - The action should be:
`https://www.webucator.com/materials/htm101/`
 - The method should be “post”.
3. Add the following form elements:
 - Hidden field: name and id should be “secretcode”; value should be “42”.
 - Text field: name and id should be “username”. The input should have a corresponding label.
 - Password field: name and id should be “pw”. The input should have a corresponding label.
 - Repeat password field: name and id should be “pw2”. The input should have a corresponding label.

- Submit button: `value` should be “Register”.
- Reset button.

4. When you are done, open the page in a browser and fill out and submit the form.

You will need internet access to see the resulting page, which will look something like this:

HTML Form Submission

secretcode:	secretcode
username:	jkrowlings
pw:	H@rryP0tt3r
pw2:	H@rryP0tt3r

Note that the resulting page shown above is not something you coded. It is the `action` page we used in the form. This page just dumps back the data submitted in the form. We use it simply to show you that the form does indeed get submitted and the data is now available on the server.

Solution: Forms/Solutions/registration-1.html

```
1.  <!DOCTYPE html>
2.  <html>
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1.0">
6.  <title>Registration Form</title>
7.  </head>
8.  <body>
9.  <h1>Registration Form</h1>
10. <form method="post"
11.     action="https://www.webucator.com/materials/htm101/">
12. <input type="hidden" name="secretcode" id="secretcode" value="42">
13. <div>
14.   <label for="username">Username:</label>
15.   <input type="text" name="username" id="username" size="15">
16. </div>
17. <div>
18.   <label for="pw">Password:</label>
19.   <input type="password" name="pw" id="pw" size="15">
20. </div>
21. <div>
22.   <label for="pw2">Repeat Password:</label>
23.   <input type="password" name="pw2" id="pw2" size="15">
24. </div>
25. <div>
26.   <input type="submit" name="submitbutton" id="submitbutton"
27.     value="Register">
28.   <input type="reset" name="resetbutton" id="resetbutton">
29. </div>
30. </form>
31. </body>
32. </html>
```

Note that we have used `div` elements to group the `label`-`input` pairs.

Be sure to correct any mistakes that you made as you will be starting with your solution to this exercise in the next exercise.



19.5. Checkboxes

Checkboxes are created with the `input` element with the `type` attribute set to “checkbox”. The default value for a checkbox is “on”. Although the value of a checkbox can be changed with the `value` attribute, there is usually no reason to do so, as the name-value pair only gets sent to the server if the checkbox is checked. In other words, the code on the server only needs to check for the existence of the variable name to see if the checkbox was checked or not.

Take a look at the following code:

Toppings:

```
<input type="checkbox" name="sprinkles" id="sprinkles">  
<label for="sprinkles">Sprinkles</label>  
<input type="checkbox" name="nuts" id="nuts">  
<label for="nuts">Nuts</label>  
<input type="checkbox" name="whip" id="whip">  
<label for="whip">Whipped Cream</label>
```

This will appear as follows in the browser:



Toppings: ☐ Sprinkles ☐ Nuts ☐ Whipped Cream

❖ 19.5.1. Checked By Default

Checkboxes can be set to be checked by default using the `checked` attribute:

```
<input type="checkbox" checked name="sprinkles" id="sprinkles">
```

❖ 19.5.2. Making a Checkbox Required

To force the user to check a checkbox (e.g., to accept a user agreement), checkboxes can be set to be required using the `required` attribute:

```
<input type="checkbox" required name="terms" id="terms">  
<label for="terms">Check to indicate that you accept our terms.</label>
```



19.6. Radio Buttons

Radio buttons are created with the `input` element with the `type` attribute set to “radio”. Radio buttons generally come in groups, in which each radio button has the same name. Only one radio button in the group can be checked at any given time. Each radio button in the group should have a unique value – the value to be sent to the server if that radio button is selected.

Take a look at the following code:

```
Cup or Cone?
<label>
  <input type="radio" name="container" value="cup">
  Cup
</label>
<label>
  <input type="radio" name="container" value="plaincone">
  Plain cone
</label>
<label>
  <input type="radio" name="container" value="sugarcone">
  Sugar cone
</label>
<label>
  <input type="radio" name="container" value="wafflecone">
  Waffle cone
</label>
```

This will appear as follows in the browser:

Cup or Cone: ☐ Cup ☐ Plain cone ☐ Sugar cone ☐ Waffle Cone

❖ 19.6.1. Radio Buttons, Labels, and the id Attribute

You will notice that we used `<label>` differently with radio buttons. Instead of using the `for` attribute, we wrapped each radio button in `<label>` tags. This is because our radio buttons don't include `id` attributes.

In form elements such as text fields and checkboxes, the `id` is usually the same as the name. Remember that `ids` and `names` serve different purposes. Again, here is the difference between these attributes:

1. The `name` attribute is used to hold the value of the field when data is sent to the server.

2. The `id` attribute is used by the browser to identify a specific element.

In most cases, it's simplest to use the same value for both the `name` and the `id`. But for radio buttons, this isn't possible, because radio buttons in the same set must all have the same name, but they cannot all have the same `id`. All `id` values must be unique on a given page.

The following code shows radio buttons with `ids`:

Dominant Hand:

```
<input type="radio" name="dom-hand" id="hand-right" value="right">
<label for="hand-right">Right</label>
<input type="radio" name="dom-hand" id="hand-left" value="left">
<label for="hand-left">Left</label>
```

❖ 19.6.2. Checked By Default

Like checkboxes, radio buttons can be set to be checked by default using the `checked` attribute:

```
<input type="radio" name="container" value="cone" checked>
```

❖ 19.6.3. Requiring a Selection


To make a selection required, add the `required` attribute to at least one of the radio buttons in the named group:

Dominant Hand:

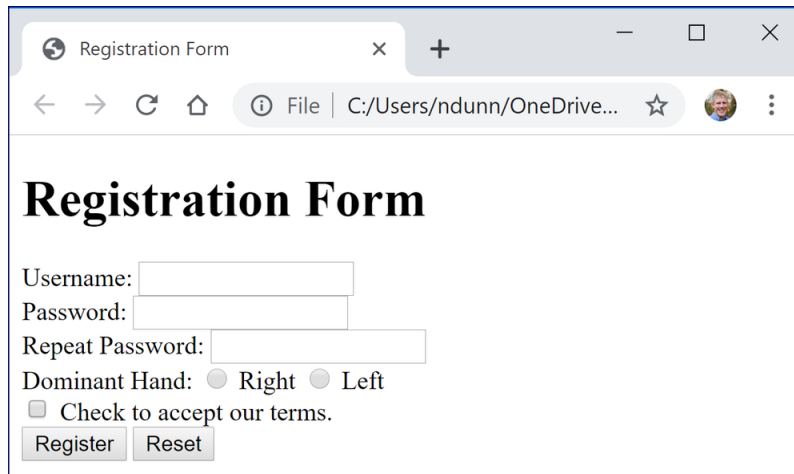
```
<input type="radio" name="dom-hand" id="hand-right" value="right" required>
<label for="hand-right">Right</label>
<input type="radio" name="dom-hand" id="hand-left" value="left" required>
<label for="hand-left">Left</label>
```

There is no effective difference between adding the `required` attribute to one radio button or to all the radio buttons in a named group, but the code is clearer if you add it to all of them. Marking just one as required makes it look like you have to choose that option, which is not the case.

Exercise 36: Adding Checkboxes and Radio Buttons

 10 to 15 minutes

In this exercise, you will add a checkbox and radio buttons to the registration form. On completion, the form should look like this:



The screenshot shows a web browser window with the title 'Registration Form'. The address bar shows the file path 'C:/Users/ndunn/OneDrive...'. The form itself has a title 'Registration Form' and the following elements:

- Username:
- Password:
- Repeat Password:
- Dominant Hand: ☐ Right ☐ Left
- ☐ Check to accept our terms.
- Register
- Reset

1. Open `Forms/Exercises/registration.html` for editing if you don't have it open already.
2. Add the following input elements:
 - A. Two radio buttons:
 - i. The name should be "dom-hand" for both.
 - ii. The id of each should be unique, say "hand-right" and "hand-left".
 - iii. The value should also be unique, say "right" and "left".
 - iv. The user should see the radio buttons labeled as "Right" and "Left", respectively.
 - B. A checkbox: the name and id should both be "terms" and the field should be required.
3. When you are done, open the page in a browser and fill out and submit the form. You shouldn't be able to submit unless you have accepted the terms.

You will need internet access to see the resulting page, which will look something like this:

HTML Form Submission

secretcode:	42
username:	rafanadal
pw:	iL0v3T3nn!\$
pw2:	iL0v3T3nn!\$
dom-hand:	left
terms:	on
submitbutton:	Register

Solution: Forms/Solutions/registration-2.html

```
-----Lines 1 through 24 Omitted-----
25. <div>
26.   Dominant Hand:
27.   <label>
28.     <input type="radio" name="dom-hand" id="hand-right" value="right">
29.     Right
30.   </label>
31.   <label>
32.     <input type="radio" name="dom-hand" id="hand-left" value="left">
33.     Left
34.   </label>
35. </div>
36. <div>
37.   <input type="checkbox" name="terms" id="terms" required>
38.   <label for="terms">Check to accept our terms.</label>
39. </div>
-----Lines 40 through 47 Omitted-----
```

Be sure to correct any mistakes that you made as you will be starting with your solution to this exercise in the next exercise.



19.7. Fieldsets

The `fieldset` element is used to group a set of inputs. The `legend` element is used within a `fieldset` to give the group of elements a legend, which is similar to a caption. Here is a simple example with radio buttons:

Demo 19.3: Forms/Demos/fieldset.html

```
-----Lines 1 through 7 Omitted-----
8.  <body>
9.  <form method="post"
10.    action="https://www.webucator.com/materials/htm101/">
11.    <fieldset>
12.      <legend>Cup or Cone*:</legend>
13.      <label>
14.        <input type="radio" name="container" value="cup" required>
15.        Cup
16.      </label>
17.      <label>
18.        <input type="radio" name="container" value="plaincone"
19.          required>
20.        Plain cone
21.      </label>
22.      <label>
23.        <input type="radio" name="container" value="sugarcone"
24.          required>
25.        Sugar cone
26.      </label>
27.      <label>
28.        <input type="radio" name="container" value="wafflecone"
29.          required>
30.        Waffle Cone
31.      </label>
32.    </fieldset>
33.    <input name="submitbutton" id="submitbutton" type="submit">
34.    <input name="resetbutton" id="resetbutton" type="reset">
35.  </form>
-----Lines 36 through 37 Omitted-----
```

And here is how it appears in Google Chrome:



19.8. Select Menus

Select menus are created with the `select` element, which must contain one or more `option` elements. The text between the opening and closing `<option>` tags appears in the select menu. The value of the `option` element's `value` attribute is what gets passed to the server if that option is selected.

Here is a simple example:

```
<label for="flavor">Flavor:</label>
<select name="flavor" id="flavor">
  <option value="chocolate">Chocolate</option>
  <option value="strawberry">Strawberry</option>
  <option value="vanilla">Vanilla</option>
</select>
```

And here is how it appears in Google Chrome:

The `<select>` and `<option>` tags' attributes are listed below:

select Attributes

1. **size** – Number of options to appear at once.
2. **multiple** – Indicates that multiple options can be selected. Value must be “multiple” or left out.
3. **required** – When present, the user must select an option that has a value at least one character long.

option Attributes

1. **value** – Value to send to server if option is selected.
2. **selected** – Indicates that option is pre-selected. Value must be “selected” or left out.

❖ 19.8.1. Option Groups

Options can be arranged in groups using the `<optgroup>` tag. The `label` attribute is used to set the option group heading.

```
<label for="flavor">Flavor:</label>
<select name="flavor" id="flavor">
  <option value="0"></option>
  <optgroup label="Soft Flavors">
    <option value="softChoc">Chocolate</option>
    <option value="softStraw">Strawberry</option>
    <option value="softVan">Vanilla</option>
  </optgroup>
  <optgroup label="Hard Flavors">
    <option value="hardChoc">Chocolate</option>
    <option value="hardStraw">Strawberry</option>
    <option value="hardVan">Vanilla</option>
    <option value="hardMint">Mint Chocolate Chip</option>
    <option value="hardCC">Cookies & Cream</option>
  </optgroup>
</select>
```

This will appear as follows in Google Chrome:

Flavor:

Soft Flavors

Chocolate

Strawberry

Vanilla

Hard Flavors

Chocolate

Strawberry

Vanilla

Mint Chocolate Chip

Cookies & Cream

Evaluation Copy

19.9. Textareas

Textareas are created with the `<textarea>` tag. Take a look at the following code:

```
<label for="requests">Special Requests:</label><br>
<textarea name="requests" id="requests" cols="40" rows="6"></textarea>
```

This would appear as follows after the user has entered data:

Special Requests:

I want a really big cone!

