7.1 Using JavaScript with HTML

Basics

JavaScript allows webpages to be more interactive. JavaScript can execute in response to user interactions and alter the contents of the webpage. Ex: A user clicks on a button, and JavaScript executes and changes the color of the webpage.

The ***Document Object Model*** (or ***DOM***) is a data structure that represents all parts of an HTML document. The JavaScript object ***document*** represents the entire DOM and is created from the document's HTML. Changes made to `document` are reflected in the browser presentation and/or behavior.

Webpages add JavaScript code by using the `<script>` tag. JavaScript code between `<script></script>` tags is executed by the browser's JavaScript engine.

PARTICIPATION ACTIVITY 7.1.1: Writing JavaScript within the body of an HTML file.

The JavaScript code below uses the ***document.writeln()*** method, which outputs HTML into the document and alters the DOM.

1. Read the HTML and JavaScript below.
2. Render the webpage to run the JavaScript code that displays a randomly generated response.
3. Add more responses to the `responses` array, and render the webpage a few times until one of your new responses is displayed.

```
<!DOCTYPE html>
<html>
<head>
   <title>Magic 8 Ball</title>
   <meta charset="UTF-8">
</head>
<body>
   <h1>Magic 8 Ball</h1>
   <script>

   // Possible 8 Ball responses
   let responses = [ "Without a doubt", "Ask again later", "Don't cou

   // Display a randomly chosen response
   let randomNum = Math.floor(Math.random() * responses.length);
   document.writeln("<p>Magic 8 Ball says... <strong>" + responses[ra
```

Render webpage Reset code

Your webpage

Magic 8 Ball

Magic 8 Ball says... **Don't count on it**.

▶ View solution

PARTICIPATION ACTIVITY 7.1.2: JavaScript Basics.

1) The DOM is created from a document's HTML.

   - True
   - False

2) The DOM is accessible via the global object named `document`.

   - True
   - False

3) `document.writeln("<div>test</div>")` adds a `div` element to the DOM.

   - True
   - False

Window object

JavaScript running in a web browser has access to the ***window*** object, which represents an open browser window. In a tabbed browser, each tab has a `window` object. The `document` object is a property of the `window` object and can be accessed as `window.document` or just `document`. Other properties of the `window` object include:

- ***window.location*** is a location object that contains information about the window's current URL. Ex: `window.location.hostname` is the URL's hostname.
- ***window.navigator*** is a navigator object that contains information about the browser. Ex: `window.navigator.userAgent` returns the browser's user agent string.
- ***window.innerHeight*** and ***window.innerWidth*** are the height and width in pixels of the window's content area. Ex: `window.innerWidth` returns 600 if the browser's content area is 600 pixels wide.

The `window` object defines some useful methods:

- ***window.alert()*** displays an alert dialog box. Ex: `window.alert("Hello")` displays a dialog box with the message "Hello".
- ***window.confirm()*** displays a confirmation dialog box with OK and Cancel buttons.

`confirm()` returns true if OK is pressed and false if Cancel is pressed. Ex: `window.confirm("Are you sure?")` displays a dialog box with the question.

- ***window.open()*** opens a new browser window. Ex: `window.open("http://www.twitter.com/")` opens a new browser that loads the Twitter webpage.

PARTICIPATION ACTIVITY 7.1.3: Using the window object.

Use the `window.confirm()` method to ask if the user would like to see a popup window:

```
let okPressed = window.confirm("Would you like to see a popup window?");
```

Then render the webpage, and click the OK button when prompted to see a small browser window created by `window.open()`. You may need to give your browser permission to show the popup window since many browsers prevent popups from displaying by default.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4     <title>JavaScript Demo</title>
5     <meta charset="UTF-8">
6  </head>
7  <body>
8     <h1>Popup Demo</h1>
9     <script>
10
11    let okPressed = false;
12    if (okPressed) {
13       let myWindow = window.open("", "", "width=250, height=100");
14       myWindow.document.writeln("<h1>Hello, Popup!</h1>");
15    }
16
```

Render webpage Reset code

Your webpage

Popup Demo

▶ View solution

PARTICIPATION ACTIVITY 7.1.4: Window object.

1) Can window object properties and methods be accessed without putting `window.` in front of the property or method?

   - ○ Yes
   - ○ No

2) What window object property is useful for determining if the webpage is loaded with HTTPS or HTTP?

   - ○ location
   - ○ navigator
   - ○ innerHeight

3) What window object property likely produces the following output?

```
document.writeln(window.navigator._____);
```

```
Mozilla/5.0 (Windows NT 10.0; WOW64)
AppleWebKit/537.36 Chrome/53.0.2785.116
```

- language
- userAgent
- vendor

4) The _____ window method is ideal for displaying a pop-up advertisement.

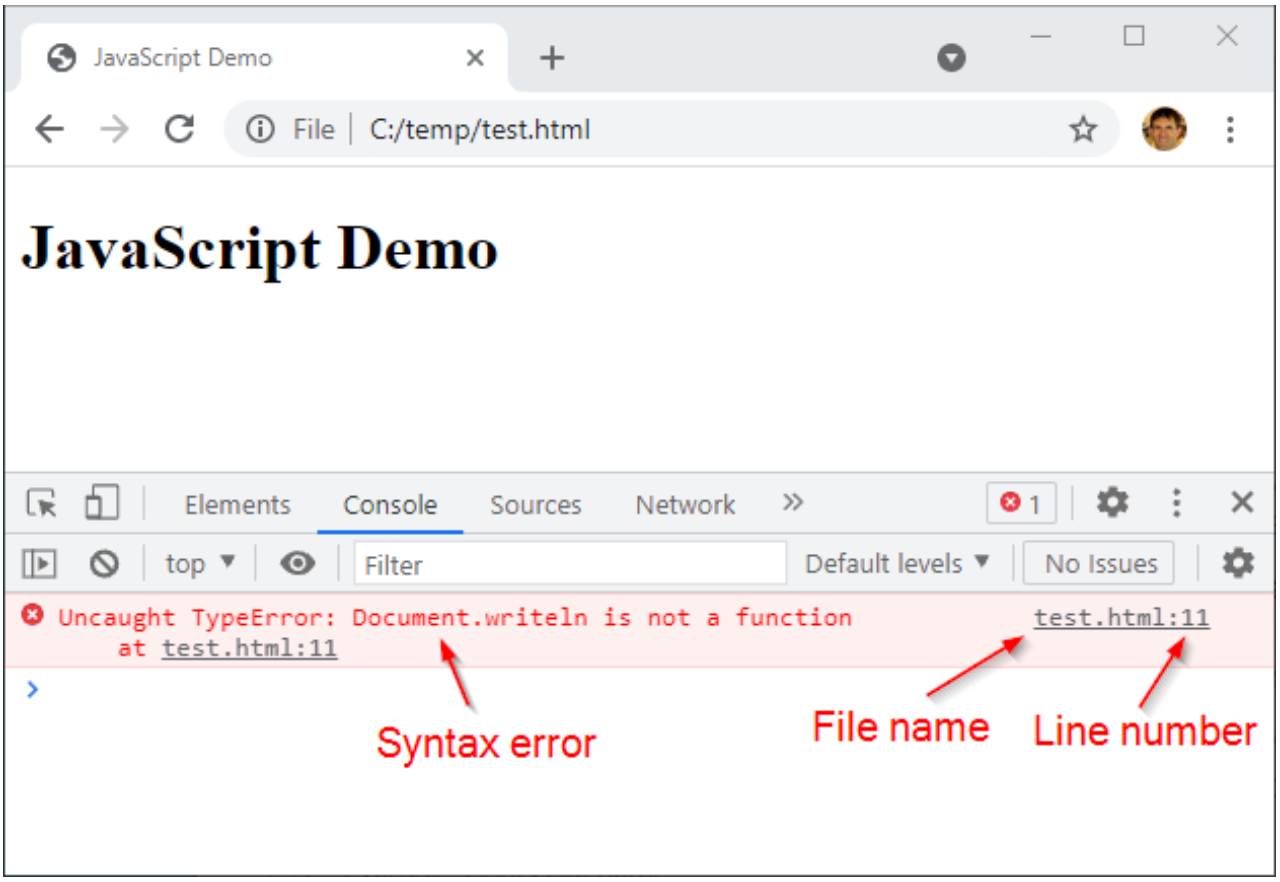
- alert()
- confirm()
- open()

Using the console

Modern browsers provide a ***console*** that allows the JavaScript code to produce informational and debugging output for the web developer, which does not affect the functionality or presentation of the webpage. By default, the console is not visible. The console is viewable in Chrome by pressing Ctrl+Shift+J in Windows/Linux or Cmd+Opt+J on a Mac.

When a syntax error is present in JavaScript code or a run-time error occurs, the error is only made visible in the console. The figure below shows the syntax error created when the developer accidentally typed `Document.writeln()` with a capital "D". The console appears underneath the webpage. *Good practice is to leave the console open while writing and testing JavaScript code.*

Figure 7.1.1: Chrome console showing a syntax error on line 11 of test.html.




```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>JavaScript Demo</title>
</head>
<body>
   <h1>JavaScript Demo</h1>
   <script>

   Document.writeln("<p>Hello, JavaScript!</p>");

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

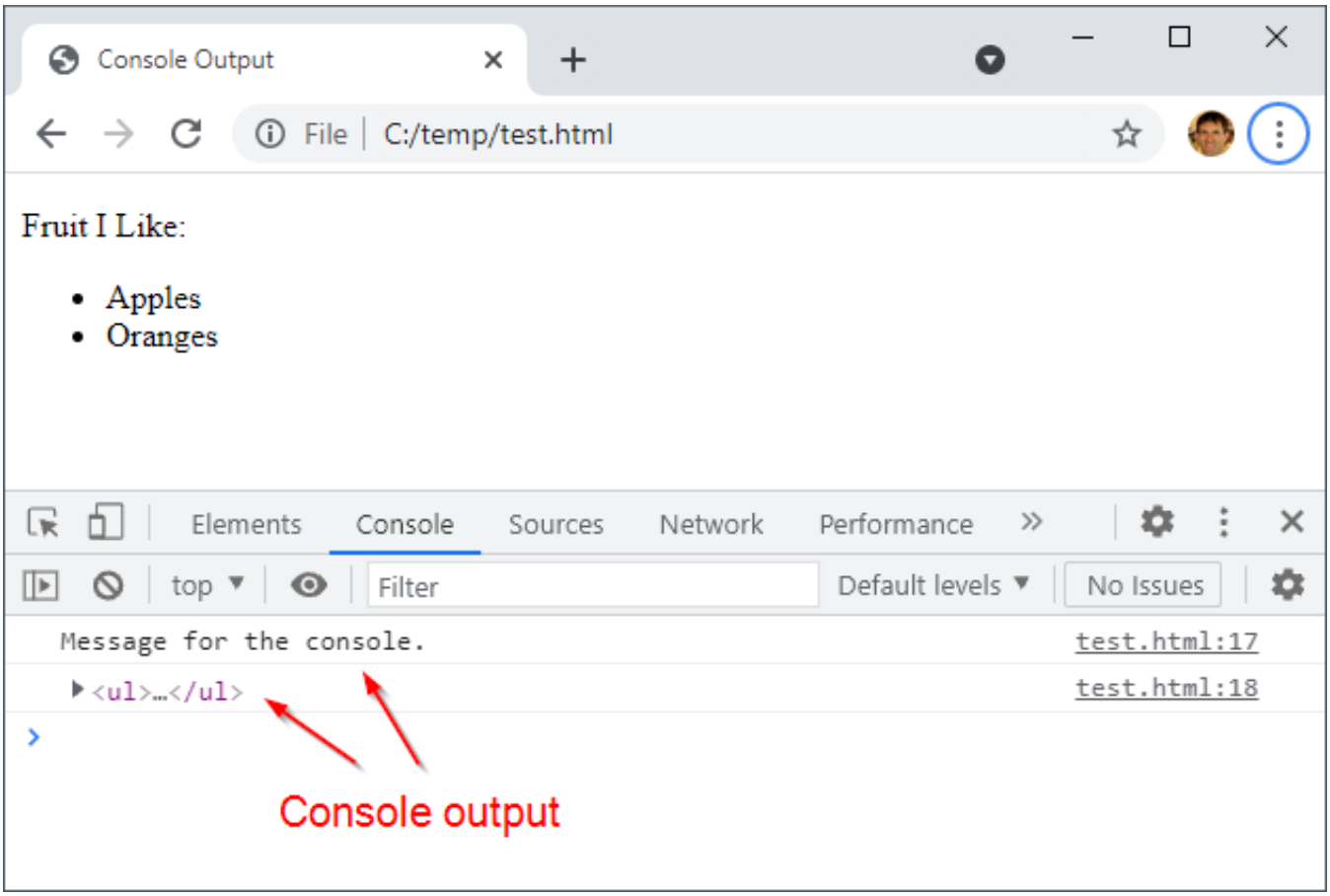
The browser provides a `console` object with a defined set of methods, or API, that the `console` object supports. An ***API*** (***Application Programming Interface***) is a specification of the methods and

objects that defines how a programmer should interact with software components. The console API includes the following methods:

- ***console.log()*** displays informational data to the console.
- ***console.warn()*** displays warnings to the console. The browser usually has a special indicator to differentiate a warning from the standard log message. Ex: A yellow warning box.
- ***console.error()*** displays errors to the console. The browser usually has a special indicator to differentiate an error from a warning or the standard log message. Ex: A red error box.
- ***console.dir()*** displays a JavaScript object to the console. The browser usually supports a method for compactly representing the object. Ex: a hierarchical tree representation allowing a developer to expand and collapse the object contents.

Figure 7.1.2: console.log() output example.

When the web browser console is open, both the webpage and the console are simultaneously visible.



console.log() can print both strings and concise representations of HTML elements.

```
<body>
   <p>
      Fruit I Like:
   </p>
   <ul>
      <li>Apples</li>
      <li>Oranges</li>
   </ul>

   <script>
console.log("Message for the console.");
console.log(document.getElementsByTagName("ul")[0]);
   </script>
</body>
```

PARTICIPATION ACTIVITY 7.1.5: Console methods.

Match the console method with the best use for that method.

If unable to drag and drop, refresh the page.

warn() dir() error() log()

	Helping determine why code isn't working as expected.
	Displaying a structured JavaScript object.
	Checking that assumptions in code are correct.
	Reporting unexpected problems.

Reset

Loading JavaScript from an external file

Including JavaScript directly within an HTML file is common practice when using small amounts of JavaScript. However, writing JavaScript directly within the document may lead to problems as a webpage or website gets larger.

Good practice is to use ***<script>*** *tags to load JavaScript from an external file rather than writing the JavaScript directly within the HTML file.* The **<script>** tag's **src** attribute specifies a JavaScript file to load.

Example 7.1.1: Loading JavaScript from an external file.

```
<script src="bootstrap.js">
</script>
```

A common error when loading an external JavaScript file is to forget the closing ***</script>*** *tag or trying to use a self-closing* ***<script />*** *tag as in* ***<script src="bootstrap.js" />****. All modern browsers require a closing* ***</script>*** *tag.*

PARTICIPATION ACTIVITY 7.1.6: Loading an external JavaScript file.

index.html

```
<!DOCTYPE html>
<html>
  <title>JavaScript Example</title>
  <script src="file.js"></script>
  <body>
    <p>A piece of text.</p>
    <img src="dog.jpg" alt="dog">
    <p>Some more text.</p>
  </body>
</html>
```

file.js

```
alert("Press enter to continue.");
```

Web browser

A piece of text.

Some more text.

Web server

Animation content:

The following code snippets are displayed:
Index.html:

```
<!DOCTYPE html>
<html>
 <title>JavaScript Example</title>
 <script src="file.js"></script>
 <body>
  <p>A piece of text.</p>
  <img src="dog.jpg" alt="dog">
  <p>Some more text.</p>
 </body>
</html>
```

file.js:

```
alert("Press enter to continue.");
```

A web browser and a web server are also shown.

Step 1:
index.html code snippet is displayed coming from the web server when the web browser requests the code.

Step 2:
The code line " <script src="file.js"></script>" is highlighted, which indicates to the browser that the browser should load JavaScript from an external file.

Step 3:
The web browser requests file.js from the web server.

Step 4:
The code in file.js "alert("Press enter to continue.");" is highlighted, and a box saying "Please enter to continue" shows up on the web browser.

Step 5:
After the user presses enter, the web browser finishes reading the javascript file and continues in the HTML file. The line " A piece of text. " is highlighted, and the line "A piece of text." appears on the web browser window.

Step 6:
The line " <img src="dog.jpg" alt="dog">" is highlighted, and a request is shown being made to the web server for the "dog.jpg" file. The image is then shown on the web browser under the "A piece of text" line.

Step 7:
The web browser finishes reading the HTML file, highlighting the line " Some more text. ", and showing "Some more text." on the web browser, under the dog image.

Animation captions:

1. The web server sends index.html to the web browser.
2. Web browser reads the HTML file. The <script> tag with src attribute indicates the browser should load JavaScript from an external file.
3. Web browser requests file.js from the web server.
4. Web browser reads and executes the JavaScript file. The alert() function displays a dialog box and waits for the user to press enter.
5. After the user presses enter, web browser finishes reading the JavaScript file and continues reading the HTML file.
6. Web browser requests the image file, and the web server responds with the image file.
7. Web browser finishes reading HTML file.

PARTICIPATION ACTIVITY 7.1.7: Downloading JavaScript files.

1) A web browser will process the HTML following a script element that uses an external JavaScript file while the browser waits for the web server to return the JavaScript file.
   - True
   - False

2) One script element can be used to include both inline JavaScript and a reference to an external JavaScript file.

- True
- False

3) One script element can be used to reference multiple external JavaScript files.

- True
- False

Loading JavaScript with async and defer

Although the `<script>` tag can be included anywhere in the head or body, good practice is to include the `<script>` tag in the head with the `async` or `defer` attributes set.

The `<script>` tag's ***async attribute*** allows the browser to process the webpage concurrently with loading and processing the JavaScript.

The `<script>` tag's ***defer attribute*** allows the browser to load the webpage concurrently with loading the JavaScript, but the JavaScript is not processed until the webpage is completely loaded.

PARTICIPATION ACTIVITY 7.1.8: Using the async attribute with the <script> tag.

index.html

```
<!DOCTYPE html>
<html>
  <title>Async Example</title>
  <script src="file.js" async></script>
  <body>
    <p>A piece of text.</p>
    <img src="dog.jpg" alt="dog">
    <p>Some more text.</p>
  </body>
</html>
```

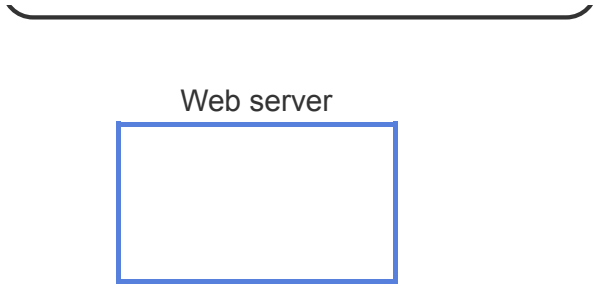
file.js

Web browser

A piece of text.

Some more text.

```
alert("Press enter to continue.");
```



Animation content:

The following code snippets are displayed:
Index.html:
<!DOCTYPE html>
<html>
 <title>JavaScript Example</title>
 <script src="file.js" async></script>
 <body>
  A piece of text.
  <img src="dog.jpg" alt="dog">
  Some more text.
 </body>
</html>

file.js:
alert("Press enter to continue.");

A web browser window and a web server window are also shown.

Step 1:
Index.html code snippet is displayed coming from the web server when the web browser requests it. The line
" <script src="file.js" async></script>" is highlighted. The web browser then makes a request to the web server for the file.jsm but keeps on going through the HTML code without waiting for the requested file to load. The code " A piece of text. " is then highlighted and the text "A piece of text." appears on the web browser.

Step 2:
The code " <img src="dog.jpg" alt="dog">" is then highlighted and the web browser makes a

request to the web server for the dog.jpg file. At the same time, the web server returned the file.js file.

Step 3:
The web browser now displays the text "dog.jpg". Following is an alert window saying "Press enter to continue.".

Step 4:
The code "alert("Press enter to continue.");" is highlighted. Once the user presses enter, the "dog.jpg" file finishes loading and displays the correct image.

Step 5:
The web browser finishes reading the HTML file, highlighting the line " Some more text. ". The text "Some more text." appears on the web browser window, below the dog image.

Animation captions:

1. Web browser reads index.html. <script> tag's async attribute causes browser to continue reading HTML without waiting for JavaScript file to load.
2. Web server responds with file.js while the browser requests the image file.
3. Web browser begins reading and executing the JavaScript file and pauses reading the HTML file. The web server concurrently responds to the image request.
4. After the user presses enter, web browser finishes reading the JavaScript file and continues processing the HTML file by displaying the dog.jpg image that was received.
5. Web browser finishes reading HTML file.

PARTICIPATION ACTIVITY 7.1.9: Using the defer attribute with the <script> tag.

index.html

```
<!DOCTYPE html>
<html>
  <title>Defer Example</title>
  <script src="file.js" defer></script>
  <body>
    <p>A piece of text.</p>
    <img src="dog.jpg" alt="dog">
    <p>Some more text.</p>
  </body>
```

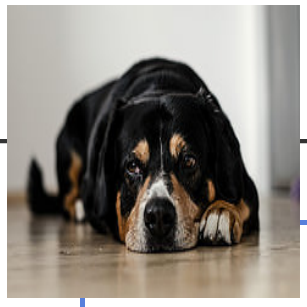
Web browser

A piece of text.

```
</html>
```

file.js

```
alert("Press enter to continue.");
```



Animation content:

The following code snippets are displayed:
Index.html:
<!DOCTYPE html>
<html>
 <title>JavaScript Example</title>
 <script src="file.js" defer></script>
 <body>
  A piece of text.
  <img src="dog.jpg" alt="dog">
  Some more text.
 </body>
</html>

file.js:
alert("Press enter to continue.");

A web browser window and a web server window are also shown.

Step 1:
Index.html code snippet is displayed coming from the web server when the web browser requests the code. The line
" <script src="file.js" defer></script>" is highlighted. The web browser then makes a request to the web server for the file.jsm but keeps on going through the HTML code without waiting for the requested file to load. The code " A piece of text. " is then highlighted and the text "A piece of text." appears on the web browser.

Step 2:
The code " <img src="dog.jpg" alt="dog">" is then highlighted and the web browser makes a request to the web server for the dog.jpg file. At the same time, the web server returned the file.js file.

Step 3:
The web browser now displays the text "dog.jpg". The web browser does not process the file.js due to the defer attribute so the web browser continues to process the HTML. The dog image is displayed on the web browser. The web browser finishes reading the HTML file, highlighting the line " Some more text. ". The text "Some more text." appears on the web browser window, below the dog image.

Step 4:
After the web browser has finished processing the HTML file, the web browser then goes to process the file.js file. An alert window is displayed on the web browser with the text "Please enter to continue.".

Step 5:
Once the user presses enter, the web browser finishes reading the Javascript file, and the alert window disappears.

Animation captions:

1. Web browser reads index.html. <script> tag's defer attribute causes browser to continue reading HTML without waiting for JavaScript file to load.
2. Web server responds with the JavaScript file while the browser requests the image file.
3. Web browser does not immediately process the JavaScript file due to the defer attribute. Instead, the browser continues to process the HTML.
4. After reading the HTML file, the web browser reads and executes the JavaScript file.
5. After the user presses enter, web browser finishes reading the JavaScript file.

PARTICIPATION ACTIVITY 7.1.10: Loading JavaScript.

1) The browser interprets the `defer` and `async` attributes for the script element the same.

   - True
   - False

2) When using a third-party JavaScript library, the `defer` attribute is usually better than the `async` attribute.

   - True
   - False

3) When writing custom JavaScript, the `defer` attribute is usually better than the `async` attribute.

   - True
   - False

4) Most webpages on the internet were written before the `defer` or `async` attributes were standardized.