The `cols` and `rows` attributes indicate the number of columns and rows (in characters) that the `textarea` should span.

An initial value can be entered into the `textarea` by adding text between the opening and closing `<textarea>` tags. For example:

```
<textarea name="requests" cols="40" rows="6">Hello, there!</textarea>
```

This is often done when a user submits the form with errors. The server-side code then returns the form already populated with the previously entered values, so the user doesn't have to re-enter them.

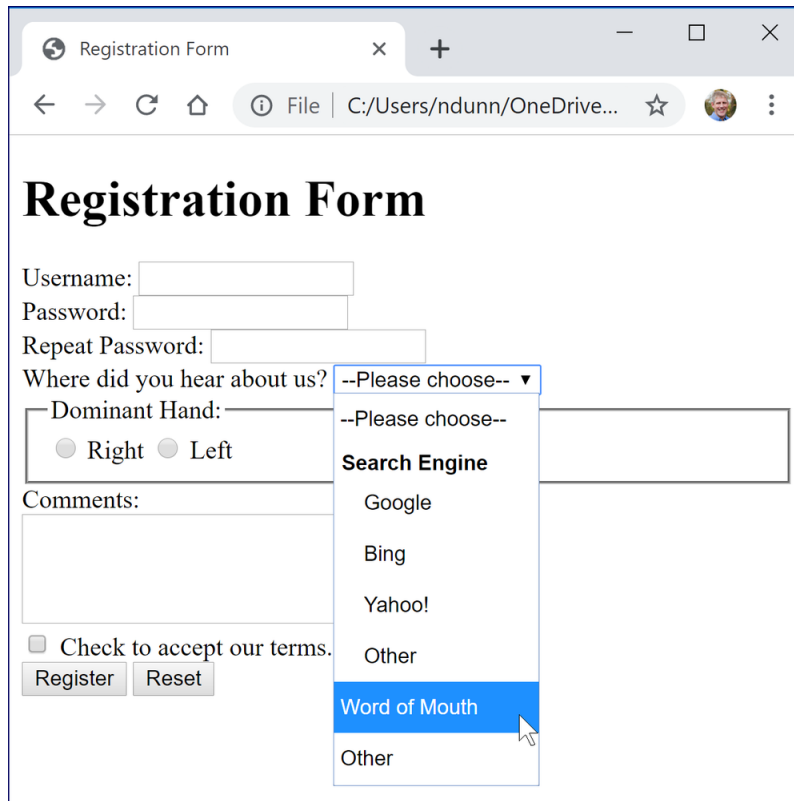
The `textarea` element's attributes are shown below:

1. `name` – Variable name.
2. `cols` – Width of `textarea` in average width of characters.
3. `rows` – Height of `textarea` in number of lines.
4. `minlength` – Minimum number of characters that must be entered.
5. `maxlength` – Maximum number of characters that can be entered.
6. `pattern` – A regular expression expressing a valid value for the field.
7. `placeholder` – A placeholder value that will disappear as soon as data is entered into the field.
8. `required` – When present, the user must fill in a value before submitting the form.

Exercise 37: Adding a Select Menu and a Textarea

🕒 15 to 25 minutes

In this exercise, you will add a select menu and textarea to the registration form. On completion, the form should look like this:



Registration Form

Username:

Password:

Repeat Password:

Where did you hear about us? **--Please choose--** ▼

Dominant Hand: ☐ Right ☐ Left

Comments:

☐ Check to accept our terms.

Search Engine

- Google
- Bing
- Yahoo!
- Other
- Word of Mouth**
- Other

1. Open `Forms/Exercises/registration.html` for editing if you don't have it open already.
2. Add the following form elements:
 - A. `select`: name and id should be "referral". Options should be broken up as follows (refer to the screenshot above):
 - i. "--Please choose--" with value of "0"
 - ii. An option group with the label of "Search Engine" and the following options:
 - a. "Google" with value of "Google"

- b. “Bing” with value of “Bing”
 - c. “Yahoo!” with value of “Yahoo”
 - d. “Other” with value of “OtherSearchEngine”
- iii. “Word of Mouth” with value of “wom”
 - iv. “Other” with value of “other”

B. `textarea`: name and id should be “comments”. Be sure to include `cols` and `rows` attributes.

3. Put the “dominant hand” radio buttons in a `fieldset`.

When you are done, open the page in a browser and fill out and submit the form. You will need internet access to see the resulting page, which will look something like this:

HTML Form Submission	
secretcode:	42
username:	bspringsteen
pw:	!m0nF1r3
pw2:	!m0nF1r3
referral:	wom
dom-hand:	right
comments:	Please deliver my cone to Asbury Park, NJ.
terms:	on
submitbutton:	Register

Solution: Forms/Solutions/registration-3.html

```
-----Lines 1 through 24 Omitted-----
25. <div>
26.   <label for="referral">Where did you hear about us?</label>
27.   <select name="referral" id="referral">
28.     <option value="0">--Please choose--</option>
29.     <optgroup label="Search Engine">
30.       <option value="Google">Google</option>
31.       <option value="Bing">Bing</option>
32.       <option value="Yahoo">Yahoo!</option>
33.       <option value="OtherSearchEngine">Other</option>
34.     </optgroup>
35.     <option value="wom">Word of Mouth</option>
36.     <option value="other">Other</option>
37.   </select>
38. </div>
39. <div>
40.   <fieldset>
41.     <legend>Dominant Hand:</legend>
42.     <label>
43.       <input type="radio" name="dom-hand" id="hand-right" value="right">
44.       Right
45.     </label>
46.     <label>
47.       <input type="radio" name="dom-hand" id="hand-left" value="left">
48.       Left
49.     </label>
50.   </fieldset>
51. </div>
52. <div>
53.   <label for="comments">Comments:</label><br>
54.   <textarea name="comments" id="comments"
55.     cols="40" rows="4"></textarea>
56. </div>
-----Lines 57 through 68 Omitted-----
```



19.10. HTML Forms and CSS

As with all elements, forms can be laid out better and made to look much prettier with CSS. For example, here is a completed ice cream form without CSS:

Username: Email:

Telephone:

Cup or Cone:
☒ Cup ☐ Plain cone ☐ Sugar cone ☐ Waffle Cone

Flavor:

Toppings:
☐ Sprinkles ☐ Nuts ☒ Whipped Cream


Special Requests:

☐ I understand that I'm really not going to get any ice cream.

And here is the same form with CSS added:

Username:*

Email:*



Telephone:

Cup or Cone:*

☒ Cup ☐ Plain cone ☐ Sugar cone ☐ Waffle Cone

Flavor:*

Toppings:

☐ Sprinkles ☐ Nuts ☒ Whipped Cream

Special Requests:

☐ I understand that I'm really not going to get any ice cream. *

Conclusion

In this lesson, you have learned to create HTML forms.

LESSON 20

JavaScript Form Validation

Topics Covered

- ☑ Accessing data entered by users in forms.
- ☑ Validating text fields, textareas, radio buttons, checkboxes, and select menus.
- ☑ Writing clean, reusable validation functions.

Introduction

In this lesson, you will learn to validate forms with HTML and JavaScript.



20.1. Server-side Form Validation

In most cases, when a user submits a registration form, a login form, a purchase form, or any other type of form, the data gets sent to a web server for processing. That data should be validated as soon as it hits the server, before doing anything else with it. This is true even if the data has already been validated on the client (the browser). That's because, it is impossible for the server to know that the client-side validation actually occurred. Nefarious hackers can easily get around JavaScript validation. In fact, it can be as simple as turning JavaScript off in the browser. To see how easy that is, simply google "Turn off JavaScript in Google Chrome" or in whichever browser you want to test. Server-side validation can be done with PHP, Java, Python, or any server-side programming language.

So, is it worth validating the code on the client if you have to do it again on the server anyway? Yes, it is. Here's why:

1. **Better user experience.** Client-side validation is practically immediate. In fact, it can be done as the user types. So, the user gets immediate feedback as to what needs to be corrected.
2. **Less work for your server.** If the data has already been validated on the client, then it is likely to pass server-side validation as well, meaning your server will only have to check it once.



20.2. HTML Form Validation

HTML form fields include validation attributes, such as `required`, `pattern`, `min` and `max`, and `minlength` and `maxlength`. In addition, setting the type to “email” or “url” forces the user to enter a valid email or URL. These are called constraints. While adding HTML constraints is very simple, it is difficult to control the user experience. To illustrate, let’s look at an ice cream order form:

Demo 20.1: FormValidation/Demos/ice-cream.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>Ice Cream Order Form</title>
9.  </head>
10. <body>
11. <form id="ice-cream-form" method="post">
12.   <label for="username">Username*:</label>
13.   <input type="text" id="username" name="username"
14.     required minlength="8" maxlength="25">
15.   <label for="email">Email*:</label>
16.   <input type="email" id="email" name="email" required>
17.   <label for="phone">Telephone:</label>
18.   <input type="tel" id="phone" name="phone"
19.     pattern="[1-9]\d{2}-\d{3}-\d{4}">
20.
21.   <fieldset>
22.     <legend>Cup or Cone*:</legend>
23.     <label>
24.       <input type="radio" name="container" value="cup" required>
25.       Cup
26.     </label>
27.     <label>
28.       <input type="radio" name="container" value="plaincone" required>
29.       Plain cone
30.     </label>
31.     <label>
32.       <input type="radio" name="container" value="sugarcone" required>
33.       Sugar cone
34.     </label>
35.     <label>
36.       <input type="radio" name="container" value="wafflecone" required>
37.       Waffle Cone
38.     </label>
39.   </fieldset>
40.
41.   <label for="flavor">Flavor*:</label>
42.   <select name="flavor" id="flavor" required>
43.     <option value="0" selected="selected">--Please Choose--</option>
44.     <optgroup label="Soft Flavors">
```

```

45.     <option value="softChoc">Chocolate</option>
46.     <option value="softStraw">Strawberry</option>
47.     <option value="softVan">Vanilla</option>
48. </optgroup>
49. <optgroup label="Hard Flavors">
50.     <option value="hardChoc">Chocolate</option>
51.     <option value="hardStraw">Strawberry</option>
52.     <option value="hardVan">Vanilla</option>
53.     <option value="hardMint">Mint Chocolate Chip</option>
54.     <option value="hardCC">Cookies & Cream</option>
55. </optgroup>
56. </select>
57.
58. <fieldset id="toppings">
59.     <legend>Toppings:</legend>
60.     <input type="checkbox" name="sprinkles" id="sprinkles">
61.     <label for="sprinkles">Sprinkles</label>
62.     <input type="checkbox" name="nuts" id="nuts">
63.     <label for="nuts">Nuts</label>
64.     <input type="checkbox" name="whip" id="whip">
65.     <label for="whip">Whipped Cream</label>
66. </fieldset>
67.
68. <label for="requests">Special Requests:</label>
69. <textarea name="requests" id="requests"
70.     minlength="10" maxlength="200"></textarea><br>
71.
72. <input type="checkbox" name="terms" id="terms" required>
73. <label for="terms">
74.     I understand that I'm really not going to get any ice cream.
75. </label>
76.
77. <button>Submit</button>
78. </form>
79. </body>
80. </html>

```

Code Explanation

We have added some basic CSS to style the form:

Ice Cream Order Form

file:///Users/natdunn/...

Username*:

Email*:

Telephone:

Cup or Cone*:
☐ Cup ☐ Plain cone ☐ Sugar cone ☐ Waffle Cone

Flavor*:

Toppings:
☐ Sprinkles ☐ Nuts ☐ Whipped Cream

Special Requests:

☐ I understand that I'm really not going to get any ice cream.

Submit

This form has no JavaScript validation, but the form fields do include HTML constraints:

1. The `username` must be between 8 and 25 characters and is required.
2. The `email` must be a valid email address and is required.
3. The phone must match a regular expression⁵⁸ if it is entered. It is not required.

58. Regular expressions are used by many programming languages for pattern matching.

4. A container selection is required.
5. A flavor selection is required.
6. A requests entry must be between 10 and 200 characters if it is entered. It is not required.
7. The terms checkbox must be checked.

While this is functional, it's not very user friendly. Watch what happens when we submit the form without entering any data:

A screenshot of a web form with the following fields: Username*, Email*, Telephone, and Cup or Cone*. The Username* field is highlighted with a blue border, and an error message "Please fill out this field." with an exclamation mark icon is displayed next to it. The Email* field is also empty. The Telephone field is empty. The Cup or Cone* field has four radio button options: Cup, Plain cone, Sugar cone, and Waffle Cone. A large red "Evaluation Copy" watermark is visible across the center of the image.

Notice that, while both the username and the email fields are invalid, we only get an error for the username field.

Now let's fill in a username and resubmit:

A screenshot of the same web form after the Username* field has been filled with "webucator". The Email* field is now highlighted with a blue border, and an error message "Please fill out this field." with an exclamation mark icon is displayed next to it. The Telephone field is empty. The Cup or Cone* field has four radio button options: Cup, Plain cone, Sugar cone, and Waffle Cone. A large red "Evaluation Copy" watermark is visible across the center of the image.

This time we get an error on the email field, but not on any of the other fields.

So, the user only finds out one error at a time and then must attempt to submit after each correction.

JavaScript to the rescue! With JavaScript, we can enhance and customize the validation of forms.



20.3. Accessing Form Data

All forms on a web page are stored in the `document.forms[]` array. The first form on a page is `document.forms[0]`, the second form is `document.forms[1]`, and so on. However, it is usually easier to reference forms via their `id` attribute and refer to them that way. For example, a form with `id login-form` can be referenced as `document.getElementById('login-form')`. The major advantage of referencing forms by `id` is that the forms can be repositioned on the page without affecting the JavaScript.

Elements within a form are properties of that form and can be referenced as follows, where `elementName` is the value of the element's `name` attribute:

```
document.getElementById('login-form').elementName
```

For example, given the following form:

```
<form id="login-form">
  <label for="uname">Username:</label>
  <input type="text" name="username" id="uname">
  <input type="submit" value="Log in">
</form>
```

You can access the `username` field like this:

```
const username = document.getElementById('login-form').username;
```

As with all elements, you can also access a field by its `id`:

```
const username = document.getElementById('uname');
```

Names and IDs

It is common to use the same value for both the name and id of form fields. In the example above, we use different values simply to distinguish between the examples of accessing the field by name and id.

All text-like fields (e.g., text, password, url, email, tel, etc.) have a value property that holds the text value of the field. Take a look at the following example:

Demo 20.2: FormValidation/Demos/text-like-fields.html

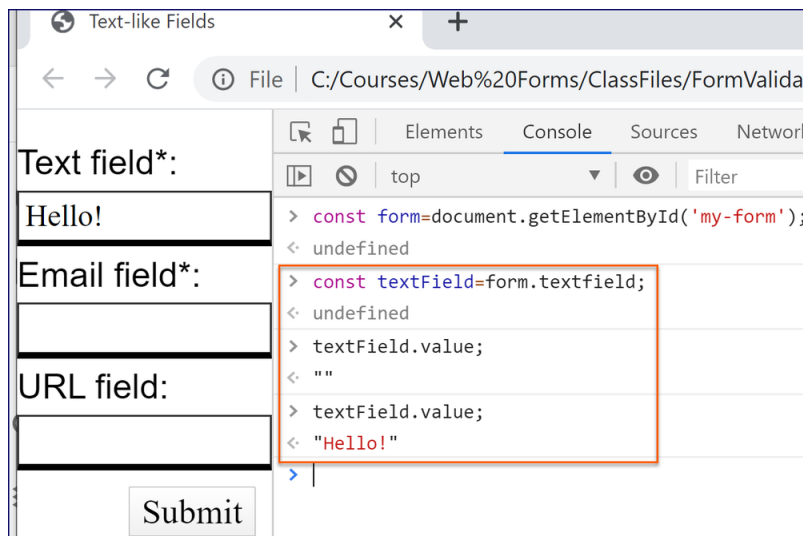
```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <title>Text-like Fields</title>
9.  </head>
10. <body>
11. <form id="my-form" method="post" novalidate>
12.   <label for="textfield">Text field*:</label>
13.   <input type="text" id="textfield" name="textfield"
14.     required minlength="5" maxlength="10">
15.   <label for="emailfield">Email field*:</label>
16.   <input type="email" id="emailfield" name="emailfield" required>
17.   <label for="urlfield">URL field:</label>
18.   <input type="url" id="urlfield" name="urlfield" pattern="https:.*">
19.
20.   <button>Submit</button>
21. </form>
22. </body>
23. </html>
```


Code Explanation

1. Open `FormValidation/Demos/text-like-fields.html` in the browser, open the console, and type:

```
const form = document.getElementById('my-form');
const textField = form.textfield;
textField.value;
```

2. You will see the above code outputs an empty string. That is because the field has no text in it.
3. Now type “Hello!” in the text field and type `textField.value;` in the console again. This time, `textField.value` returns “Hello!”:



4. All values of text-like fields can be accessed in this way. Try it with `emailfield` and `urlfield`.



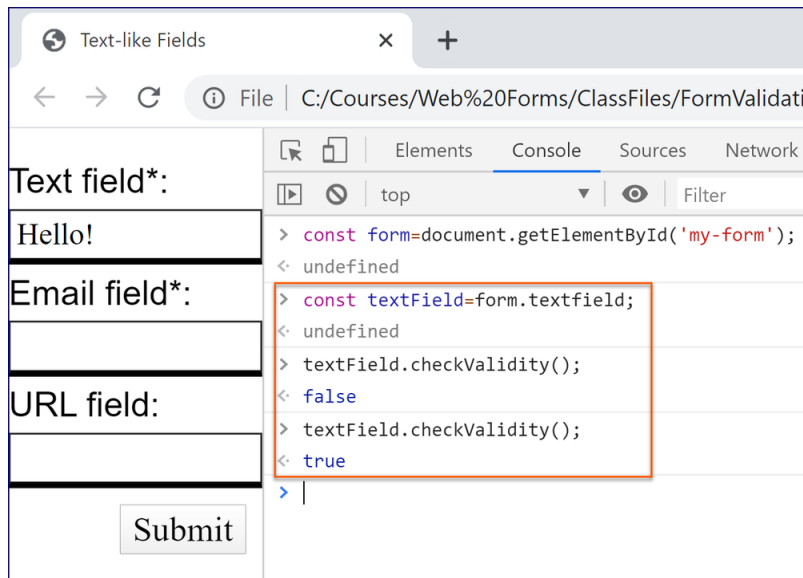
20.4. Form Validation with JavaScript

The first step to using JavaScript to validate your forms is to turn off the default HTML validation. You do this by adding the `novalidate` attribute to the `<form>` tag:

```
<form id="my-form" method="post" novalidate>
```

This prevents the default HTML validation from happening when the form is submitted, so you can use JavaScript to create custom validation.

To check if a field's value is valid, use the `checkValidity()` method of the field. In the following screenshot, we first ran `textField.checkValidity()`; when the field was empty and then ran it again after typing "Hello!" into the text field:



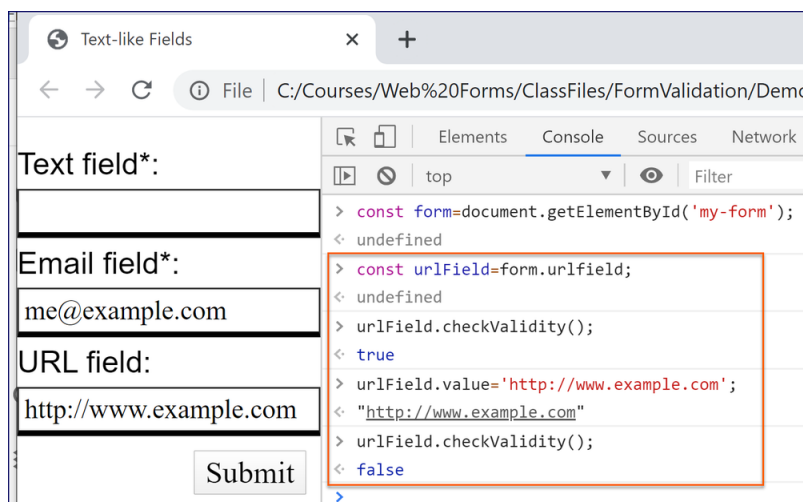
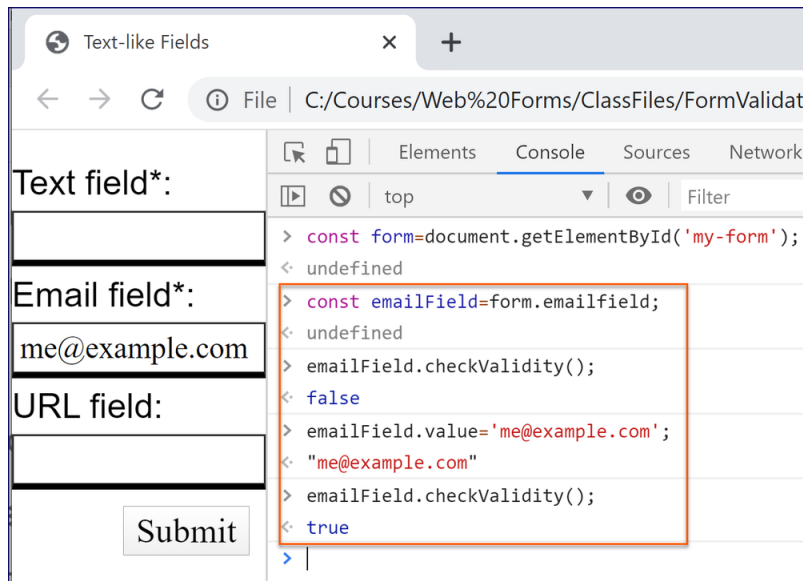


Exercise 38: Checking the Validity of the Email and URL Fields

⌚ 5 to 10 minutes

1. From your Demos folder, open `text-like-fields.html` in your browser, if it's not open already.
2. In the console, check the validity of the `emailfield` and `urlfield` fields. If the field is **valid** to start, modify the field value to make it **invalid** and check it again. If it is **invalid** to start, modify the field value to make it **valid** and check it again.
3. Note that `emailfield` is required and must be a well-formed email, and `urlfield` is not required, but if entered, it must be a valid URL beginning with "https:".

Solution



The url field isn't valid in the last example, because it starts with "http:" and the pattern requires that it starts with "https:".



20.5. Checking Validity on Input and Submit Events

For the best user experience, in addition to checking all fields when the user submits the form, we should check the validity of each field when the user changes the value of that field. To do this, we will

listen for submit events on the form, and input events on the text-like fields. We will add those listeners as soon as the page loads. We can use the following function to check the validity of a field and turn the background color of the control to pink if the entry is invalid:

```
function checkField(field) {
  if (!field.checkValidity()) {
    field.style.backgroundColor = 'pink';
  } else {
    field.style.backgroundColor = '';
  }
}
```

We will call the `checkField()` function when the user inputs a value in the field we want to check and when the user submits the form. To do that, we add our event listeners:

```
window.addEventListener('load', function(e) {
  const form = document.getElementById('my-form');
  const textField = form.textfield;

  textField.addEventListener('input', function(e) {
    checkField(textField);
  });

  form.addEventListener("submit", function(e) {
    // Check errors
    checkField(textField);