   - True
   - False

Minification and obfuscation

To reduce the amount of JavaScript that must be downloaded from a web server, developers often minify a website's JavaScript. ***Minification*** or ***minimization*** is the process of removing unnecessary characters (like whitespace and comments) from JavaScript code so the code executes the same but with fewer characters. Minification software may also rename identifiers into shorter ones to reduce space. Ex: `let totalReturns = 10;` may be converted into `let a=10;`.

Minified JavaScript is typically stored in a file with a ".min.js" file extension. An example of minified code from the Bootstrap project is shown below.

```
// Excerpt from bootstrap.min.js
a.fn.button=b,a.fn.button.Constructor=c,a.fn.button.noConflict=function()
{
return a.fn.button=d,this},a(document).on("click.bs.button.data-api",
'[data-toggle^="button"]',function(c){let d=a(c.target).closest(".btn");
b.call(d,"toggle"),a(c.target).is('input[type="radio"],
```

A JavaScript ***obfuscator*** is software that converts JavaScript into an unreadable form that is very difficult to convert back into readable JavaScript. Developers obfuscate a website's JavaScript to prevent the code from being read or re-purposed by others. Obfuscated code may also be minified and appear in a ".min.js" file.

CHALLENGE ACTIVITY 7.1.1: JavaScript with HTML.

550544.4142762.qx3zqy7

Start

Use the document object's writeln() method to display the current platform in a <p> element in the webpage. Hint: Provide document.writeln() with a string argument that uses the plus operator, +, to join p's starting tag, window.navigator.platform, and p's ending tag.

```
1 <h1>Demo</h1>
2 <script>
3    document.writeln(/* Your solution goes here */);
4 </script>
```

1 2

Check Next

Exploring further:

- Window object from MDN
- Console object from MDN
- async vs defer attributes from Growing with the Web
- JavaScript minifiers: javascript-minifier.com and jscompress.com
- JavaScript obfuscators: javascriptobfuscator.com and JS-obfus

7.2 Document Object Model (DOM)

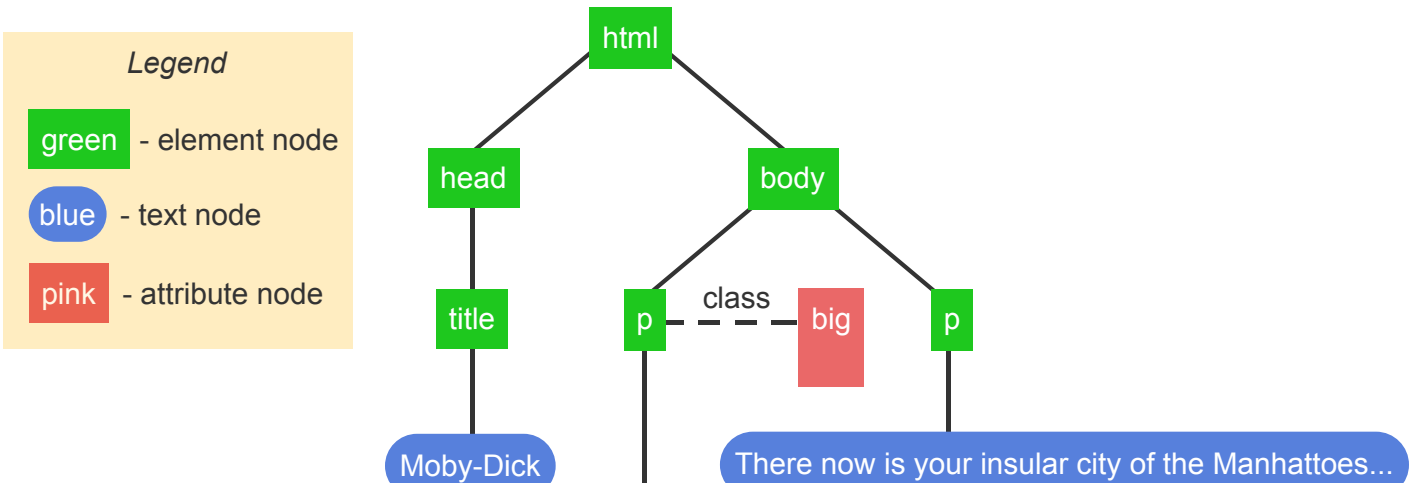
DOM structure

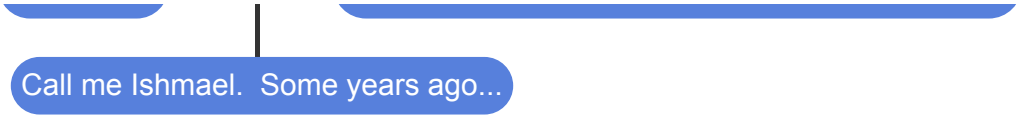
The Document Object Model (DOM) is a data structure corresponding to the HTML document displayed in a web browser. A DOM tree is a visualization of the DOM data structure. A ***node*** is an individual object in the DOM tree. Nodes are created for each element, the text between an element's tags, and the element's attributes.

- The ***root node*** is the node at the top of the DOM.
- A ***child node*** is the node directly under another node. A node can have zero, one, or more child nodes (children).
- A ***parent node*** is the node directly above another node. All nodes, except the root node, have one parent node.

PARTICIPATION ACTIVITY 7.2.1: Creating the DOM from HTML.

```
<html>
  <title>Moby-Dick</title>
  <body>
    <p class="big">Call me Ishmael.  Some years ago...</p>
    <p>There now is your insular city of the Manhattoes...</p>
  </body>
</html>
```





Animation content:

The following code snippet is displayed:

```
<html>
 <title>Moby-Dick</title>
 <body>
   <p class="big">Call me Ishmael.  Some years ago...</p>
   <p>There now is your insular city of the Manhattoes...</p>
 </body>
<head>
```

A legend contains a green element node, a blue text node, and a pink attribute node.

A tree is shown. At the top of the tree is an element node named html with two child element nodes: head and body. Head has a child element node title, which also has a child text node "Moby-Dick". Body has two p child element nodes. The first p node has an attribute node connected to it labeled class=big, and a child text node that reads: "Call me Ishmael. Some years ago...". The second p node has only one child text node that reads: "There now is your insular city of the Manhattoes...".

Step 1:
The web browser reads the HTML and creates the DOM's root node from the html element.

Step 2:
Although no head element exists in the HTML, a head node is created as a child of the html node. The title node is added as a child of the head node.

Step 3:
A text node is created for the title element's text content.

Step 4:
The body node is a child of the root node. The p element is contained within the body element, so the p node is a child of the body node.

Step 5:
An attribute node is created for the p element's class attribute. Attribute nodes are always connected to element nodes and are not considered children.

Step 6:
The browser continues reading the HTML and creating DOM nodes.

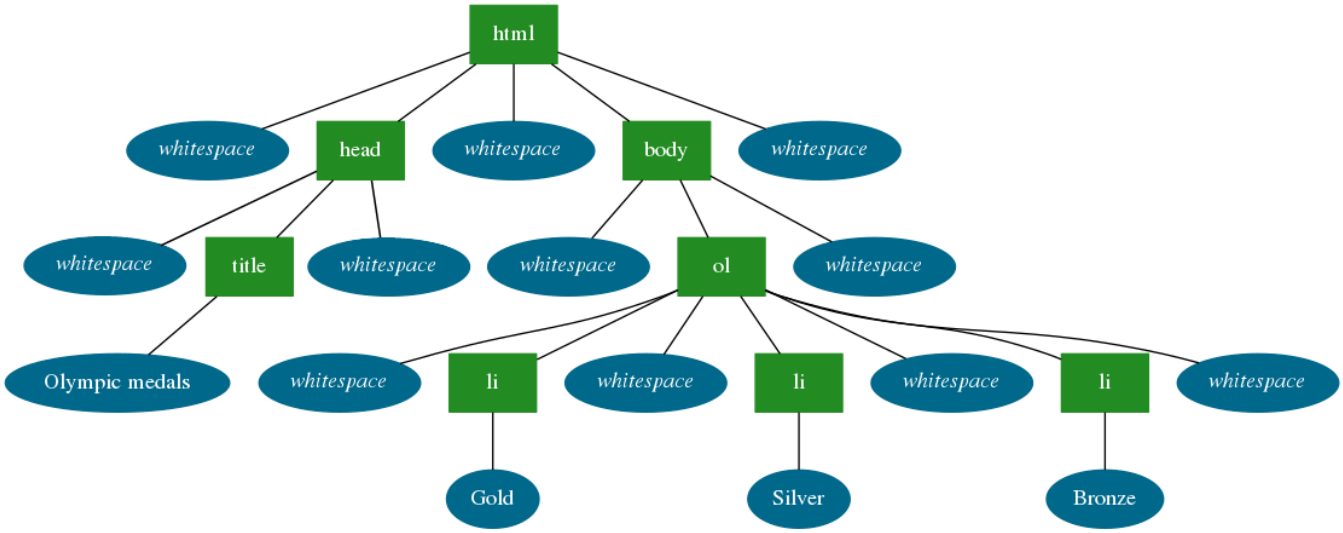
Animation captions:

1. The web browser reads the HTML and creates the DOM's root node from the html element.
2. Although no head element exists in the HTML, a head node is created as a child of the html node. The title node is added as a child of the head node.
3. A text node is created for the title element's text content.
4. The body node is a child of the root node. The p element is contained within the body element, so the p node is a child of the body node.
5. An attribute node is created for the p element's class attribute. Attribute nodes are always connected to element nodes and are not considered children.
6. The browser continues reading the HTML and creating DOM nodes.

An idealized representation of the DOM tree excludes text nodes that only contain whitespace. However, a web developer occasionally needs to know the complete DOM tree, which includes whitespace as shown in the example below.

Figure 7.2.1: Complete DOM tree visualization with whitespace text nodes.

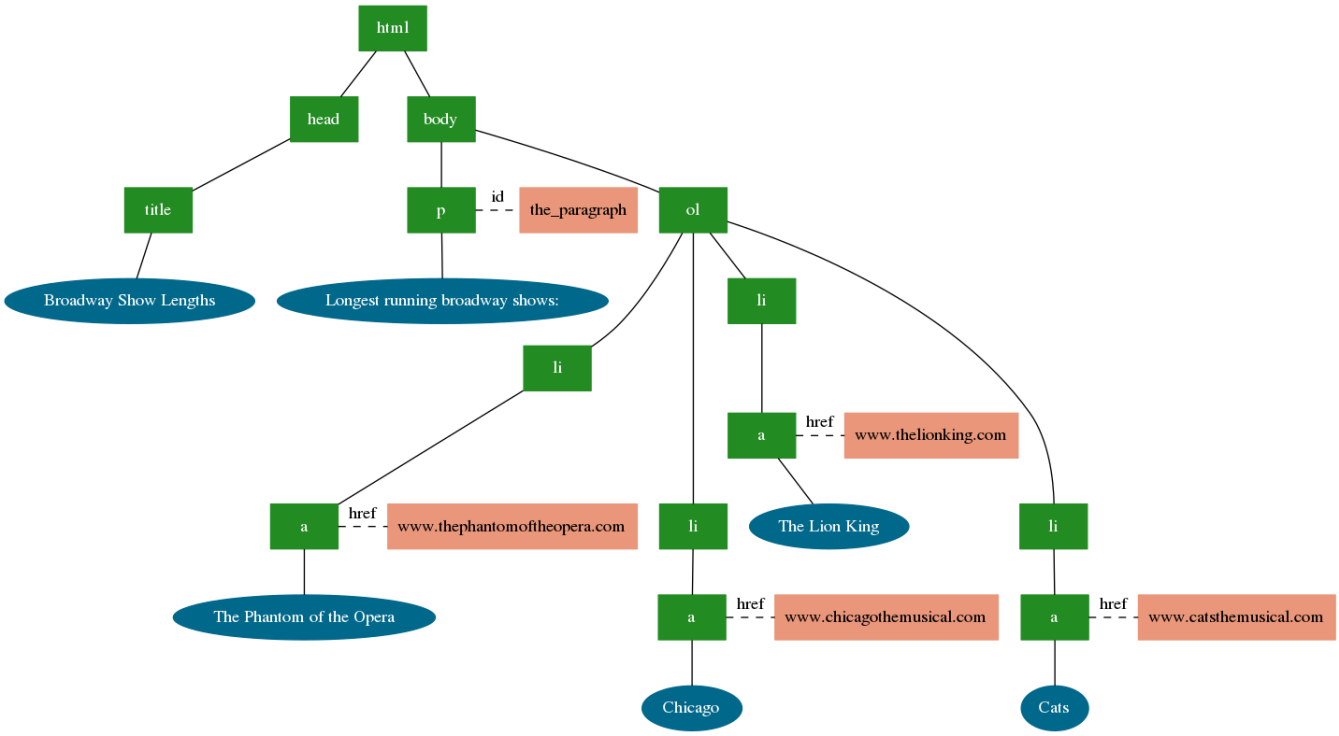
```
<!DOCTYPE html>
<html>
   <head>
      <title>Olympic medals</title>
   </head>
   <body>
      <ol>
         <li>Gold</li>
         <li>Silver</li>
         <li>Bronze</li>
      </ol>
   </body>
</html>
```



PARTICIPATION ACTIVITY 7.2.2: DOM structure.