    // If form is invalid, prevent submission
    if (!form.checkValidity()) {
      e.preventDefault();
      alert('Please fix form errors.');
```

Evaluation
Copy

A few things to note:

1. The event we are listening for is an input event. This fires whenever the value of an input, select, or textarea changes; however, at the time of this writing, it is only reliable for text-like input and textarea fields.
 - A. For checkboxes and radio buttons, it is better to listen for click events.

- B. For `select` fields, it is better to listen for change events.
2. For now, we're just changing the background color of invalid fields to pink. Later, we'll add useful messages.
 3. form elements also take the `checkValidity()` method, which returns `false` if any of the form's fields are invalid. Otherwise, it returns `true`.
 4. The `preventDefault()` method of an event is used to prevent what would normally happen when that event occurs. In this case, the event is `submit`, which would normally cause the form to be submitted. To prevent that from happening when the form is invalid, we call `e.preventDefault()`.
 5. We include an `alert()` call only for illustration purposes. In practice, you probably don't need the alert. You can just show the errors in the form.

Input Event

The `input` event is meant to be fired when the user changes the value of `input`, `select`, and `textarea` elements, but at the time of this writing, it is not yet fully reliable. Check <https://caniuse.com/#search=input> for information on current browser support.

Below, we have the complete file:

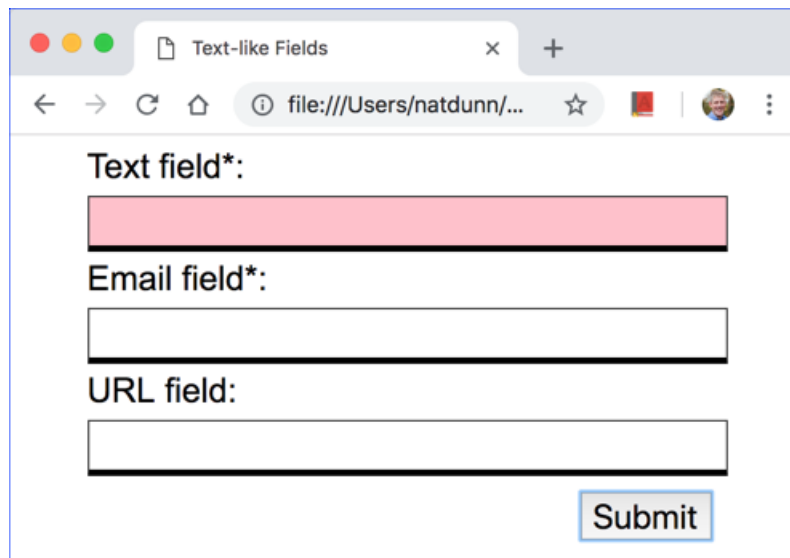
Demo 20.3: FormValidation/Demos/text-like-fields-validate.html

```
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 checkField(field) {
10.         if (!field.checkValidity()) {
11.             field.style.backgroundColor = 'pink';
12.         } else {
13.             field.style.backgroundColor = '';
14.         }
15.     }
16.
17.     window.addEventListener('load', function(e) {
18.         const form = document.getElementById('my-form');
19.         const textField = form.textfield;
20.
21.         textField.addEventListener('input', function(e) {
22.             checkField(textField);
23.         });
24.
25.         form.addEventListener('submit', function(e) {
26.             // Check errors
27.             checkField(textField);
28.
29.             // If form is invalid, prevent submission
30.             if (!form.checkValidity()) {
31.                 e.preventDefault();
32.                 alert('Please fix form errors.');
```

```
45.     <input type="email" id="emailfield" name="emailfield" required>
46.     <label for="urlfield">URL field:</label>
47.     <input type="url" id="urlfield" name="urlfield" pattern="https:.*">
48.
49.     <button>Submit</button>
50. </form>
51. </body>
52. </html>
```

Code Explanation

1. Open `FormValidation/Demos/text-like-fields-validate.html` in your browser and submit the form without filling in any of the fields. The text field should turn pink:



2. Enter a value of at least 5 characters in the text field. You will see that the pink background goes away.

Now let's add validation for the email and url fields:

Demo 20.4: FormValidation/Demos/text-like-fields-validate2.html

```
-----Lines 1 through 16 Omitted-----
17. window.addEventListener('load', function(e) {
18.     const form = document.getElementById('my-form');
19.     const textField = form.textfield;
20.     const emailField = form.emailfield;
21.     const urlField = form.urlfield;
22.
23.     textField.addEventListener('input', function(e) {
24.         checkField(textField);
25.     });
26.
27.     emailField.addEventListener('input', function(e) {
28.         checkField(emailField);
29.     });
30.
31.     urlField.addEventListener('input', function(e) {
32.         checkField(urlField);
33.     });
34.
35.     form.addEventListener('submit', function(e) {
36.         // Check errors
37.         checkField(textField);
38.         checkField(emailField);
39.         checkField(urlField);
40.
41.         // If form is invalid, prevent submission
42.         if (!form.checkValidity()) {
43.             e.preventDefault();
44.             alert('Please fix form errors.');
```

```
-----Lines 48 through 64 Omitted-----
```

Code Explanation

1. Notice that we reuse the `checkField()` function without modification.
2. Open `FormValidation/Demos/text-like-fields-validate2.html` in your browser and submit the form without filling in any of the fields. You will notice that only the Text and Email fields appear pink. That's because the URL field is not required.

3. Enter an invalid URL in the URL field (e.g., “hello”) and resubmit. This time all three fields will appear pink.



20.6. Adding Error Messages

There are many ways to add error messages to the form. One common practice is to have the error messages within the HTML, but to hide them until it is time to report an error. You could do this by setting the `display` property to “none” by default and then changing it to “block” to report the error. One downside to this method is that it makes the HTML messy.

Instead of adding error messages to our HTML that may never be shown to the user, we will use JavaScript to dynamically add elements containing error messages.

❖ 20.6.1. The dataset Property

HTML elements can take a `dataset` property to associate non-standard data with an element. We will use this to assign error messages to our form fields. The following code illustrates this:

```
const form = document.getElementById('my-form');
const textField = form.textfield;
textField.dataset.errorMsg = 'You must enter a value.';
```

Sometimes it is useful to use other properties of the field in the error message. For example, our `textField` field is created with the following HTML:

```
<input type="text" id="textfield" name="textfield"
  required minlength="5" maxlength="10">
```

Below we use the `minLength` and `maxLength` properties of `textField` to create the error message. Note that JavaScript uses camelCase for the property names (e.g., `minLength` and `maxLength`):

```
textField.dataset.errorMsg = 'Your entry must be between ' +
  textField.minLength + ' and ' +
  textField.maxLength + ' characters.';
```

Now we’ll look at how we can report these custom errors to the user in a friendly way:

Demo 20.5: FormValidation/Demos/text-like-fields-add-error1.html

```
-----Lines 1 through 7 Omitted-----
8.     <script>
9.         function addError(field) {
10.            const error = document.createElement('div');
11.            error.innerHTML = field.dataset.errorMsg;
12.            error.className = 'error';
13.            field.parentNode.insertBefore(error, field);
14.        }
15.
16.        function removeError(field) {
17.            // to do
18.        }
19.
20.        function checkField(field) {
21.            if (!field.checkValidity()) {
22.                addError(field);
23.            } else {
24.                removeError(field);
25.            }
26.        }
27.
28.        window.addEventListener('load', function(e) {
29.            const form = document.getElementById('my-form');
30.            const textField = form.textfield;
31.            textField.dataset.errorMsg = 'Your entry must be between ' +
32.            textField.minLength + ' and ' + textField.maxLength +
33.            ' characters.';
34.
35.            const emailField = form.emailfield;
36.            emailField.dataset.errorMsg = 'You must enter a valid email.';
37.
38.            const urlField = form.urlfield;
39.            urlField.dataset.errorMsg = 'The URL must begin with "https."';
40.
41.            textField.addEventListener('input', function(e) {
42.                checkField(textField);
43.            });
44.
45.            emailField.addEventListener('input', function(e) {
46.                checkField(emailField);
47.            });
48.
49.            urlField.addEventListener('input', function(e) {
50.                checkField(urlField);
```

```
51.     });
52.
53.     form.addEventListener('submit', function(e) {
54.         // Check errors
55.         checkField(textField);
56.         checkField(emailField);
57.         checkField(urlField);
58.
59.         // If form is invalid, prevent submission
60.         if (!form.checkValidity()) {
61.             e.preventDefault();
62.             alert('Please fix form errors.');
```

-----Lines 67 through 82 Omitted-----

Code Explanation

1. Notice that we have added `errorMsg` properties to the `dataset` properties of our field elements on the `load` event.
2. Notice that we have changed `checkField()` to call `addError(field)` and `removeError(field)` instead of setting and unsetting a pink background.
3. Open `FormValidation/Demos/text-like-fields-add-error1.html` in your browser and submit the form without filling in any of the fields. After seeing an alert warning you that there are errors, you should see those errors appear right above the two empty required input fields:

Text field*:

Your entry must be between 5 and 10 characters.

Email field*:

You must enter a valid email.

URL field:

Submit

4. The `addError()` function takes one argument: the `field` to which to add an error message. That field should be sent to the function with an `errorMsg` property in its `dataset`. The function does the following:

- A. Creates a new `div` element and assigns it to `error`:

```
const error = document.createElement('div');
```

- B. Sets the `innerHTML` of the error to the field's `errorMsg`:

```
error.innerHTML = field.dataset.errorMsg;
```

- C. Sets the `className` of `error` to `'error'`:

```
error.className = 'error';
```

- D. Inserts the new error element before the passed-in `field`, so that the error shows up right above the field itself.

5. Now, type “hello” into the `textfield` input:

Text field*:

Your entry must be between 5 and 10 characters.

Your entry must be between 5 and 10 characters.

Your entry must be between 5 and 10 characters.

Your entry must be between 5 and 10 characters.

Your entry must be between 5 and 10 characters.

hello

Email field*:

6. It validates the field with each user input. And each time it finds the field invalid, it adds another error div. To prevent that from happening, we should check to see if the error div already exists and, if it does, exit the function without adding another error div. The following example does that:

Demo 20.6: FormValidation/Demos/text-like-fields-add-error2.html

```

-----Lines 1 through 8 Omitted-----
9.     function addError(field) {
10.        if (field.previousElementSibling &&
11.            field.previousElementSibling.className === 'error') {
12.            // error message already showing
13.            return;
14.        }
15.        const error = document.createElement('div');
16.        error.innerHTML = field.dataset.errorMsg;
17.        error.className = 'error';
18.        field.parentNode.insertBefore(error, field);
19.    }
-----Lines 20 through 85 Omitted-----

```

Code Explanation

Now the `addError()` function first checks to see if the passed-in field has a `previousElementSibling` and, if it does, if that element's `className` is "error". If it is, that means the error div is already present, so we return. We don't actually return anything. We are simply exiting the function and effectively passing the ball back to the code that called the function.

Open `FormValidation/Demos/text-like-fields-add-error2.html` in your browser to test it out.

Now we need to write our `removeError()` function so that we can remove the error when the user corrects the problem:

Demo 20.7: FormValidation/Demos/text-like-fields-remove-error.html

```
-----Lines 1 through 8 Omitted-----
9.     function addError(field) {
10.    if (field.previousElementSibling &&
11.        field.previousElementSibling.className === 'error') {
12.        // error message already showing
13.        return;
14.    }
15.    const error = document.createElement('div');
16.    error.innerHTML = field.dataset.errorMsg;
17.    error.className = 'error';
18.    field.parentNode.insertBefore(error, field);
19.  }
20.
21.  function removeError(field) {
22.    if (field.previousElementSibling &&
23.        field.previousElementSibling.className === 'error') {
24.        field.previousElementSibling.remove();
25.    }
26.  }
-----Lines 27 through 88 Omitted-----
```

Code Explanation

1. Like with the `addError()` function, the `removeError()` function first checks to see if the passed-in field has a `previousElementSibling` and, if it does, if that element's `className` is "error". If it is, that means the error div is present, so we remove it using the `remove()` method.
 2. Now, type "Hello" into the `textfield` input. The error should disappear as soon as the entry is valid.
 3. Add and remove values from the other fields as well to see if they work as expected.
-



20.7. Validating Textareas

Validating `textarea` elements is similar to validating text-like input fields. In the following code sample we have added a `textarea`, which is not required, but if included must be between 10 and 200 characters:

Demo 20.8: FormValidation/Demos/text-like-fields-complete.html

```
-----Lines 1 through 46 Omitted-----
47.     const urlField = form.urlfield;
48.     urlField.dataset.errorMsg = 'The URL must begin with "https."';
49.
50.     const comments = form.comments;
51.     comments.dataset.errorMsg = 'Your comment must be between ' +
52.         comments.minLength + ' and ' + comments.maxLength +
53.         ' characters.';
54.
55.     textField.addEventListener('input', function(e) {
56.         checkField(textField);
57.     });
58.
59.     emailField.addEventListener('input', function(e) {
60.         checkField(emailField);
61.     });
62.
63.     urlField.addEventListener('input', function(e) {
64.         checkField(urlField);
65.     });
66.
67.     comments.addEventListener('input', function(e) {
68.         checkField(comments);
69.     });
70.
71.     form.addEventListener('submit', function(e) {
72.         // Check errors
73.         checkField(textField);
74.         checkField(emailField);
75.         checkField(urlField);
76.         checkField(comments);
77.
78.         // If form is invalid, prevent submission
79.         if (!form.checkValidity()) {
80.             e.preventDefault();
81.             alert('Please fix form errors.');
```