Refer to the HTML and resulting DOM below.

```
<!DOCTYPE html>
<html>
   <head>
      <title>Broadway Show Lengths</title>
   </head>
   <body>
      <p id="the_paragraph">Longest running broadway shows:</p>
      <ol>
         <li><a href="http://www.thephantomoftheopera.com/">The Phantom
of the Opera</a></li>
         <li><a href="http://www.chicagothemusical.com/">Chicago</a></li>
         <li><a href="http://www.thelionking.com/">The Lion King</a></li>
         <li><a href="http://www.catsthemusical.com/">Cats</a></li>
      </ol>
   </body>
</html>
```



1) Which node is the root of the DOM tree?

- html
- a
- body

2) Which of the following DOM nodes is never a child node?

- Element node
- Attribute node
- Text node

3) How many child nodes does the p element have?

- 0
- 1
- 2

4) How many child nodes does the ol element have?

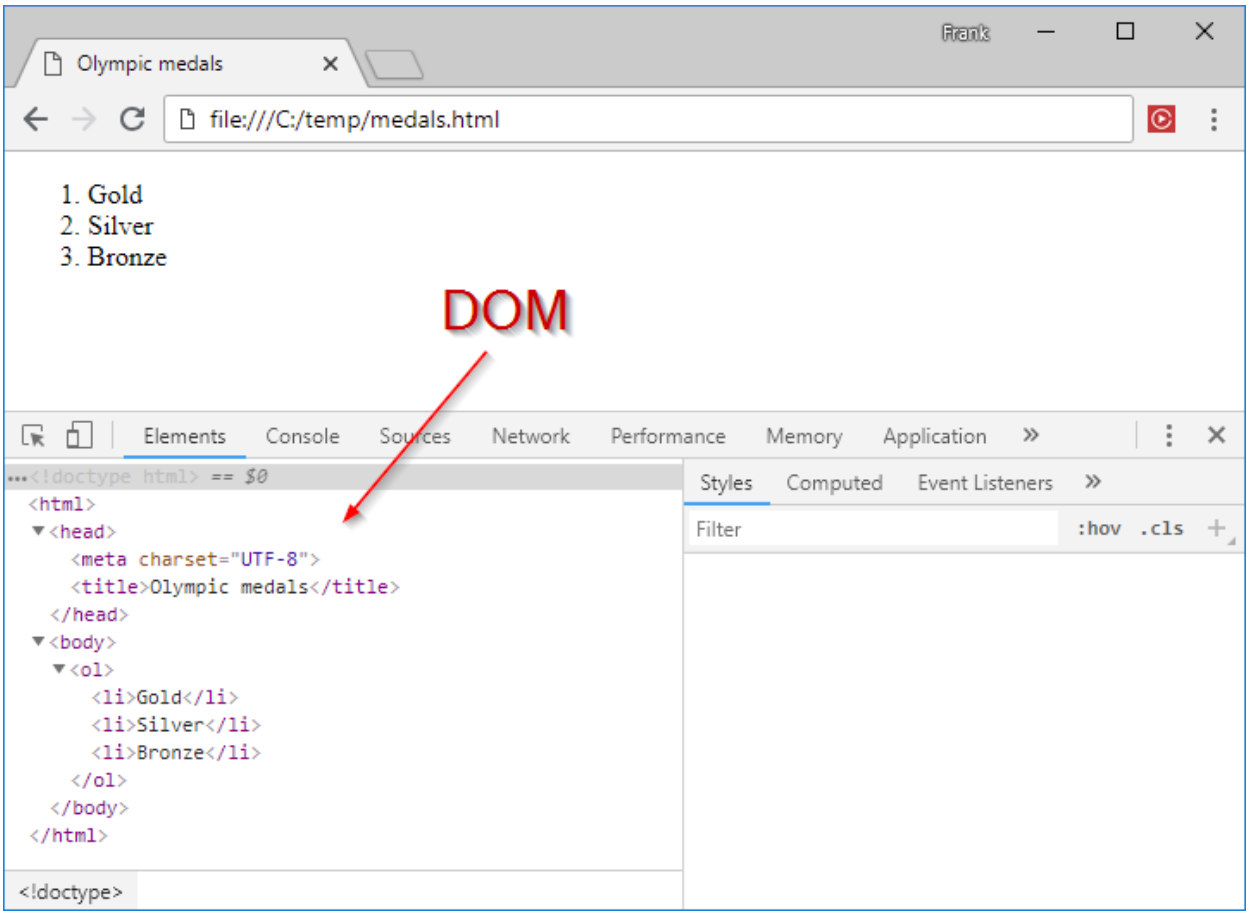
- 0
- 2
- 4

5) Which DOM nodes never have child nodes?

- Element nodes
- Attribute nodes only
- Text nodes and attribute nodes

Viewing the DOM in Chrome

The Chrome DevTools can display an HTML document's DOM by pressing Ctrl+Shift+C on Windows or Ctrl+Option+C on a Mac. The DOM may differ from the HTML. Ex: The `<head>` tag may be missing from the HTML file but is visible in the DOM below because `<meta>` and `<title>` elements are always placed in the `<head>` element.



Searching the DOM

JavaScript is commonly used to search the DOM for a specific node or set of nodes and then change the nodes' attributes or content. Ex: In an email application, the user may click a Delete button to delete an email. The JavaScript must search the DOM for the node containing the email's contents and then change the contents to read "Email deleted".

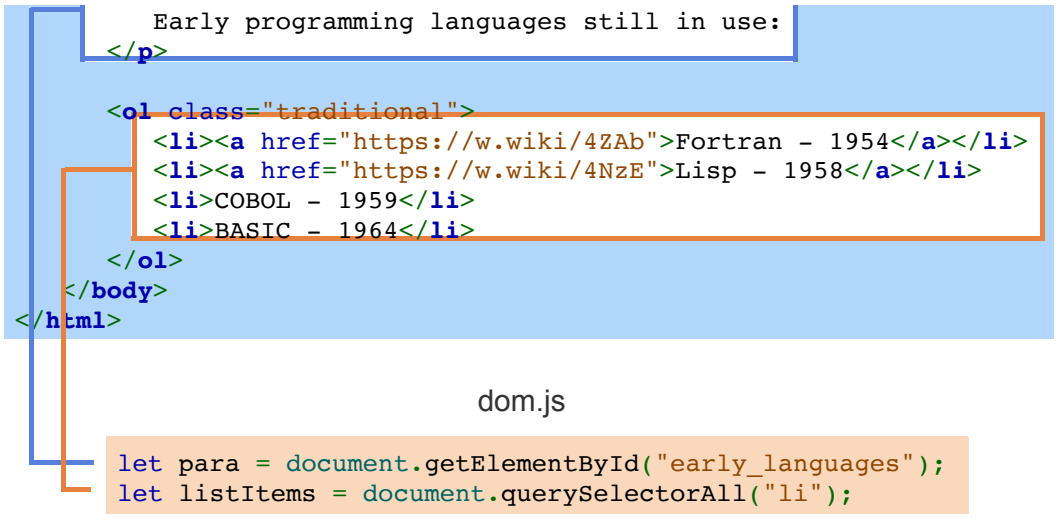
The `document` object provides five primary methods that search the DOM for specific nodes:

1. The ***document.getElementById()*** method returns the DOM node whose `id` attribute is the same as the method's parameter.
   Ex: `document.getElementById("early_languages")` returns the p node in the HTML below.
2. The ***document.getElementsByTagName()*** method returns an array of all the DOM nodes whose type is the same as the method's parameter.
   Ex: `document.getElementsByTagName("li")` returns an array containing the four li nodes from in the HTML below.
3. The ***document.getElementsByClassName()*** method returns an array containing all the DOM nodes whose `class` attribute matches the method's parameter.
   Ex: `document.getElementsByClassName("traditional")` returns an array containing the `ol` node with the `class` attribute matching the word traditional.
4. The ***document.querySelectorAll()*** method returns an array containing all the DOM nodes that match the CSS selector passed as the method's parameter.
   Ex: `document.querySelectorAll("li a")` returns an array containing the two anchor nodes in the HTML below.
5. The ***document.querySelector()*** method returns the first element found in the DOM that matches the CSS selector passed as the method's parameter. `querySelector()` expects the same types of parameters as `querySelectorAll()` but only returns the first element found while navigating the DOM tree in a depth-first traversal.
   Ex: `document.querySelector("li")` returns the li node about Fortran.

A DOM search method name indicates whether the method returns one node or an array of nodes. If the method name starts with "getElements" or ends in "All", then the method returns an array, even if the array contains one node or is empty. `getElementById()` and `querySelector()` either return a single node or null if no node matches the method arguments.

PARTICIPATION ACTIVITY 7.2.3: Searching the DOM by id and by element.

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <title>Early Programming Languages</title>
      <script src="dom.js" defer></script>
   </head>
   <body>
      <p id="early_languages">
```



Animation content:

The following code is displayed.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Early Programming Languages</title>
    <script src="dom.js" defer></script>
  </head>
  <body>
    <p id="early_languages">
      Early programming languages still in use:
    </p>

    <ol class="traditional">
      <li><a href="https://w.wiki/4ZAb">Fortran - 1954</a></li>
      <li><a href="https://w.wiki/4NzE">Lisp - 1958</a></li>
      <li>COBOL - 1959</li>
      <li>BASIC - 1964</li>
    </ol>
  </body>
</html>
```

Step 1: The line of code reading "<script src="dom.js" defer></script>" is highlighted. dom.js is

displayed as follows.

```
let para = document.getElementById("early_languages");
let listItems = document.querySelectorAll("li");
```

Step 2: The line of code assigning the para variable is highlighted and connects to the <p> element.

Step 3: The line of code assigning the listItems variable is highlighted and connects to the <li> elements in the HTML.

Animation captions:

1. The webpage loads dom.js with the defer attribute, so the code in dom.js only executes after the webpage and DOM are finished loading.
2. dom.js calls getElementById() to find the paragraph with id "early_languages".
3. dom.js calls querySelectorAll() to find all four li elements.

HTMLCollection and NodeList

Technically, `getElementsByTagName()` and `getElementsByClassName()` return an HTMLCollection, and `querySelectorAll()` returns a NodeList. ***HTMLCollection*** *is an interface representing a generic collection of elements. A* ***NodeList*** *is an object with a collection of nodes. HTMLCollection and NodeList both act like an array. Both have a `length` property, and elements can be accessed with braces. Ex: `elementList[0]` is the first element in an HTMLCollection or NodeList.*

PARTICIPATION ACTIVITY 7.2.4: DOM traversal.

Refer to the HTML below.

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Web development languages</title>
      <link rel="stylesheet" href="styles.css">
      <script src="dom.js" defer></script>
   </head>
   <body>
      <p class="special">
         Languages used in web development:
      </p>
      <ul id="list">
         <li id="item-1">HTML for content</li>
         <li id="item-2">CSS for presentation</li>
         <li id="item-3">JavaScript for functionality</li>
      </ul>
      <p class="special">
         <a href="https://en.wikipedia.org/wiki/Web_development">More
information</a>
      </p>
   </body>
</html>
```


Select the element(s) or value returned by the given search method.

1) `document.getElementById("list")`

- ul element
- First li element
- All li elements

2) `document.getElementsByTagName("li")`

- ul element
- First li element
- All li elements

3) `document.getElementsByTagName("ul")[0]`

- ul element
- First li element
- null

4) `document.querySelectorAll("li")`

- ul element
- First li element
- All li elements

5) `document.querySelector("#list")`

- ul element
- First li element
- null

6) `document.querySelector(".special")`

- First p element
- Last p element
- Both p elements

7) `document.querySelector("item-3")`

- ul element
- Last li element
- null

Modifying DOM node attributes

After searching the DOM for an element, JavaScript may be used to examine the element's attributes or to change the attributes. By modifying attributes, JavaScript programs can perform actions including:

- Change which image is displayed by modifying an img element's **`src`** attribute.
- Determine which image is currently displayed by reading the img element's **`src`** attribute.
- Change an element's CSS styling by modifying an element's **`style`** attribute.

Every attribute for an HTML element has an identically named property in the element's DOM node. Ex: `<a href="https://www.nasa.gov/" id="nasa_link">NASA</a>` has a corresponding DOM node with properties named **`href`** and **`id`**. Each attribute property name acts as both a getter and a setter.

- Getter: Using the property name to read the value allows a program to examine the attribute's value. Ex: `nasaUrl = document.getElementById("nasa_link").href` assigns `nasaUrl` the string `"https://www.nasa.gov/"` from the anchor element's `href` attribute.
- Setter: Writing to a property allows a program to modify the attribute, which is reflected in the rendered webpage. Ex: `document.getElementById("nasa_link").href = "https://www.spacex.com/"` changes the element's hyperlink to the SpaceX URL.

An element's attribute can be removed using the element method ***removeAttribute()***. Ex: `document.getElementById("nasa_link").removeAttribute("href")` removes the link from the anchor element so that clicking on the HTML element no longer performs an action.

PARTICIPATION ACTIVITY 7.2.5: Modifying DOM node attributes.

```
<body>
   <img src="mountain.jpg"
        alt="Mountain photo">
   <p style="color: green">
      I love the outdoors!
   </p>
</body>
```

```
let img = document.querySelector("img");
img.src = "mountain.jpg";
img.alt = "Mountain photo";

let p = document.querySelector("p");
p.style = "color: green";
```

Animation content:

Step 1: The following code is displayed.
<body>
  <img src="mountain.jpg"
     alt="Mountain photo">
  <p>

```
    I love the outdoors!
  </p>
</body>
```

A photo of a lake is displayed with the text "I love the outdoors!" beneath the photograph.

Step 2: The following code is added.

```
let img = document.querySelector("img");
img.src = "lake.jpg";
img.alt = "Lake photo";

let p = document.querySelector("p");
p.style = "color: green";
```

The first block of code is edited to read the following.

```
<body>
  <img src="mountain.jpg"
      alt="Mountain photo">
  <p>
    I love the outdoors!
  </p>
</body>
```

A photo of a mountain is displayed with the text "I love the outdoors!" beneath the photograph.

Step 3: The following lines of code are highlighted.

```
let p = document.querySelector("p");
p.style = "color: green";
```

The first block of code is edited to read the following.

```
<body>
  <img src="mountain.jpg"
      alt="Mountain photo">
  <p style="color: green">
    I love the outdoors!
```

```
  </p>
</body>
```

The line of text reading "I love the outdoors!" is now displayed in a green font.

Animation captions:

1. The webpage shows a photo of a lake with a sentence underneath.
2. Changing the img element's src attribute causes the webpage to replace lake.jpg with mountain.jpg.
3. Setting the p element's style attribute changes the paragraph's CSS styling, making the text green.

PARTICIPATION ACTIVITY 7.2.6: Reading and modifying DOM node attribute values.

Refer to the HTML below.

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Orion mission</title>
      <script src="dom.js" defer></script>
   </head>
   <body>
      <h1>Orion Moon Mission Update</h1>
      <img src="inFlight.png" id="status" width="200"
           alt="Image of spaceship en route to the moon">
   </body>
</html>
```

1) Enter the missing attribute to make the image display success.png.

```
let pic =
document.getElementById("status");
pic._____ = "success.png";
```

Check Show answer

2) Enter the node and attribute required to read the current image name.

```
let pic =
document.getElementById("status");
if ( _____ != "success.png") {
   alert("Wrong picture!");
}
```

Check Show answer

3) Enter the method name to remove the image's `width` attribute.

```
let pic =
document.getElementById("status");
pic._____("width");
```

Check Show answer

Modifying DOM node content

After searching the DOM for an element, JavaScript may be used to examine or change the element's content.

Two common properties are used to get or set an element's content:

1. The ***textContent*** property gets or sets a DOM node's text content. Ex: `document.querySelector("p").textContent = "$25.99";` changes the paragraph to `<p>$25.99</p>`.
2. The ***innerHTML*** property gets or sets a DOM node's content, including all of the node's children, using an HTML-formatted string. Ex: `document.querySelector("p").innerHTML = "<strong>$25.99</strong>";` changes the paragraph to `<p><strong>$25.99</strong></p>`.

The `innerHTML` property uses an internal parser to create any new DOM nodes. `textContent`, however, only creates or changes a single text node. For setting an element's text, `textContent` is

somewhat faster than `innerHTML` because no HTML parsing is performed.

PARTICIPATION ACTIVITY 7.2.7: Modifying a webpage with textContent and innerHTML.

```
<body>
   <p>
      Most common girl names in 2020:
   </p>
   <ol>
      <li>Emma</li>
      <li>Olivia</li>
      <li>Ava</li>
   </ol>
</body>
```

Most common girl names in 2020:

1. Emma
2. Olivia
3. Ava

```
let p = document.querySelector("p");
p.textContent = "Most common girl names in 2020:";

let ol = document.querySelector("ol");
ol.innerHTML = "<li>Emma</li><li>Olivia</li><li>Ava</li>";
```

Animation content:

The following HTML is displayed:
<body>

    Most common girl names in 2020:

    Emily
    Hannah
    Madison

</body>

The following JavaScript is displayed:
let p = document.querySelector("p");
p.textContent = "Most common girl names in 2020:";

```
let ol = document.querySelector("ol");
ol.innerHTML = "<li>Emma</li><li>Olivia</li><li>Ava</li>";
```

Step 1: The line of code assigning p.textContent changes the HTML p element to read "Most common girl names in 2020:" and displays the text in the browser.

Step 2: The line of code assigning ol.innerHTML changes the HTML li elements to the new names Emma, Olivia, and Ava. The new names replace the previous names in the browser.

Animation captions:

1. Assigning the textContent property replaces the paragraph's text.
2. Assigning the innerHTML property replaces the entire list.

Less common ways to change node content

*The **nodeValue** property gets or sets the value of text nodes. As the DOM tree represents textual content separately from HTML elements, the textual content of an HTML element is the first child node of the HTML element's node. So, to access the textual content of an HTML element within the DOM, `firstChild.nodeValue` is used to access the value of the HTML's element's first child.*

Ex:
`document.getElementById("saleprice").firstChild.nodeValue = "$25.99"`:

1. *Gets the DOM node for the element with id "saleprice".*
2. *Uses `firstChild` to access the textual content node for the element.*
3. *Uses `nodeValue` to update the content.*

*The **innerText** property gets or sets a DOM node's rendered content. `innerText` is similar to `textContent`, but `innerText` is aware of how text is rendered in the browser. Ex: In `<p>Testing one</p>`, `textContent` returns "Testing one" with spaces, but `innerText` returns "Testing one" with the spaces collapsed into a single space.*

PARTICIPATION ACTIVITY 7.2.8: Modify the DOM nodes.

An HTML element using the ***hidden attribute*** is not displayed by the web browser.

Add JavaScript in the `changePage()` function so that clicking on the Use Current Astronomy button does the following:

1. Uses `removeAttribute()` to remove the `hidden` attribute from the paragraph with the id `p2`, causing the paragraph to become visible.
2. Uses the `textContent` property of the span with id `lastPlanet` to change the name of the farthest planet to "Neptune".
3. Uses the `innerHTML` property of the paragraph with id `p4` to add a link: `<a href="https://en.wikipedia.org/wiki/Pluto">Source</a>`

Use an appropriate DOM search method like `document.getElementById()` to access the DOM nodes.

HTML JavaScript

```
<body>
   <h1>The farthest planet</h1>

   <p id="p1">Pluto was discovered in 1930 and designated as a planet
   <p id="p2" hidden>In 2006, Pluto was reclassified as a dwarf plane
   <p id="p3"><span id="lastPlanet">Pluto</span> is the farthest plar
   <p id="p4"></p>

   <input type="button" value="Use Current Astronomy">
</body>
```

Render webpage Reset code

Your webpage

The farthest planet

Pluto was discovered in 1930 and designated as a planet.

Pluto is the farthest planet from the Sun.

Use Current Astronomy

Expected webpage

The farthest planet

Pluto was discovered in 1930 and designated as a planet.

Pluto is the farthest planet from the Sun.

Use Current Astronomy

▶ View solution

PARTICIPATION ACTIVITY 7.2.9: Changing DOM node content.

Refer to the HTML and JavaScript below.

sitcom.html

```
<!DOCTYPE HTML>
<html lang="en">
   <head>
      <title>Sitcoms</title>
      <script src="sitcom.js" defer></script>
   </head>
   <body>
      <p>My <span>favorite</span> TV sitcoms:</p>
      <ol>
         <li><cite>The Office</cite></li>
         <li><cite>Community</cite></li>
         <li><cite>The Fresh Prince of Bel-Air</cite></li>
      </ol>
   </body>
</html>
```

sitcom.js

```
let span = document.querySelector("span");
span.__A__ = "least favorite";

let listItems = document.__B__("li");
for (let item of listItems) {
   item.__C__ = "<cite>Happy Hour</cite>";
}
```

Match the letters to the missing JavaScript property or method.

If unable to drag and drop, refresh the page.

C B A

	querySelectorAll
	innerHTML

textContent

Reset

CHALLENGE ACTIVITY 7.2.1: Using the Document Object Model.

550544.4142762.qx3zqy7

Start

All steps in this Challenge Activity require calling a document method to search the DOM. Write the JavaScript to assign listNodes with all elements with a class name of 'special-language'.

HTML JavaScript

```
<h1>Top 10 TIOBE index for March 2018:</h1> <!-- https://www.tiobe.co
<ol class="languages-list">
   <li class="special-language">Java</li>
   <li>C</li>
   <li class="special-language">C++</li>
   <li class="special-language">Python</li>
   <li class="special-language">C#</li>
   <li>Visual Basic .NET</li>
   <li>PHP</li>
   <li>JavaScript</li>
   <li>Ruby</li>
   <li class="special-language">SQL</li>
</ol>
```

1 2 3 4

Check Next

Exploring further:

- Document Object Model (DOM) from MDN

7.3 More DOM modification

Accessing nodes

The JavaScript object ***document.documentElement*** is the root of the DOM tree. Ex: `let html = document.documentElement;` assigns the `html` variable with the HTML document's root node.

DOM nodes have properties for accessing a node's parent, children, and siblings:

1. The ***parentNode*** property refers to the node's parent. Ex: In the figure below, the ol node is the `parentNode` for all li nodes.
2. The ***childNodes*** property is an array-like collection of objects for each of the node's children. Ex: In the figure below, the li nodes and whitespace text nodes are the ol node's `childNodes`.
3. The ***children*** property is similar to the `childNodes` except the array contains only element nodes and no text nodes. Ex: In the figure below, the li nodes are the ol node's `children`.
4. The ***nextElementSibling*** property refers to the element node with the same parent following the current node in the document. Ex: In the figure below, the ol node is the p node's `nextElementSibling`.
5. The ***previousElementSibling*** property refers to the element node with the same parent preceding the current node in the document. Ex: In the figure below, the p node is the ol node's `previousElementSibling`.

A common error is to use the `childNodes` property instead of the `children` property when iterating through the items of a list. The `children` property only contains the list items, while the `childNodes` property also contains the whitespace text nodes between the list items.

Figure 7.3.1: Example HTML for node properties.