```
82.         }
83.     });
84. });
-----Lines 85 through 97 Omitted-----
98.     <label for="comments">Comments:</label>
99.     <textarea name="comments" id="comments"
100.         minlength="10" maxlength="200"></textarea><br>
```

101.

-----Lines 102 through 105 Omitted-----

Code Explanation

The image belows shows how the form appears when all fields are invalid:

The screenshot shows a web browser window with the title "Text-like Fields". The address bar shows a file path: "file:///Users/natdun...". The form contains four input fields, each with a red error message above it:

- Text field*:** "Your entry must be between 5 and 10 characters."
- Email field*:** "You must enter a valid email."
- URL field:** "You must enter a valid URL beginning with \"https\"." The input contains "http://foo".
- Comments:** "Your comment must be between 10 and 200 characters." The input contains "hello".

A "Submit" button is located at the bottom right of the form.




20.8. Validating Checkboxes

Like the text-like fields we've been looking at, checkboxes are created with the `input` tag. We can reuse the same `checkField()` function we have created; however, rather than listening for `input` events, we will listen for `click` events.

Demo 20.9: FormValidation/Demos/checkbox.html

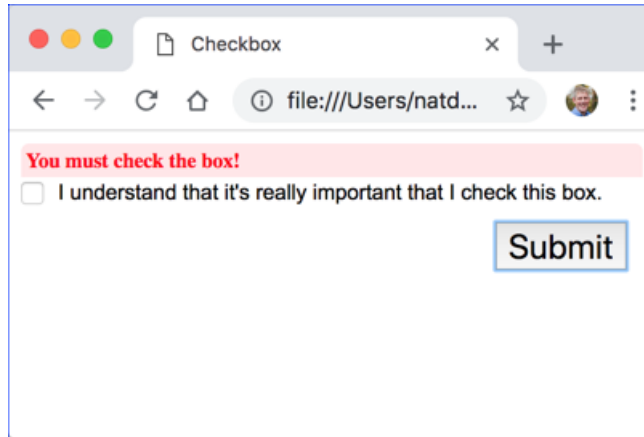
```
-----Lines 1 through 35 Omitted-----
36.   window.addEventListener('load', function(e) {
37.       const form = document.getElementById('my-form');
38.       const cb = form.terms;
39.       cb.dataset.errorMsg = 'You must check the box!';
40.
41.       cb.addEventListener('click', function(e) {
42.           checkField(cb);
43.       });
44.
45.       form.addEventListener("submit", function(e) {
46.           // Check errors
47.           checkField(cb);
48.
49.           // If form is invalid, prevent submission
50.           if (!form.checkValidity()) {
51.               e.preventDefault();
52.               alert('Please fix form errors.');
```



```
53.           }
54.       });
55.   });
-----Lines 56 through 59 Omitted-----
60.   <form id="my-form" method="post" novalidate>
61.       <input type="checkbox" name="terms" id="terms" required>
62.       <label for="terms">
63.           I understand that it's really important
64.           that I check this box.
65.       </label>
-----Lines 66 through 70 Omitted-----
```

Code Explanation

1. This code should all be clear as it's similar to the text-field validation in the earlier examples.
2. Open FormValidation/Demos/checkbox.html and submit the form without checking the checkbox. You should get an alert saying "Please fix form errors." and see an error above the checkbox:



3. As you check and uncheck the box, the error should disappear and reappear.

20.9. Validating Radio Buttons


Radio buttons are similar to checkboxes; however, since our `checkField()` function only expects a single element, we have to pass just one radio button input element to the function. Luckily, if any one of a group of radio buttons is invalid, they are all invalid, so we can just send the first radio button, which we can get like this:

```
const answer = form.answer[0];
answer.dataset.errorMessage = 'Please answer the question.';
```

As with checkboxes, we will listen for `click` events. We loop through `form.answer`, which is an array of radio buttons, adding the click event to each one:

Demo 20.10: FormValidation/Demos/radio-buttons.html

```
-----Lines 1 through 35 Omitted-----
36.   window.addEventListener('load', function(e) {
37.       const form = document.getElementById('my-form');
38.       const answer = form.answer[0];
39.       answer.dataset.errorMsg = 'Please answer the question.';
40.
41.       // Add click event handler to each radio button
42.       for (let button of form.answer) {
43.           button.addEventListener('click', function(e) {
44.               checkField(answer);
45.           });
46.       }
47.
48.       form.addEventListener("submit", function(e) {
49.           // Check errors
50.           checkField(answer);
51.
52.           // If form is invalid, prevent submission
53.           if (!form.checkValidity()) {
54.               e.preventDefault();
55.               alert('Please fix form errors.');
```

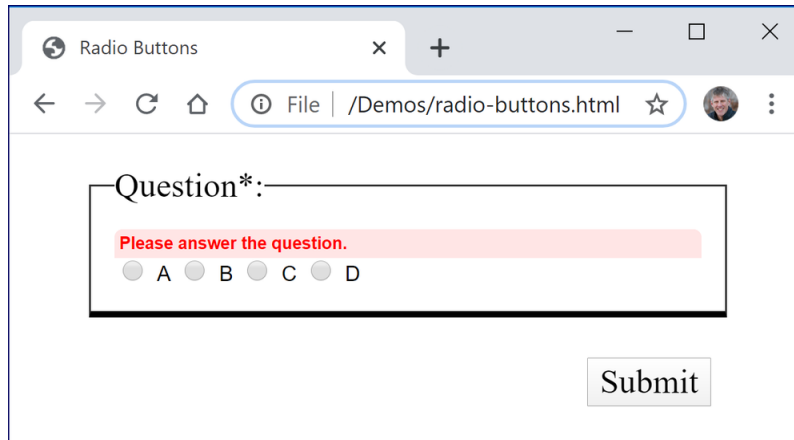


```
56.           }
57.       });
58.   });
59. </script>
60. <title>Radio Buttons</title>
61. </head>
62. <body>
63. <form id="my-form" method="post" novalidate>
64.     <fieldset>
65.         <legend>Question*:</legend>
66.         <label>
67.             <input type="radio" name="answer" value="1" required> A
68.         </label>
69.         <label>
70.             <input type="radio" name="answer" value="2" required> B
71.         </label>
72.         <label>
73.             <input type="radio" name="answer" value="3" required> C
74.         </label>
75.         <label>
76.             <input type="radio" name="answer" value="4" required> D
77.         </label>
78.     </fieldset>
```

-----Lines 79 through 83 Omitted-----

Code Explanation

1. This code should all be clear as it's similar to the text-field validation in the earlier examples.
2. Open `FormValidation/Demos/radio-buttons.html` and submit the form without selecting an answer. You should get an alert to fix form errors and see an error message:



3. As soon as you make a selection, the error should disappear.

❖ 20.9.1. Which Radio Button was Selected?

The following function is useful for checking which radio button in a group of radio buttons is selected. This might be useful in an upcoming challenge exercise!

```
function getSelectedRadio(radioArray) {  
  for (btn of radioArray) {  
    if (btn.checked) {  
      return btn;  
    }  
  }  
  return null;  
}
```

The function simply loops through the buttons until it finds one that is checked, at which point it returns that button. If it doesn't find any checked buttons, it returns null.



20.10. Validating Select Menus

When select menus allow for only one option to be selected (i.e., when the `multiple` attribute is not used), then there is always a default selection. If it is not set specifically using the `selected` attribute, then the first option (at index 0) will be selected by default. As such, the `required` attribute, while valid, doesn't actually change anything. It is a common practice to make the first option of `select` menu an invalid option that might be blank or read "--Please choose--"; however, there is no way to enforce this with HTML alone. Using JavaScript, we can report an error if the first option is left selected. In addition, because leaving the first option selected doesn't make the `select` menu invalid, we need to explicitly make the field invalid. This is done with the element's `setCustomValidity()` method, which takes a string with an error message:


```
function checkSelect(field) {  
  if ( field.selectedIndex === 0 ) {  
    field.setCustomValidity('Invalid');  
    addError(field);  
  } else {  
    field.setCustomValidity('');  
    removeError(field);  
  }  
}
```

By setting the custom validity to a string with at least one character, we make the field invalid, which in turn makes the whole form invalid. Then in the else block, we set the custom validity to an empty string, which indicates that the element has no custom validity error.

The following example shows how we incorporate this `checkSelect()` function in our form validation:

Demo 20.11: FormValidation/Demos/select-menus.html

```
-----Lines 1 through 27 Omitted-----
28.  function checkSelect(field) {
29.      if ( field.selectedIndex === 0 ) {
30.          field.setCustomValidity('Invalid');
31.          addError(field);
32.      } else {
33.          field.setCustomValidity('');
34.          removeError(field);
35.      }
36.  }
37.
38.  window.addEventListener('load', function(e) {
39.      const form = document.getElementById('my-form');
40.      const problem = form.problem;
41.      problem.dataset.errorMsg = 'Please choose an answer.';
42.
43.      problem.addEventListener('change', function(e) {
44.          checkSelect(problem);
45.      });
46.
47.      form.addEventListener("submit", function(e) {
48.          // Check errors
49.          checkSelect(problem);
50.
51.          // If form is invalid, prevent submission
52.          if (!form.checkValidity()) {
53.              e.preventDefault();
54.              alert('Please fix form errors.');
```

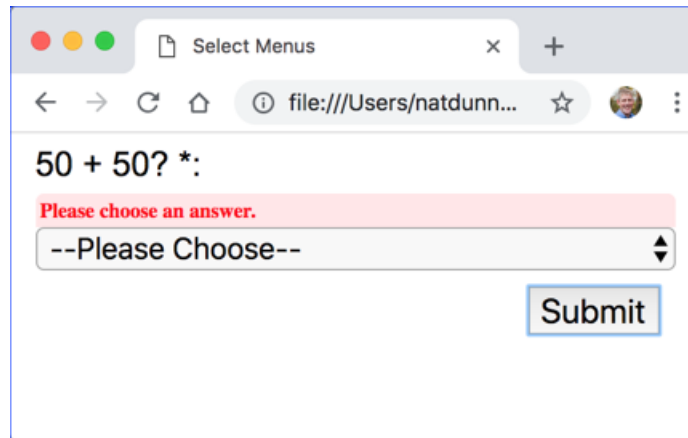


```
55.          }
56.      });
57.  });
58. </script>
59. <title>Select Menus</title>
60. </head>
61. <body>
62. <form id="my-form" method="post" novalidate>
63.     <label for="problem">50 + 50? *:</label>
64.     <select name="problem" id="problem" required>
65.         <option value="0">--Please Choose--</option>
66.         <option value="50">50</option>
67.         <option value="100">100</option>
68.         <option value="250">250</option>
69.     </select>
```


-----Lines 70 through 74 Omitted-----

Code Explanation

1. Note that the code listens for a `change` event on the `select` field.
2. Open `FormValidation/Demos/select-menus.html` and submit the form without selecting an answer. You should see an error:



3. As soon as you make a selection, the error should disappear. If you re-select the first option, the error should reappear.
-

Exercise 39: Validating the Ice Cream Order Form

⌚ 25 to 40 minutes

In this exercise, you will write JavaScript to validate the ice cream order form we saw at the beginning of this lesson.

1. Open `FormValidation/Exercises/ice-cream.html` for editing.
 - A. Add a script tag to use `ice-cream.js`.
 - B. Turn off HTML form validation.
2. Open `FormValidation/Exercises/ice-cream.js` for editing.
 - A. Notice that the following functions are already written:
 - i. `addError()`
 - ii. `removeError()`
 - iii. `checkField()`
 - iv. `checkSelect()`
3. Write code to add validation for the following fields:
 - A. `email`
 - B. `phone`
 - C. `username`
 - D. `container`
 - E. `flavor`
 - F. `requests`
 - G. `terms`
4. Each of the above fields should be validated when its value is changed. All fields should be validated when the form is submitted. Be sure to prevent the form from submitting if any field is invalid.
5. Test your solution in a browser.

The following screenshot shows the form (split into two parts) after submitting with errors in all fields:

<p>Username*:</p> <p>⚡ Username must be 8 to 25 characters.</p> <input type="text"/>	<p>Flavor*:</p> <p>⚡ Please select a flavor.</p> <p>--Please Choose--</p>
<p>Email*:</p> <p>⚡ Invalid Email</p> <input type="text"/>	<p>Toppings:</p> <p><input type="checkbox"/> Sprinkles <input type="checkbox"/> Nuts <input type="checkbox"/> Whipped Cream</p>
<p>Telephone:</p> <p>⚡ Invalid Phone. Use format: ###-###-####</p> <input type="text" value="5555555555"/>	<p>Special Requests:</p> <p>⚡ Your comment must be between 10 and 200 characters.</p> <p>Hello</p>
<p>Cup or Cone*:</p> <p>⚡ Please select a container.</p> <p><input type="radio"/> Cup <input type="radio"/> Plain cone <input type="radio"/> Sugar cone <input type="radio"/> Waffle Cone</p>	<p>⚡ You must accept the terms.</p> <p><input type="checkbox"/> I understand that I'm really not going to get any ice cream.</p>

Challenge

Write code to add an error if the user orders whipped cream on a cone:

Cup or Cone*:

☐ Cup ☒ Plain cone ☐ Sugar cone ☐ Waffle Cone

Flavor*:

--Please Choose--

Toppings:

☐ Sprinkles ☐ Nuts

⚡ You cannot have whipped cream on a cone.

☒ Whipped Cream

Note that you will want to perform this check:

1. When the user clicks the “Whipped Cream” checkbox.
2. When the user selects a container.
3. When the user submits the form.


Solution: FormValidation/Solutions/ice-cream.html