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Geologic eons of earth</title>
   </head>
   <body>
      <p>The four geologic eons on earth:</p>
      <ol>
         <li><a href="https://en.wikipedia.org/wiki/Hadean">Hadean</a>
</li>
         <li><a href="https://en.wikipedia.org/wiki/Archean">Archean</a>
</li>
         <li><a
href="https://en.wikipedia.org/wiki/Proterozoic">Proterozoic</a></li>
         <li><a
href="https://en.wikipedia.org/wiki/Phanerozoic">Phanerozoic</a></li>
      </ol>
   </body>
</html>
```

PARTICIPATION ACTIVITY 7.3.1: Using nextElementSibling, previousElementSibling, and parentNode.

Refer to the HTML below.

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Web development languages</title>
   </head>
   <body>
      <p>Languages used in web development.</p>
      <ul>
         <li id="item-1">HTML for content</li>
         <li id="item-2">CSS for presentation</li>
         <li id="item-3">JavaScript for functionality</li>
      </ul>
   </body>
</html>
```

Select the HTML element or node referenced by the given property.

1) `document.documentElement`

- html
- body
- meta

2) `document.documentElement.children[1]`

- html
- body
- title

3) `document.documentElement.children[1].children[1]`

- p
- ul
- One of the li elements

4) `document.documentElement.children[1].parentNode`

- html
- body
- title

5) `document.getElementById("item-2").previousElementSibling`

- li with id item-1
- li with id item-2
- li with id item-3

6) `document.getElementById("item-2").parentNode`

- li with id item-1
- li with id item-2
- ul

7) `document.getElementById("item-1").nextElementSibling`

- li with id item-1
- li with id item-2
- li with id item-3

8) `document.querySelector("ul").childNodes[0]`

- li with id item-1
- li with id item-2
- text node

Modifying the DOM structure

Various DOM node methods can change a node's location within the DOM or remove nodes:

- The ***appendChild()*** method appends a DOM node to the object's child nodes. The code below moves the ordered list's first list item to the last list item of the same ordered list.

```
ol = document.getElementsByTagName("ol")[0];
li = ol.getElementsByTagName("li")[0];
ol.appendChild(li);
```

- The ***insertBefore()*** method inserts a DOM node as a child node before an object's existing child node. The code below moves the ordered list's first list item before the fourth list item.

```
ol = document.getElementsByTagName("ol")[0];
items = ol.getElementsByTagName("li");
ol.insertBefore(items[0], items[3]);
```

- The ***removeChild()*** method removes a node from the object's children. The most common usage pattern is to get a DOM node, n, and call `removeChild()` on n's parent passing n as an argument. Ex: `n.parentNode.removeChild(n)`

The following methods are for creating new nodes or duplicating existing nodes:

- The `document` method ***createElement()*** returns a new element node, created from a string argument that names the HTML element. Ex: `document.createElement("p")` creates a new paragraph node.
- The `document` method ***createTextNode()*** returns a new text node containing the text

specified by a string argument. Ex: `document.createTextNode("Hello there!")` creates the text node with "Hello there!", which can then be appended to an element node.

- The node method ***cloneNode()*** returns an identical copy of a DOM node. The method's boolean argument indicates whether the method should also clone the node's children. Ex: `x.cloneNode(true)` creates an identical tree rooted at x, but `x.cloneNode(false)` creates only a single node identical to x. *A common error is to forget to modify any id attributes in the cloned tree. The `cloneNode()` method does not ensure that new nodes have unique id attributes.*

After creating or cloning a node, the node does not appear in the webpage until the node is attached to the DOM tree. A programmer must use `appendChild()` or `insertBefore()` to add the new node to the existing DOM tree.

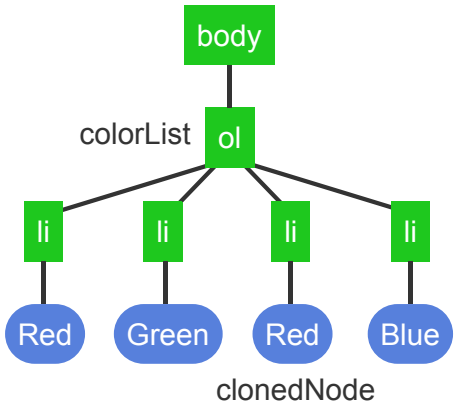
PARTICIPATION ACTIVITY 7.3.2: Modifying the DOM structure.

example.html

```
<!DOCTYPE HTML>
<html lang="en">
   <title>DOM Example</title>
   <script src="script.js" defer></script>
   <body>
      <ol>
         <li>Red</li>
         <li>Green</li>
      </ol>
   </body>
</html>
```

script.js

```
let listNode = document.createElement("li");
let textNode = document.createTextNode("Blue");
listNode.appendChild(textNode);

let colorList = document.querySelector("ol");
colorList.appendChild(listNode);

let itemNodes = colorList.querySelectorAll("li");
let clonedNode = itemNodes[0].cloneNode(true);
colorList.insertBefore(clonedNode, itemNodes[2]);
```



1. Red
2. Green
3. Red
4. Blue

Animation content:

The following code is displayed.

example.html

```
<!DOCTYPE HTML>
<html lang="en">
  <title>DOM Example</title>
  <script src="script.js" defer></script>
  <body>
    <ol>
      <li>Red</li>
      <li>Green</li>
    </ol>
  </body>
</html>
```

script.js

```
let listNode = document.createElement("li");
let textNode = document.createTextNode("Blue");
listNode.appendChild(textNode);

let colorList = document.querySelector("ol");
colorList.appendChild(listNode);

let itemNodes = colorList.querySelectorAll("li");
let clonedNode = itemNodes[0].cloneNode(true);
colorList.insertBefore(clonedNode, itemNodes[2]);
```

Step 1: The following lines of code are highlighted.

```
 <ol>
      <li>Red</li>
      <li>Green</li>
    </ol>
```

A "body" node is created as the head node. An "ol" is connected to the "body" node, and is connected to two "li" listNode objects. The textNode objects "Red" and "Green" are attached to their respective "li" listNodes.

The following ordered list is defined.

1. Red
2. Green

Step 2: The line of code reading "let listNode = document.createElement("li");" is highlighted, and a node labeled "li" is created to represent the listNode. Next, the line of code reading "let textNode = document.createTextNode("Blue");" is highlighted, and a node labeled "blue" is created to represent the textNode. Next, the line of code reading "listNode.appendChild(textNode);" is highlighted, and the "blue" node is attached to the "li" node.

Step 3: The following lines of code are highlighted.

```
let colorList = document.querySelector("ol");
colorList.appendChild(listNode);
```

The listNode created in the previous step is connected to the "ol" node. The browser now reads the following.

1. Red
2. Green
3. Blue

Step 4: The following lines of code are highlighted.

```
let itemNodes = colorList.querySelectorAll("li");
let clonedNode = itemNodes[0].cloneNode(true);
```

A copy is created of the "li" node and its child "Red" node.

Step 5: The line of code reading "colorList.insertBefore(clonedNode, itemNodes[2]);" is highlighted. The cloned "li" and "Red" nodes are placed between the "Green" and "Blue" nodes. The browser is updated to read the following.

1. Red
2. Green
3. Red
4. Blue

Animation captions:

1. The HTML file defines an ordered list with two colors.
2. After creating an element node and a text node, the text node is appended to the element node. listNode becomes textNode's parent.
3. appendChild() appends listNode to colorList's child nodes. Since listNode is now attached to the DOM, the "Blue" list item appears in the browser.
4. The cloneNode() method creates a copy of itemNodes[0] and child, duplicating the "Red" list item.
5. insertBefore() inserts the cloned "Red" list item before the "Blue" list item, which changes the DOM.

PARTICIPATION ACTIVITY 7.3.3: Adding and removing DOM nodes.

If unable to drag and drop, refresh the page.

removeChild() **appendChild()** **insertBefore()**

	A method on a DOM node which moves one DOM node to be a previous sibling to another DOM node.
	A method on a DOM node that deletes a DOM node from the DOM tree.
	A method on a DOM node that turns another DOM node into the first DOM node's last child.

Reset

PARTICIPATION ACTIVITY 7.3.4: Creating new DOM nodes.

1) Which DOM node method creates a copy of the node and the node's children?

- createElement()
- createTextNode()
- cloneNode(true)
- cloneNode(false)

2) Which `document` method creates a new DOM node for a specific HTML element?

- createElement()
- createTextNode()
- cloneNode(true)
- cloneNode(false)

3) Which `document` method creates a new DOM node to hold content?

- createElement()
- createTextNode()
- cloneNode(true)
- cloneNode(false)

4) Which DOM node method creates a copy of another DOM node, but not the node's children?

- createElement()
- createTextNode()
- cloneNode(true)
- cloneNode(false)

CHALLENGE ACTIVITY | 7.3.1: Manipulating the Document Object Model.

550544.4142762.qx3zqy7

Start

Assign newElement with a new h4 element. Assign textNode with a new textNode containing "Betty".

HTML | JavaScript

```
1 <div id="list">
2    <h4>Ann</h4>
3    <h4>John</h4>
4    <h4>Rose</h4>
5 </div>
```

1 | 2 | 3

Check Next

Exploring further:

- Document Object Model (DOM) from MDN

7.4 Event-driven programming

Events and event handlers

An ***event*** is an action, usually caused by a user, that the web browser responds to. Ex: A mouse movement, a key press, or a network response from a web server. Typically, the occurrence and timing of an event are unpredictable, since the user or web server can perform an action at any time.

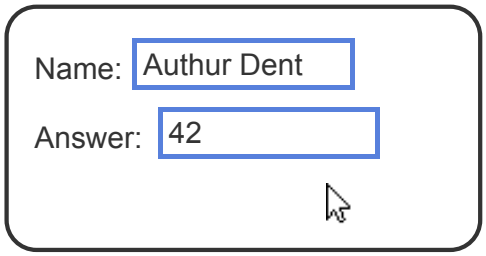
Event-driven programming is a programming style where code runs only in response to various events. Code that runs in response to an event is called an ***event handler*** or ***event listener***.

The web browser supports event-driven programming to simplify handling the many events a webpage must process. When an event happens, the browser calls the event's specified handlers. The web browser internally implements the code for detecting events and executing event handlers.

The example below modifies an input's `style` property, which sets the element's inline CSS styles. The input's border color changes colors when the input receives the focus or when focus is removed.

PARTICIPATION ACTIVITY 7.4.1: Focus and blur event handling.

```
<p>
   <label for="name">Name:</label>
   <input type="text" id="name">
</p>
<p>
   <label for="answer">Answer:</label>
   <input type="number" id="answer">
</p>
```




```
let inputs = document.querySelectorAll("input");

for (let i = 0; i < inputs.length; i++) {
   let input = inputs[i];
   input.style.border = "1px solid red";
   input.addEventListener("focus", function() {
      this.style.border = "1px solid green";
   });
```

```
    input.addEventListener("blur", function() {
       this.style.border = "1px solid blue";
    });
}
```

Animation content:

The following HTML is displayed:

```
<p>
  <label for="name">Name:</label>
  <input type="text" id="name">
</p>
<p>
  <label for="answer">Answer:</label>
  <input type="number" id="answer">
</p>
```

The following JavaScript is displayed:

```
let inputs = document.querySelectorAll("input");

for (let i = 0; i < inputs.length; i++) {
  let input = inputs[i];
  input.style.border = "1px solid red";
  input.addEventListener("focus", function() {
    this.style.border = "1px solid green";
  });
  input.addEventListener("blur", function() {
    this.style.border = "1px solid blue";
  });
}
```

The browser shows an input box labeled "Name: " and an input box labeled "Answer: ".

Step 1: The "Name: " input box is clicked. The following lines of code are highlighted:

```
 input.addEventListener("focus", function() {
```

```
    this.style.border = "1px solid green";
  });
```

The "Name: " input box is outlined in green.

Step 2: "Authur Dent" is entered into the "Name: " input box.

Step 3: The "Answer: " input box is clicked. The following lines of code are highlighted:

```
 input.addEventListener("blur", function() {
    this.style.border = "1px solid blue";
  });
```

The "Name:" input box is outlined in blue, and the following lines of code are highlighted.

```
input.addEventListener("focus", function() {
    this.style.border = "1px solid green";
  });
```

The "Answer: " input box is outlined in green.

Step 4: "42" is entered into the "Answer: " input box.

Step 5: The user clicks outside of both input boxes, and the following lines of code are highlighted.

```
 input.addEventListener("blur", function() {
    this.style.border = "1px solid blue";
  });
```

The "Answer: " input box is outlined in blue.

Animation captions:

1. User clicks in the Name input box. Browser calls the input element's focus event handler, which changes the element's style. Browser then gives focus to input box.
2. User key presses are sent to Name input box.
3. User clicks the Answer input box. Browser calls the Name element's blur event handler, then calls the Answer element's focus handler, and then gives focus to the Answer input box.

4. User key presses are sent to Answer input box.
5. When the user clicks elsewhere, the browser calls the Answer blur event handler.

PARTICIPATION ACTIVITY 7.4.2: Event-driven programming.

1) The actions a web browser notices are called event handlers.
   - True
   - False

2) A web developer must implement the code to detect events and call the appropriate handlers.
   - True
   - False

3) A mouse click causes an event.
   - True
   - False

Common events

Each event is given a name that represents the corresponding action. Ex: The event name for a mouse movement is `mousemove`, and the event name for a key down is `keydown`.

PARTICIPATION ACTIVITY 7.4.3: Mouse and keyboard events.

If unable to drag and drop, refresh the page.

mousemove **mouseover** **keyup** **click** **mouseout** **keydown**

	Caused by a mouse click.
	Caused by mouse entering the area defined by an HTML element.
	Caused by mouse exiting the area defined by an HTML element.
	Caused by mouse moving within the area defined by an HTML element.
	Caused by the user pushing down a key on the keyboard.
	Caused by the user releasing a key on the keyboard.

Reset

The following are events for which web developers commonly write handlers:

- A ***change*** event is caused by an element value being modified. Ex: Selecting an item in a radio button group causes a change event.
- An ***input*** event is caused when the value of an input or textarea element is changed.
- A ***load*** event is caused when the browser completes loading a resource and dependent resources. Usually load is used with the body element to execute code once all the webpage's CSS, JavaScript, images, etc. have finished loading.
- A ***DOMContentLoaded*** event is caused when the HTML file has been loaded and parsed, although other related resources such as CSS, JavaScript, and image files may not yet be loaded.
- A ***focus*** event is caused when an element becomes the current receiver of keyboard input. Ex: Clicking in an input field causes a focus event.
- A ***blur*** event is caused when an element loses focus and the element will no longer receive future keyboard input.

- A ***submit*** event is caused when the user submits a form to the web server.

PARTICIPATION ACTIVITY 7.4.4: Common browser events.

1) A submit event occurs when any button is clicked.
   - True
   - False

2) A blur event occurs when the mouse is moved over another input element.
   - True
   - False

3) The DOMContentLoaded event is likely to occur before the load event.
   - True
   - False

Registering event handlers

Handlers are written in three ways:

1. Embedding the handler as part of the HTML. Ex: `<button onclick="clickHandler()">Click Me</button>` sets the click event handler for the button element by using the `onclick` attribute. The attribute name used to register the handler adds the prefix "on" to the event name. Ex: The attribute for a mousemove event is `onmousemove`. Embedding a handler in HTML mixes content and functionality and thus should be avoided whenever possible.
2. Setting the DOM node event handler property directly using JavaScript. Ex: `document.querySelector("#myButton").onclick = clickHandler` sets the click event handler for the element with an id of "myButton" by overwriting the `onclick` JavaScript property. Using DOM node properties is better than embedding handlers within the HTML but has the disadvantage that setting the property only allows one handler for that element to be registered.

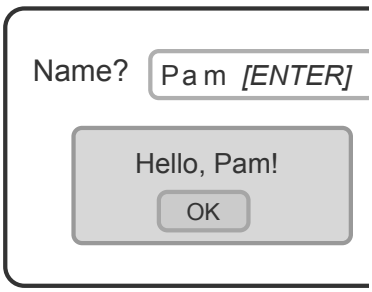
3. Using the JavaScript ***addEventListener()*** method to register an event handler for a DOM object. Ex: `document.querySelector("#myButton").addEventListener("click", clickHa` registers a click event handler for the element with the id "myButton". *Good practice is to use the `addEventListener()` method whenever possible, rather than using element attributes or overwriting JavaScript properties. The `addEventListener()` method allows for separation of content and functionality and allows multiple handlers to be registered with an element for the same event.*

Every handler has an optional ***event object*** parameter that provides details of the event. Ex: For a keyup event, the event object indicates which key was pressed and released, or for a click event, which element was clicked.

In the animation below, `keyupHandler()` uses `event.target` to access the text box object where the keyup event occurred. Inside an event handler, the `this` keyword refers to the element to which the handler is attached. So `event.target` and `this` both refer to the text box object in the event handler.

PARTICIPATION ACTIVITY 7.4.5: Registering event handlers with addEventListener().

```
<!DOCTYPE html>
<html>
  <title>Keyup Demo</title>
  <script>

 window.addEventListener("DOMContentLoaded", loadedHandler);

 function loadedHandler() {
    let textBox = document.querySelector("#name");
    textBox.addEventListener("keyup", keyupHandler);
 }

 function keyupHandler(event) {
    if (event.key == "Enter") {
       let textBox = event.target;
       alert("Hello, " + textBox.value + "!");
    }
 }
  </script>
  <body>
    <label for="id">Name?</label>
    <input type="text" id="name">
  </body>
</html>
```



Animation content:

The following HTML is displayed:

```
<!DOCTYPE html>
<html>
 <title>Keypress Demo</title>
 <script>

window.addEventListener("DOMContentLoaded", loadedHandler);

function loadedHandler() {
  let textBox = document.querySelector("#name");
  textBox.addEventListener("keypress", keyupHandler);
}

function keyupHandler(event) {
  if (event.key == "Enter") {
    let textBox = event.target;
    alert("Hello, " + textBox.value + "!");
  }
}
 </script>
 <body>
  <label for="id">Name?</label>
  <input type="text" id="name">
 </body>
</html>
```

An input box labeled "Name?" is displayed on the browser.

Step 1: The line of code reading "window.addEventListener("DOMContentLoaded", loadedHandler);" is highlighted.

Step 2: The line of code reading "function loadedHandler()" is highlighted.

Step 3: The line of code reading "textBox.addEventListener("keyup", keyupHandler);" is highlighted.

Step 4: "P" is entered in the "Name?" input box. The line of code reading "function keypressHandler(event)" is highlighted.

Step 5: The code 'if (event.key == "Enter")' is highlighted and evaluated to false.

Step 6: "Pam" is entered in the "Name?" input box, and then the user presses enter. The code 'if (event.key == "Enter")' is highlighted and evaluates to true.

Step 7: The following lines of code are highlighted.

```
let textBox = event.target;
alert("Hello, " + textBox.value + "!");
```

An alert dialog reading "Hello, Pam!", with a button reading "ok" is shown on the browser.

Animation captions:

1. The window's addEventListener() method registers the handler loadedHandler() for the DOMContentLoaded event.
2. After the rest of the HTML is loaded and parsed, the DOMContentLoaded event occurs, and loadedHandler() is called.
3. The text box's addEventListener() method registers the handler keyupHandler() for the keyup event.
4. When the user types the first letter, a keyup event occurs, which results in keyupHandler() being called.
5. The event.key is a string representing the pressed key ("P" for key P).
6. Each keyup causes keyupHandler() to execute. When the user presses Enter, event.key is "Enter", and the if statement is true.
7. event.target is the text box object that caused the keyup event. An alert dialog displays "Hello, Pam!"

PARTICIPATION ACTIVITY 7.4.6: Registering event handler using addEventListener().

The JavaScript code registers mouseover and mouseout event handlers for all elements that use the `highlight` class. Create and register a JavaScript event handler called

`myClickHandler()` to handle click events for the same elements. The `myClickHandler()` function should reveal the hidden text by changing the `style.color` of the `event.target` to `"black"`.

HTML | JavaScript | CSS

```
<body>
   <p>
      Challenge your knowledge about event-driven programming by gues
      each sentence below.  Click the missing text to reveal the answ
   </p>
   <ul>
      <li>
         Event handlers are also known as
         <span class="highlight hide">callback functions</span>,
         because the handlers are "called back" when the appropriate
      </li>
      <li>
         Event-driven programming allows webpages to react to
         <span class="highlight hide">user actions</span> and
         <span class="highlight hide">web server actions</span>.
      </li>
```

Render webpage Reset code

Your webpage

Challenge your knowledge about event-driven programming by guessing the text that is missing from each sentence below. Click the missing text to reveal the answer.

- Event handlers are also known as _____, because the handlers are "called back" when the appropriate event happens.
- Event-driven programming allows webpages to react to _____ and _____.

Expected webpage

Challenge your knowledge about event-driven programming by guessing the text that is missing from each sentence below. Click the missing text to reveal the answer.

- Event handlers are also known as _____, because the handlers are "called back" when the appropriate event happens.
- Event-driven programming allows webpages to react to _____ and _____.

▶ View solution

PARTICIPATION ACTIVITY 7.4.7: Registering event handlers.

Refer to the HTML below.

```
<body>
  <h1>Calculator</h1>
  <p>
    <input type="text" id="num1" size="5">
    <input type="text" id="num2" size="5">
    <span id="result"></span>
  </p>
  <input type="button" value="Add" id="addBtn">
</body>
```

1) What event registers `loadedHandler()` to be executed after the HTML has been loaded and parsed?

```
window.addEventListener("_____",
loadedHandler);
```

- click
- DOMContentLoaded
- ready

2) What is missing to register the `addNumbers()` function as a click event handler?

```
function loadedHandler() {
   let addBtn =
document.querySelector("#addBtn");
   addBtn.addEventListener("click",
_______);
}

function addNumbers() {
   let num1 = parseFloat(

document.querySelector("#num1").value);
   let num2 = parseFloat(

document.querySelector("#num2").value);

document.querySelector("#result").innerHTML
= num1 + num2;
}
```

- addNumbers()
- addNumbers(1, 2)
- addNumbers

3) What code registers an anonymous function as a click event handler for the add button?

```
function loadedHandler() {
   let addBtn =
document.querySelector("#addBtn");

addBtn.addEventListener("click",
_______);
}
```

- function addNumbers() { ... }
- function() { ... }
- function { ... }

4) The `highlightField()` function is an event handler for the mouseover and mouseout events. What parameter is `highlightField()` missing?

```
function
highlightField(_____) {
   if
(event.target.style.background
== "yellow") {

event.target.style.background
= "white";
   }
   else {

event.target.style.background
= "yellow";
   }
}
```

- event
- field
- color

5) What parameter is `highlightField()` missing to change the `textBox` background color to yellow?

```
textBox.addEventListener("mouseover",
highlightField);
function highlightField() {
   ____.style.background = "yellow";
}
```

- event.target
- event
- this

Capturing, at target, and bubbling phases

When an event occurs, the browser follows a simple DOM traversal process to determine which

handlers are relevant and need to be called. This traversal process follows three phases: capturing, at target, and bubbling.

1. In the ***event capturing*** phase, the browser traverses the DOM tree from the root to the event target node, at each node calling any event-specific handlers that were explicitly registered for activation during the capturing phase.
2. In the ***at target*** phase, the browser calls all event-specific handlers registered on the target node.
3. In the ***event bubbling*** phase, the browser traverses the DOM tree from the event target node back to the root node, at each node calling all event-specific handlers registered for the bubbling phase on the current node.

The optional third parameter for the `addEventListener()` method indicates whether the handler is registered for the capturing phase or bubbling phase. If the third parameter is `false` or not specified, or if the event handler is registered using any other mechanism, the browser registers the handler for the event bubbling phase. If the parameter is `true`, the browser registers the handler for the capturing phase.

Some events do not bubble, such as blur, focus, and change. When a non-bubbling event occurs, the browser will follow the event capturing phase, the at target phase, and then stop.