```
-----Lines 1 through 6 Omitted-----  
7.  <link rel="stylesheet" href="../styles.css">  
8.  <script src="ice-cream.js"></script>  
9.  <title>Ice Cream Order Form</title>  
10. </head>  
11. <body>  
12. <form id="ice-cream-form" method="post" novalidate>  
-----Lines 13 through 80 Omitted-----
```

Solution: FormValidation/Solutions/ice-cream.js

```
-----Lines 1 through 38 Omitted-----
39. window.addEventListener('load', function(e) {
40.     const form = document.getElementById('ice-cream-form');
41.     const email = form.email;
42.     email.dataset.errorMsg = 'Invalid Email';
43.     const phone = form.phone;
44.     phone.dataset.errorMsg = 'Invalid Phone. Use format: ###-###-####';
45.     const username = form.username;
46.     username.dataset.errorMsg = 'Username must be 8 to 25 characters.';
47.     const container = form.container[0];
48.     container.dataset.errorMsg = 'Please select a container.';
49.     const flavor = form.flavor;
50.     flavor.dataset.errorMsg = 'Please select a flavor.';
51.     const terms = form.terms;
52.     terms.dataset.errorMsg = 'You must accept the terms.';
53.     const requests = form.requests;
54.     requests.dataset.errorMsg = 'Your comment must be between ' +
55.         requests.minLength + ' and ' + requests.maxLength +
56.         ' characters.';
57.
58.     username.addEventListener("input", function(e) {
59.         checkField(username);
60.     });
61.
62.     email.addEventListener("input", function(e) {
63.         checkField(email);
64.     });
65.
66.     phone.addEventListener("input", function(e) {
67.         checkField(phone);
68.     });
69.
70.     for (radio of form.container) {
71.         radio.addEventListener("click", function(e) {
72.             checkField(container);
73.         });
74.     }
75.
76.     flavor.addEventListener("change", function(e) {
77.         checkSelect(flavor);
78.     });
79.
80.     requests.addEventListener("input", function(e) {
81.         checkField(requests);
```

```
82.     });
83.
84.     terms.addEventListener("click", function(e) {
85.         checkField(terms);
86.     });
87.
88.     form.addEventListener("submit", function(e) {
89.         // Check errors
90.         checkField(username);
91.         checkField(email);
92.         checkField(phone);
93.         checkField(container);
94.         checkSelect(flavor);
95.         checkField(requests);
96.         checkField(terms);
97.
98.         // If form is invalid, prevent submission
99.         if (!form.checkValidity()) {
100.             e.preventDefault();
101.             alert('Please fix form errors.');
```



```
102.         }
103.     });
104.
105. });
```

Challenge Solution: FormValidation/Solutions/ice-cream-challenge.js

```
-----Lines 1 through 38 Omitted-----
39. function getSelectedRadio(radioArray) {
40.   for (btn of radioArray) {
41.     if (btn.checked) {
42.       return btn;
43.     }
44.   }
45.   return null;
46. }
47.
48. function checkWhip(whip) {
49.   const radioArray = document.querySelectorAll('input[name="container"]');
50.   const selectedContainer = getSelectedRadio(radioArray);
51.   if (!selectedContainer) {
52.     return true; //container not selected
53.   }
54.   if (whip.checked && selectedContainer.value !== 'cup') {
55.     addError(whip);
56.     return false;
57.   } else {
58.     removeError(whip);
59.   }
60.   return true;
61. }
62.
63. window.addEventListener('load', function(e) {
64.   const form = document.getElementById('ice-cream-form');
65.   const email = form.email;
66.   email.dataset.errorMsg = 'Invalid Email';
67.   const phone = form.phone;
68.   phone.dataset.errorMsg = 'Invalid Phone. Use format: ###-###-####';
69.   const username = form.username;
70.   username.dataset.errorMsg = 'Username must be 8 to 25 characters.';
71.   const container = form.container[0];
72.   container.dataset.errorMsg = 'Please select a container.';
73.   const flavor = form.flavor;
74.   flavor.dataset.errorMsg = 'Please select a flavor.';
75.   const terms = form.terms;
76.   terms.dataset.errorMsg = 'You must accept the terms.';
77.   const requests = form.requests;
78.   requests.dataset.errorMsg = 'Your comment must be between ' +
79.     requests.minLength + ' and ' + requests.maxLength +
80.     ' characters.';
81.   const whip = form.whip;
```

Evaluation
Copy

```
82.     whip.dataset.errorMsg = 'You cannot have whipped cream on a cone.';
83.
84.     username.addEventListener("input", function(e) {
85.         checkField(username);
86.     });
87.
88.     email.addEventListener("input", function(e) {
89.         checkField(email);
90.     });
91.
92.     phone.addEventListener("input", function(e) {
93.         checkField(phone);
94.     });
95.
96.     for (radio of form.container) {
97.         radio.addEventListener("click", function(e) {
98.             checkField(container);
99.             checkWhip(whip);
100.        });
101.    }
102.
103.    flavor.addEventListener("change", function(e) {
104.        checkSelect(flavor);
105.    });
106.
107.    requests.addEventListener("input", function(e) {
108.        checkField(requests);
109.    });
110.
111.    terms.addEventListener("click", function(e) {
112.        checkField(terms);
113.    });
114.
115.    whip.addEventListener("click", function(e) {
116.        checkWhip(whip);
117.    });
118.
119.    form.addEventListener("submit", function(e) {
120.        // Check errors
121.        checkField(username);
122.        checkField(email);
123.        checkField(phone);
124.        checkField(container);
125.        checkSelect(flavor);
126.        checkField(requests);
```



```
127.     checkField(terms);
128.     const whipValid = checkWhip(whip);
129.
130.     // If form is invalid, prevent submission
131.     if (!form.checkValidity() || !whipValid) {
132.         e.preventDefault();
133.         alert('Please fix form errors.');
```



20.11. Giving the User a Chance

While it is great to capture errors early, so the user can fix them right away, some users might be a little put off by errors that show up before they have had a chance to finish an individual entry. For example, the way our ice cream form works now, as soon as the user starts entering an email address, an error shows up:

A screenshot of a web form with a label "Email*:" and a text input field. The input field contains the letter "a". Above the input field, there is a red error message that says "Invalid Email" with a small lightning bolt icon to its left. The entire form is enclosed in a blue border.

One solution would be to listen for change events, which do not fire until the field loses focus, instead of input events, which fire every time a character is added or removed. This solution works well the first time the user enters data, but it is not as nice when the user is correcting a field that already has an error as in the following scenario:

1. The user enters an invalid email address and moves on to the next field, causing an error to show up.
2. The user returns to the email field and fixes the email address. The error will not go away until the change event fires, which won't occur until the user leaves the field.

In the above scenario, it would be nicer to hide the error as soon as the field is valid and then show it again if the field becomes invalid again. We can do this by keeping track of which fields have been “touched.” We will consider a field to have been touched if the user changes its value or if the user has submitted the form and received validation errors.

Note that this will only apply to text-like input and textarea fields as we use change and click events to initiate validation on other field types. We can get all input and textarea fields using:

```
const inputFields = document.querySelectorAll('input, textarea');
```

Take a look at the following code:

Demo 20.12: FormValidation/Demos/ice-cream-final.js

```
-----Lines 1 through 81 Omitted-----
82.    // Get the input and textarea fields
83.    const inputFields = document.querySelectorAll('input, textarea');
84.
85.    // Loop through the input fields, marking them all "untouched"
86.    for (field of inputFields) {
87.        field.dataset.status = 'untouched';
88.    }
89.
90.    // When a user changes the value of a text-like input or textarea,
91.    // mark the field "touched"
92.    // validate the field
93.    username.addEventListener("change", function(e) {
94.        username.dataset.status = 'touched';
95.        checkField(username);
96.    });
97.
98.    // When a user inputs data into a text-like input or textarea
99.    // that has been touched, validate the field
100.    username.addEventListener("input", function(e) {
101.        if (username.dataset.status === 'touched') {
102.            checkField(username);
103.        }
104.    });
105.
106.    email.addEventListener("change", function(e) {
107.        email.dataset.status = 'touched';
108.        checkField(email);
109.    });
110.    email.addEventListener("input", function(e) {
111.        if (email.dataset.status === 'touched') {
112.            checkField(email);
113.        }
114.    });
115.
116.    phone.addEventListener("change", function(e) {
117.        phone.dataset.status = 'touched';
118.        checkField(phone);
119.    });
120.    phone.addEventListener("input", function(e) {
121.        if (phone.dataset.status === 'touched') {
122.            checkField(phone);
123.        }
124.    });
```

```

125.
126.   requests.addEventListener("change", function(e) {
127.       requests.dataset.status = 'touched';
128.       checkField(requests);
129.   });
130.   requests.addEventListener("input", function(e) {
131.       if (requests.dataset.status === 'touched') {
132.           checkField(requests);
133.       }
134.   });
    -----Lines 135 through 155 Omitted-----
156.       // Mark all fields touched
157.       for (field of inputFields) {
158.           field.dataset.status = 'touched';
159.       }
    -----Lines 160 through 177 Omitted-----

```

Code Explanation

The following code will not validate an input or textarea field on each input unless the user has previously changed that field or submitted the form. **Things to note:**

1. We save all the input and textarea fields as a NodeList (similar to an array of elements) called `inputFields`.
2. We loop through the input fields, setting the `dataset.status` property of each field to “untouched.”
3. We add change event listeners to each text-like input and the “requests” textarea. When the change event occurs, we set the `dataset.status` property of the field to “touched” and we validate the field.
4. We modified the input event listeners to check if the `dataset.status` property is “touched”, and if it is, we validate the field.
5. We added a for loop in the callback function of the submit event listener to set the `dataset.status` property of all the input and textarea fields to “touched”.

Open `FormValidation/Demos/ice-cream-final.html` in your browser to try it out.

Conclusion

In this lesson, you have learned to write clean, reusable form validation scripts.

Evaluation
Copy

LESSON 21

Regular Expressions

Topics Covered

- ☑ Regular expressions for advanced form validation.
- ☑ Regular expressions and backreferences to clean up form entries.

Introduction

Regular expressions are used to do sophisticated pattern matching, which can often be helpful in form validation. For example, a regular expression can be used to check whether an email address entered into a form field is syntactically correct. JavaScript supports Perl-compatible regular expressions.

21.1. Getting Started

There are two ways to create a regular expression in JavaScript:

1. Using literal syntax

```
const reExample = /pattern/;
```

2. Using the `RegExp()` constructor

```
const reExample = new RegExp("pattern");
```

There is no difference between the two.

For example, we could create a regular expression for a social security number like this:

```
// literal syntax
const reSSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;

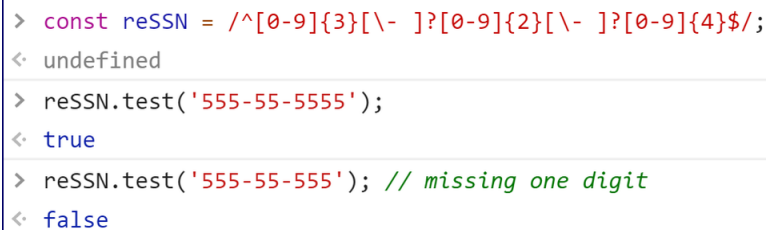
// RegExp() constructor syntax
const reSSN = new RegExp("^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$");
```

❖ 21.1.1. JavaScript's Regular Expression test() Method

The `test()` method takes one argument, a string, and checks whether that string contains a match of the pattern specified by the regular expression. It returns `true` if it does contain a match and `false` if it does not. This method is very useful in form validation scripts. The following code sample shows how it can be used for checking a social security number. Don't worry about the syntax of the regular expression itself. We'll cover that shortly.

```
const reSSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;
reSSN.test('555-55-5555');
reSSN.test('555-55-555'); // missing one digit
```

Here we show this code run in Chrome DevTools Console:



```
> const reSSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;
< undefined
> reSSN.test('555-55-5555');
< true
> reSSN.test('555-55-555'); // missing one digit
< false
```



21.2. Regular Expression Syntax

A regular expression is a pattern that specifies a list of characters. In this section, we will look at how those characters are specified.

Try These!

We encourage you to try these regular expressions in Google Chrome's console.

❖ 21.2.1. Start and End (^ \$)

A caret (^) at the beginning of a regular expression indicates that the string being searched must start with this pattern.

- The pattern ^foo can be found in “food”, but not in “barfood”.

```
> const pattern=/^foo/;
< undefined
> pattern.test('food');
< true
> pattern.test('barfood');
< false
```

A dollar sign (\$) at the end of a regular expression indicates that the string being searched must end with this pattern.

- The pattern foo\$ can be found in “curfoo”, but not in “food”.

```
> const pattern=/foo$/;
< undefined
> pattern.test('curfoo');
< true
> pattern.test('food');
< false
> |
```

❖ 21.2.2. Number of Occurrences (? + * {})

The following symbols affect the number of occurrences of the preceding character⁵⁹: ?, +, *, and {}.

A question mark (?) indicates that the preceding character should appear zero or one times in the pattern.

- The pattern foo? can be found in “food” and “fod”, but not “faod”.

59. Or characters if parentheses are used.

```

> const pattern=/foo?/;
< undefined
> pattern.test('food');
< true
> pattern.test('fod');
< true
> pattern.test('faod');
< false

```

A plus sign (+) indicates that the preceding character should appear one or more times in the pattern.

- The pattern `fo+` can be found in “fod”, “food” and “foood”, but not “fd”.

```

> const pattern=/fo+/;
< undefined
> pattern.test('fod');
< true
> pattern.test('food');
< true
> pattern.test('foood');
< true
> pattern.test('fd');
< false

```

An asterisk (*) indicates that the preceding character should appear zero or more times in the pattern.

- The pattern `fo*d` can be found in “fd”, “fod” and “food”.

```

> const pattern=/fo*d/;
< undefined
> pattern.test('fd');
< true
> pattern.test('fod');
< true
> pattern.test('food');
< true

```

Curly brackets with one parameter (`{n}`) indicate that the preceding character should appear exactly `n` times in the pattern.

- The pattern `fo{3}d` can be found in “food”, but not “food” or “foood”.

```
> const pattern=/fo{3}d/;
< undefined
> pattern.test('foood');
< true
> pattern.test('food');
< false
> pattern.test('foood');
< false
```

Curly brackets with two parameters (`{n1, n2}`) indicate that the preceding character should appear between `n1` and `n2` times in the pattern.

- The pattern `fo{2,4}d` can be found in “food”, “foood”, and “foood”, but not “fod” or “fooooood”.

```
> const pattern=/fo{2,4}d/;
< undefined
> pattern.test('foood');
< true
> pattern.test('foood');
< true
> pattern.test('foood');
< true
> pattern.test('fod');
< false
> pattern.test('fooooood');
< false
```

Curly brackets with one parameter and an empty second parameter (`{n, }`) indicate that the preceding character should appear at least `n` times in the pattern.

- The pattern `fo{2, }d` can be found in “food” and “fooooood”, but not “fod”.

```

> const pattern=/fo{2,}d/;
< undefined
> pattern.test('food');
< true
> pattern.test('fooooood');
< true
> pattern.test('fod');
< false

```

❖ 21.2.3. Common Characters (. \d \D \w \W \s \S)

A period (.) represents any character except a newline.

- The pattern `fo.d` can be found in “food”, “foad”, “fo9d”, and “fo*d”.

```

> const pattern=/fo.d/;
< undefined
> pattern.test('food');
< true
> pattern.test('foad');
< true
> pattern.test('fo9d');
< true
> pattern.test('fo*d');
< true

```

Backslash-d (\d) represents any digit. It is the equivalent of [0-9] (see page 537).

- The pattern `fo\dd` can be found in “fo1d”, “fo4d” and “fo0d”, but not in “food”.

```

> const pattern=/fo\dd/;
< undefined
> pattern.test('fo1d');
< true
> pattern.test('fo4d');
< true
> pattern.test('fo0d');
< true
> pattern.test('food');
< false

```

Backslash-D (\D) represents any character **except a digit**. It is the equivalent of [^0-9] (see page 538).

- The pattern fo\Dd can be found in “food” and “fo_d”, but not in “fo4d”.

```

> const pattern=/fo\Dd/;
< undefined
> pattern.test('food');
< true
> pattern.test('fo_d');
< true
> pattern.test('fo4d');
< false

```

Backslash-w (\w) represents any word character (letters, digits, and the underscore (_)).

- The pattern fo\wd can be found in “food”, “fo_d” and “fo4d”, but not in “fo*d”.

```

> const pattern=/fo\wd/;
< undefined
> pattern.test('food');
< true
> pattern.test('fo_d');
< true
> pattern.test('fo4d');
< true
> pattern.test('fo*d');
< false

```

Backslash-W (\W) represents any character **except a word character**.

- The pattern `fo\Wd` can be found in “fo*d”, “fo@d” and “fo.d”, but not in “food”.

```
> const pattern=/fo\Wd/;
< undefined
> pattern.test('fo*d');
< true
> pattern.test('fo@d');
< true
> pattern.test('fo.d');
< true
> pattern.test('food');
< false
```

Backslash-s (\s) represents any whitespace character (e.g. space, tab, newline, etc.).

- The pattern `fo\s d` can be found in “fo d”, but not in “food”.

```
> const pattern=/fo\s d/;
< undefined
> pattern.test('fo d');
< true
> pattern.test('food');
< false
```

Backslash-S (\S) represents any character except a whitespace character.

- The pattern `fo\Sd` can be found in “fo*d”, “food” and “fo4d”, but it cannot be found in “fo d”.

```

> const pattern=/fo\Sd/;
< undefined
> pattern.test('fo*d');
< true
> pattern.test('food');
< true
> pattern.test('fo4d');
< true
> pattern.test('fo d');
< false

```

❖ 21.2.4. Grouping ([])

Square brackets ([]) are used to group options.

- [aeiou] matches an “a”, an “e”, an “i”, an “o”, or a “u”.
 - The pattern f[aeiou]d can be found in “fad” and “fed”, but not in “fyd”, “food” or “fd”.

```

> const pattern=/f[aeiou]d/;
< undefined
> pattern.test('fad');
< true
> pattern.test('fed');
< true
> pattern.test('fyd');
< false
> pattern.test('food');
< false
> pattern.test('fd');
< false

```

- Number of occurrence characters can be used with groups.
 - The pattern f[aeiou]{2}d can be found in “food”, “faed” and “feod”, but not in “faeod”, “fed” or “fd”.

```

> const pattern=/f[aeiou]{2}d/;
< undefined
> pattern.test('food');
< true
> pattern.test('faed');
< true
> pattern.test('feod');
< true
> pattern.test('faeod');
< false
> pattern.test('fed');
< false
> pattern.test('fd');
< false

```

- Ranges can be created using a dash:
 - [A-Z] matches any upper case letter.
 - [a-z] matches any lower case letter.
 - [A-Za-z] matches any letter, regardless if it is lower case or upper case.

❖ 21.2.5. Negation (^)

When used as the first character within square brackets, the caret (^) is used for negation.

- The pattern `f[^aeiou]d` can be found in “fqd” and “f4d”, but not in “fad” or “fed”.

```

> const pattern=/f[^aeiou]d/;
< undefined
> pattern.test('fqd');
< true
> pattern.test('f4d');
< true
> pattern.test('fad');
< false
> pattern.test('fed');
< false

```


❖ 21.2.6. Subpatterns ()

Parentheses () are used to capture subpatterns.

- The pattern `f(oo)?d` indicates that the subpattern `oo` can show up zero or one time. The pattern can be found in “food” and “fd”, but not in “fod”.

```
> const pattern=/f(oo)?d/;
< undefined
> pattern.test('food');
< true
> pattern.test('fd');
< true
> pattern.test('fod');
< false
```

❖ 21.2.7. Alternatives (|)

The pipe (|) is used to create optional patterns.

- The pattern `foo$|^bar` can be found in “foo” and “bar”, but not “foobar”.

```
> const pattern=/foo$|^bar/;
< undefined
> pattern.test('foo');
< true
> pattern.test('bar');
< true
> pattern.test('foobar');
< false
```

❖ 21.2.8. Escape Character (\)

The backslash (\) is used to escape special characters, such as periods (.), dashes (-), forward slashes (/) and backslashes (\).

- The pattern `fo\d` can be found in “fo.d”, but not in “food” or “fo4d”.

```

> const pattern=/fo\d/;
< undefined
> pattern.test('fo.d');
< true
> pattern.test('food');
< false
> pattern.test('fo4d');
< false

```

❖ 21.2.9. Case-Insensitive Matches

The `i` flag makes a whole pattern match case insensitive.

- **Literal Syntax:** Flags are added after the end slash. For example, `/aeiou/i` matches all lowercase and uppercase vowels.
- **RegExp() Constructor Syntax:** Flags are added as the second argument. For example, `RegExp('abcde', 'i')` matches all lowercase and uppercase vowels.

The pattern `/f[aeiou]d/i` or `RegExp('f[aeiou]d', 'i')` can be found in “fad”, “FAD”, “FaD”, and “fAd”.

```

> const pattern=RegExp('f[aeiou]d','i');
< undefined
> pattern; // Notice it outputs literal syntax
< /f[aeiou]d/i
> pattern.test('fad');
< true
> pattern.test('FAD');
< true
> pattern.test('FaD');
< true
> pattern.test('fAd');
< true

```



21.3. Backreferences

Backreferences are special wildcards that refer back to a subpattern within a pattern. They can be used to make sure that two subpatterns match. The first subpattern in a pattern is referenced as `\1`, the second is referenced as `\2`, and so on.

For example, the pattern `/^([bmpw])o\1$/` matches “bob”, “mom”, “pop”, and “wow”, but not “bop” or “pow”.

```
> const pattern=/^([bmpw])o\1$/;
< undefined
> pattern.test('bob');
< true
> pattern.test('mom');
< true
> pattern.test('pop');
< true
> pattern.test('wow');
< true
> pattern.test('bop');
< false
> pattern.test('pow');
< false
```

A more practical example has to do with matching the delimiter in social security numbers, which are 9 digits long and separated into three parts: three digits, then two digits, then four digits. Examine the following regular expression:

```
/^\d{3}([\ - ]?)\d{2}([\ - ]?)\d{4}$/
```

Within the caret (^) and dollar sign (\$), which are used to specify the beginning and end of the pattern, there are three sequences of digits, optionally separated by a hyphen or a space. Note that the hyphen needs to be escaped with a backslash because dashes are special characters in regular expressions. This pattern will be matched in all of the following strings (and more):

1. 123-45-6789
2. 123 45 6789
3. 123456789
4. 123-45 6789

5. 123 45-6789
6. 123-456789

The last three strings are not ideal, but they do match the pattern.

```
> const reSSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;
< undefined
> reSSN.test('123-45-6789');
< true
> reSSN.test('123 45 6789');
< true
> reSSN.test('123456789');
< true
> reSSN.test('123-45 6789');
< true
> reSSN.test('123 45-6789');
< true
> reSSN.test('123-456789');
< true
```

Backreferences can be used to make sure that the second delimiter matches the first delimiter. The regular expression would look like this:

```
/^\d{3}([\- ]?)\d{2}\1\d{4}$/
```

The `\1` refers back to the first subpattern. Only the first three strings listed above match this regular expression.

```

> const reSSN = /^d{3}([\ - ]?)d{2}\1d{4}$/;
< undefined
> reSSN.test('123-45-6789');
< true
> reSSN.test('123 45 6789');
< true
> reSSN.test('123456789');
< true
> reSSN.test('123-45 6789');
< false
> reSSN.test('123 45-6789');
< false
> reSSN.test('123-456789');
< false

```



21.4. Form Validation with Regular Expressions

Regular expressions are often used to create stronger form validation. This can often be done using the `pattern` attribute in form fields. Consider the HTML `email` type. While it does force the user to enter a syntactically valid email address, it is not very restrictive. For example, the following would be considered valid “a@a” and would pass as a valid email address. You may wish to have your validation be a little stricter than that. For example, you might require the following sequence. The regular expression matching each sequence part is shown in parentheses:

1. One or more of any of the following characters: letters, numbers, underscores, dashes, and periods.
 - Matching regular expression: `([\w\-\.\])+`
2. An @ sign.
 - Matching regular expression: `@`
3. One or more of any of the following characters: letters, numbers, underscores, dashes, and periods.
 - Matching regular expression: `([\w\-\.\])+`
4. A period followed by two to four letters.
 - Matching regular expression: `\.([A-Za-z]{2,4})`

The full regular expression looks like this:

```
([\w\-\.\.])+@([\w\-\.\.]+\.[A-Za-z]{2,4})
```

Email Regular Expression

Note that the regular expression shown above prevents some syntactically valid email addresses and allows for some invalid ones. Creating a robust pattern for an email address is quite complex. The site <https://emailregex.com> offers the following regular expression, which it claims works 99.99% of the time:

```
/^((([^\<>()\[\]\.\.,;:\s@" ]+(\.[^\<>()\[\]\.\.,;:\s@" ]+)*)|(" .+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.([a-zA-Z-0-9]+\.)+[a-zA-Z]{2,}))$|
```

As mentioned earlier, you can use regular expressions to validate email addresses, zip codes, usernames, passwords, or any number of other fields simply by using the `pattern` attribute in your HTML input fields, like this:

```
<input type="email"
pattern="([\w\-\.\.])+@([\w\-\.\.]+\.[A-Za-z]{2,4})">
```



21.5. Cleaning Up Form Entries

Regular expressions can also be used to clean up user entries immediately after they are entered. This can be done using a combination of regular expressions and the `replace()` method of a string object.

The `replace()` Method

The `replace()` method of a string takes two arguments: a regular expression and a replacement string. It replaces the first regular expression match in the string with the replacement string. If the `g` flag (for global) is used in the regular expression, it replaces all matches with the string.

```
"Webucator".replace(/cat/, "dog"); //returns Webudogor
"Webucator".replace(/[aeiou]/g, "x"); //returns Wxbxcxtxr
```

And here it is in the Google Chrome console:

```
> "Webucator".replace(/cat/, "dog");  
< "Webudogor"  
  
> "Webucator".replace(/[aeiou]/g, "x");  
< "Wxbxcxtxr"
```

With the `replace()` method, it is also possible to replace a matched pattern with a new string made up of submatches from the pattern. The following example illustrates this.

Demo 21.1: RegularExpressions/Demos/ssn-cleaner.html

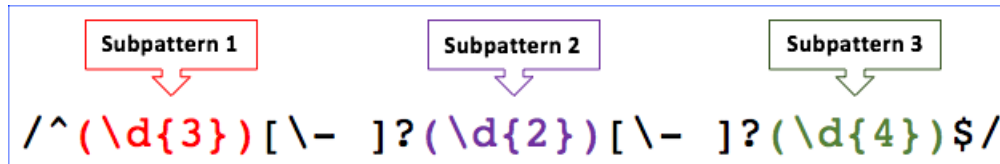
```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.  <meta charset="UTF-8">
5.  <meta name="viewport" content="width=device-width,initial-scale=1">
6.  <link rel="stylesheet" href="../normalize.css">
7.  <link rel="stylesheet" href="../styles.css">
8.  <title>SSN Cleaner</title>
9.  <script>
10. function cleanSSN(ssn) {
11.     const reSSN = /^(\\d{3})[\\- ]?(\\d{2})[\\- ]?(\\d{4})$/;
12.     if (reSSN.test(ssn)) {
13.         const cleanedSsn = ssn.replace(reSSN, "$1-$2-$3");
14.         return cleanedSsn;
15.     } else {
16.         alert("INVALID SSN");
17.         return ssn;
18.     }
19. }
20.
21. window.addEventListener('load', function(e) {
22.     const btn = document.getElementById('clean');
23.     const ssn = document.getElementById('ssn');
24.     btn.addEventListener('click', function(e) {
25.         ssn.value = cleanSSN(ssn.value);
26.         e.preventDefault(); // prevent form submission
27.     });
28. });
29. </script>
30. </head>
31. <body>
32.     <form>
33.         <input id="ssn" name="ssn" size="20">
34.         <button id="clean">Clean SSN</button>
35.     </form>
36. </body>
37. </html>
```

Code Explanation

The `cleanSSN()` function is used to “clean up” a social security number. The regular expression contained in `reSSN` contains three subexpressions, denoted with parentheses:


```
^(\d{3})[\- ]?(\d{2})[\- ]?(\d{4})$
```

1. `(\d{3})`
2. `(\d{2})`
3. `(\d{4})`



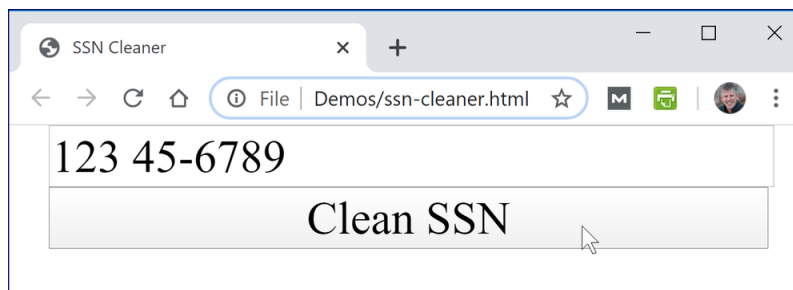
Within the second argument of the `replace()` method, these subexpressions can be referenced as `$1`, `$2`, and `$3`, respectively.

When the user clicks the “Clean SSN” button, the `cleanSSN()` function is called. This function first uses a regular expression to test that the user-entered value is a valid social security number. If it is, it then cleans it up with the following line of code, which dash-delimits the three substrings matching the subexpressions.

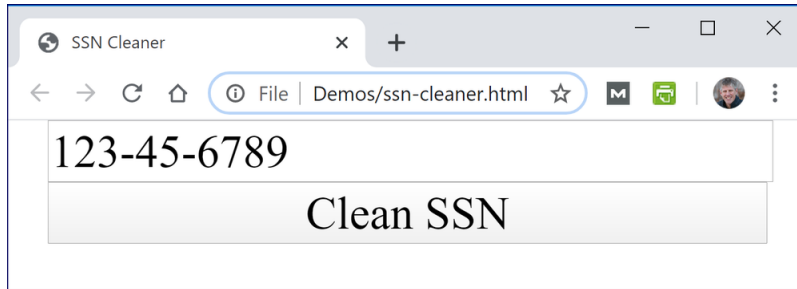
```
const cleanedSsn = ssn.replace(reSSN, "$1-$2-$3");
```

It then returns the cleaned-up social security number.

For example, in the following form, the social security number is entered with one space and one dash:



After clicking the **Clean SSN** button, the social security number gets formatted with two dashes:





Exercise 40: Cleaning Up Form Entries

⌚ 15 to 25 minutes

In this exercise, you will create a function for validating and formatting a phone number. The phone number should have the following format:

1. An optional open parentheses.
2. Three digits.
3. An optional close parentheses.
4. An optional dash, dot or space.
5. Three digits.
6. An optional dash, dot or space.
7. Four digits.

Here are some examples of valid phone numbers:

1. 5551234567
2. (555) 123 4567
3. 555-123-4567
4. 555 123 4567

Evaluation
Copy

You are to do the following:

1. Open `RegularExpressions/Exercises/phone-cleaner.html` for editing.
2. Add a `pattern` attribute to the phone field with a regular expression matching the pattern described above.
3. Write a `cleanPhone()` function that takes one argument: the field containing a phone number. It should check to see that the field contains a valid phone number.
 - A. If it does, it should return a cleaned-up version of the phone number in the format: `(555) 123-4567`.
 - B. If it does not, it should alert "INVALID PHONE" and return the current unchanged invalid phone value.
4. Test your solution in a browser.

Solution: RegularExpressions/Solutions/phone-cleaner.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>Phone Cleaner</title>
9.  <script>
10. function cleanPhone(phone) {
11.     const rePhone = RegExp(phone.pattern);
12.     const phoneNum = phone.value;
13.     const cleanedPhone = phoneNum.replace(rePhone, "($1) $2-$3");
14.     if (rePhone.test(phoneNum)) {
15.         return cleanedPhone;
16.     } else {
17.         alert("INVALID PHONE");
18.         return phoneNum;
19.     }
20. }
21.
22. window.addEventListener('load', function(e) {
23.     const btn = document.getElementById('clean');
24.     const phone = document.getElementById('phone');
25.     btn.addEventListener('click', function(e) {
26.         phone.value = cleanPhone(phone);
27.         e.preventDefault(); // prevent form submission
28.     });
29. });
30. </script>
31. </head>
32. <body>
33.     <form>
34.         <input type="tel" id="phone" name="phone" size="20"
35.             pattern="^\(?(\d{3})\)?[\-\.\ ]?(\d{3})[\-\.\ ]?(\d{4})$" />
36.         <button id="clean">Clean Phone</button>
37.     </form>
38. </body>
39. </html>
```



21.6. A Slightly More Complex Example

Some phone numbers are given as a combination of numbers and letters (e.g, 877-WEBUCATE). As is the case with 877-WEBUCATE, such numbers often have an extra character just to make the word complete.

In the following demo, we will:

1. Add a function called `convertPhone()` that:
 - Strips all characters that are not numbers or letters.
 - Converts all letters to numbers using the following rules:
 - ABC -> 2
 - DEF -> 3
 - GHI -> 4
 - JKL -> 5
 - MNO -> 6
 - PQRS -> 7
 - TUV -> 8
 - WXYZ -> 9

Notice that the regular expressions used in the conversions include both the `g` flag (for **g**lobal) so that all matches get replaced, and the `i` flag, so that the search is case-insensitive.

- Passes the first 10 characters of the resulting string, and the pattern to use to the `cleanPhone()` function.
 - Returns the resulting string.
2. Modify the form, so that it calls `convertPhone()` rather than `cleanPhone()`.

Demo 21.2: RegularExpressions/Demos/phone-converter.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>Phone Converter</title>
9.  <script>
10. function cleanPhone(phone, pattern) {
11.     const cleanedPhone = phone.replace(pattern, "($1) $2-$3");
12.     return cleanedPhone;
13. }
14.
15. function convertPhone(phone) {
16.     const rePhone = /^\\(?(\\d{3})\\)?[\\-\\. ]?([A-Za-z\\d]{3})[\\-\\. ]?([A-Za-z\\d]{4})/;
17.     if (!rePhone.test(phone.value)) {
18.         alert("INVALID PHONE");
19.         return phone.value;
20.     }
21.     let convertedPhone = phone.value.replace(/[A-Za-z\\d]/g, "");
22.     convertedPhone = convertedPhone.replace(/[ABC]/gi, "2");
23.     convertedPhone = convertedPhone.replace(/[DEF]/gi, "3");
24.     convertedPhone = convertedPhone.replace(/[GHI]/gi, "4");
25.     convertedPhone = convertedPhone.replace(/[JKL]/gi, "5");
26.     convertedPhone = convertedPhone.replace(/[MNO]/gi, "6");
27.     convertedPhone = convertedPhone.replace(/[PQRS]/gi, "7");
28.     convertedPhone = convertedPhone.replace(/[TUV]/gi, "8");
29.     convertedPhone = convertedPhone.replace(/[WXYZ]/gi, "9");
30.     phone.value = convertedPhone.substring(0, 10);
31.
32.     return cleanPhone(phone.value, rePhone);
33. }
34.
35. window.addEventListener('load', function(e) {
36.     const btn = document.getElementById('clean');
37.     const phone = document.getElementById('phone');
38.     btn.addEventListener('click', function(e) {
39.         phone.value = convertPhone(phone);
40.         e.preventDefault(); // prevent form submission
41.     });
42. });
43. </script>
44. </head>
```

```
45. <body>
46.   <form novalidate>
47.     <input type="tel" id="phone" name="phone" size="20">
48.     <button id="clean">Clean Phone</button>
49.   </form>
50. </body>
51. </html>
```

Evaluation
Copy

Conclusion

In this lesson, you have learned to work with regular expressions to validate and to clean up form entries.