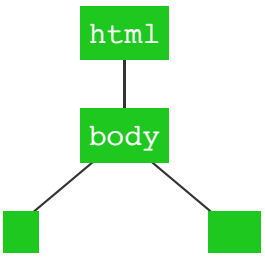
PARTICIPATION ACTIVITY 7.4.8: Capturing and bubbling.

```
<!DOCTYPE html>
<html>
<title>Meteors</title>
<script src="meteors.js" defer></script>
<body>
   <p>3 biggest meteor strikes on Earth:</p>
   <ol id="strikeList">
      <li>Vredefort Dome, South Africa</li>
      <li>Chicxulub Crater, Mexico</li>
      <li>Sudbury Basin, Ontario, Canada</li>
   </ol>
</body>
</html>
```

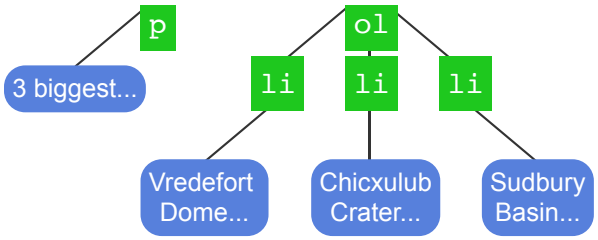
3 biggest meteor strikes on Earth:

1. Vredefort Dome, South Africa
2. Chicxulub Crater, Mexico
3. Sudbury Basin, Ontario, Canada

```
let list = document.getElementById("strikeList");

// Register handler for event bubbling phase
```



```
list.addEventListener("mouseover", function(e) {
   e.target.style.color = "red";
});

// Register handler for event capturing phase
list.addEventListener("mouseout", function(e) {
   e.target.style.color = "black";
}, true);
```



Animation content:

The following HTML is displayed:

<!DOCTYPE html>
<html>
<title>Meteors</title>
<script src="meteors.js" defer></script>
<body>
 3 biggest meteor strikes on Earth:
 <ol id="strikeList">
  Vredefort Dome, South Africa
  Chicxulub Crater, Mexico
  Sudbury Basin, Ontario, Canada

</body>
</html>

The following JavaScript is displayed:
list = document.getElementById("strikeList");

// Register handler for event bubbling phase
list.addEventListener("mouseover", function(e) {
  e.target.style.color = "red";
});

// Register handler for event capturing phase
list.addEventListener("mouseout", function(e) {
   e.target.style.color = "black";
 }, true);

The browser displays the following text.

3 biggest meteor strikes on Earth:
1. Vredefort Dome, South Africa
2. Chicxulub Crater, Mexico
3. Sudbury Basin, Ontario, Canada

A tree is shown. The head node is labeled "html", and is connected to a node labeled "body". The "body" node is connected to a node labeled "p" and a node labeled "ol". Attached to the "p" node is a text node labeled "3 biggest...". Attached to the "ol" node are three "li" nodes, each with their own children which contain text in the above list.

Step 1: The mouse hovers over the first list item.

Step 2: The DOM tree is traversed, and the "ol" node is highlighted.

Step 3: The "li" node corresponding to the first element in the list is highlighted.

Step 4: The "ol" node is highlighted once more. "phase: bubbling" is displayed beside the tree. The mouseover event handler is highlighted. The first item in the list turns red. The tree is traversed back up to the "html" node.

Step 5: The mouse moves off the first list item. "Phase: capturing" is displayed beside the tree. The "ol" node is highlighted, and then the mouseout event handler is highlighted. The first list item turns black.

Step 6: "phase: at target" is displayed next to the tree, and the li node corresponding to the first list item is highlighted.

Step 7: "phase: bubbling" is displayed next to the tree. The tree's nodes are traversed back up to the "html" node.

Animation captions:

1. The user moves the mouse cursor over the list's first item. A mouseover event occurs with the first li node as the target node.
2. Event capturing phase traverses the DOM tree from the root to the event target node. No capturing handlers are registered for the mouseover event.
3. At target phase looks for mouseover event handlers registered on the target node, but no

mouseover handlers are registered for the first li node.
4. Event bubbling traverses DOM tree from the event node back to the root node. The ol node's bubbling event handler is called and changes the item's text to red.
5. A mouseout event occurs targeting the first li node. Event capturing phase traverses the DOM tree from the root to event target node. The mouseout event handler turns the list item black.
6. The at target phase looks for relevant mouseout event handlers registered on the target node, but no mouseout handlers are registered for the first li node.
7. The event bubbling phase looks for any relevant mouseout event handlers by moving up the DOM tree, but no elements have mouseout handlers registered for the bubbling phase.

PARTICIPATION ACTIVITY 7.4.9: Capturing and bubbling.

Given the HTML and JavaScript below, match the order of alerts to the action performed by the user.

```
<div id="div1">
   <div id="div2">
      <div id="div3">
      </div>
   </div>
</div>
```

```
let div1 = document.getElementById("div1");
let div2 = document.getElementById("div2");
let div3 = document.getElementById("div3");

div1.addEventListener("click", function(){ alert("Capture 1"); }, true);
div2.addEventListener("click", function(){ alert("Capture 2"); }, true);
div3.addEventListener("click", function(){ alert("Capture 3"); }, true);

div1.addEventListener("click", function(){ alert("Bubble 1"); });
div3.addEventListener("click", function(){ alert("Bubble 3"); });
```

If unable to drag and drop, refresh the page.

Capture 1, Capture 2, Capture 3, Bubble 3, Bubble 1

Capture 1, Bubble 1

Capture 1, Capture 2, Bubble 1

	User clicks on div with `div1` id.
	User clicks on div with `div2` id.
	User clicks on div with `div3` id.

Reset

PARTICIPATION ACTIVITY 7.4.10: Bubbling and capturing.

1) The web browser performs the event capturing process before the bubbling process.
   - True
   - False

2) If a web developer creates a "default" handler for a DOM node high in the tree and a more specific handler for a DOM node lower in the tree, the web browser will run both handlers for an event.
   - True
   - False

3) Bubbling is the preferred process for the web browser to find appropriate handlers for an event.
   - True
   - False

Preventing default behavior

The event capturing and bubbling process can be stopped by calling the ***stopPropagation()*** method

on the event object provided to the handler. Once `stopPropagation()` is called, the browser stops the traversal process but still calls relevant registered handlers on the current node.

A web developer may want to prevent the browser from using a built-in handler for an event. Ex: Whenever a user clicks a form's submit button, the web browser sends the form data to the web server. The event object's ***preventDefault()*** method stops the web browser from performing the built-in handler. The built-in handlers that are often prevented are clicking elements, submitting forms, and moving the mouse into or out of an element.

The example below uses two event handlers for the password textbox:

1. `preventSpaces()` is a keydown event handler that listens for key presses. If the space key is pressed, `event.preventDefault()` stops the space from appearing in the textbox.
2. `checkPassword()` is an input event handler that is called when the password input changes. `checkPassword()` displays Weak, Stronger, Moderate, or Strong in the `<span>` tag depending on various criteria for the password.

Testing password strength.

Start typing a password. Verify the message to the right of the widget changes as the password is improved:

- abc - Weak
- abc1 - Stronger
- abc1D - Moderate
- abc1De - Strong

HTML JavaScript

```
1 <label for="password">Password:</label>
2 <input type="text" id="password">
3 <span id="strength"></span>
4
```

Render webpage Reset code

Your webpage

Password:

PARTICIPATION ACTIVITY 7.4.11: Preventing default behavior.

1) A web developer cannot prevent the web browser from performing built-in handlers.

   - True
   - False

2) If a web developer creates a "default" handler for a DOM node high in the tree and a more specific handler for a DOM node lower in the tree, `stopPropagation()` can be called in the more specific handler to keep the browser from calling the default handler.

   - True
   - False

3) In the example above, a user may press a space in the password textbox, but the space does not appear.

   - True
   - False

4) In the example above, `checkPassword()` prevents the built-in input handler from executing.

   - True
   - False

5) In the example above, the password "qwerty1" causes the webpage to display "Strong".

   - True
   - False

CHALLENGE ACTIVITY | 7.4.1: Event-driven programming.

550544.4142762.qx3zqy7

Start

Register the updateCount event handler to handle blur changes for the textarea tag. Note: The function counts the number of characters in the textarea.

HTML | JavaScript

```
1 <label for="yourName">Your name:</label>
2 <textarea id="yourName" cols="40" rows="3"></textarea><br>
3 <p id="stringLength">0</p>
4
```

1 2 3

Check Next

Exploring further:

- Event reference from MDN
- EventTarget.addEventListener() from MDN
- Event flow tutorial from Java2s

7.5 Timers

Timeouts

Some events are related to time instead of user actions. Ex: A website may wish to display an advertisement 10 seconds after the webpage loads or display inventory data that updates at regular intervals. A ***timer*** is a general name for techniques to execute JavaScript code after some amount of time has occurred.

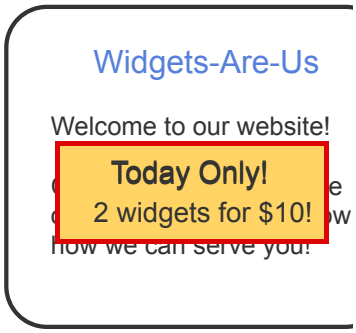
A web browser is able to execute a function after a time delay using `setTimeout()`. The ***setTimeout()*** method takes two arguments: a function and a time delay in milliseconds (1/1000th of a second). The browser calls the function after the time delay. `setTimeout()` returns a unique integer identifier that refers to the timeout that was created, and the timeout can be canceled by passing the identifier to ***clearTimeout()***.

PARTICIPATION ACTIVITY 7.5.1: Showing a daily special with setTimeout().

```
<div id="special">
  <h1>Today Only!</h1>
  <p>2 widgets for $10!</p>
</div>
```

```
#special {
   display: none;
   border: solid red 1px;
   ...
}
```

```
let timerId = setTimeout(showSpecial, 3000);

function showSpecial() {
   let special = document.getElementById("special");
   special.style.display = "block";
}
```



Animation content:

Step 1: The HTML is displayed:

```
<div id="special">
  <h1>Today Only!</h1>
  <p>2 widgets for $10!</p>
</div>
```

The JavaScript is displayed:

```
let timerId = setTimeout(showSpecial, 3000);

function showSpecial() {
   let special = document.getElementById("special");
   special.style.display = "block";
}
```

The CSS is displayed:

```
#special {
   display: none;
   border: solid red 1px;
   ...
}
```

The browser displays:

Widgets-Are-Us

Welcome to our website!

Check out our wide range
of widgets and let us know
how we can serve you!

Step 2: The following code is added.

```
let timerId = setTimeout(showSpecial, 3000);

function showSpecial() {
   let special = document.getElementById("special");
   special.style.display = "block";
```

```
}
```

The line of code reading "let timerId = setTimeout(showSpecial, 3000);" is highlighted.

Step 3: The following code is highlighted after a timer counts down from three seconds:

```
function showSpecial() {
  let special = document.getElementById("special");
  special.style.display = "block";
}
```

The following text appears over the current browser contents:

Today Only!
Two widgets for $10!

Animation captions:

1. A webpage contains a with a daily special that is not yet visible.
2. setTimeout() tells the browser to call showSpecial() in 3 seconds.
3. After 3 seconds, the browser calls showSpecial() and makes the visible.

PARTICIPATION ACTIVITY 7.5.2: Timeouts.

Refer to the animation above.

1) How many times is `showSpecial()` called when the webpage is loaded just once?

- 0
- 1
- Every 3 seconds

2) What modification to `setTimeout()` displays the daily special 5 seconds after the page loads?

- `setTimeout(showSpecial, 5)`
- `setTimeout(showSpecial(), 5000)`
- `setTimeout(showSpecial, 5000)`

3) Is the code below logically equivalent to the code in the animation?

```
let timerId = setTimeout(function()
{
   let special =
document.getElementById("special");
   special.style.display = "block";
}, 3000);
```

- Yes
- No

4) Suppose the code below is inserted immediately after the call to `setTimeout()`. What is different when the webpage loads?

```
clearTimeout(timerId);
```

- No change.
- The daily special never appears.
- The daily special appears immediately.

5) What is missing to hide the daily special after the special has displayed for 10 seconds?

```
function showSpecial() {
   let special =
document.getElementById("special");
   special.style.display = "block";
   ____;
}

function hideSpecial() {
   let special =
document.getElementById("special");
   special.style.display = "none";
}
```

- `setTimeout(hideSpecial, 10000)`
- `hideSpecial()`
- `clearTimeout(timerId)`

Intervals

A web browser is able to execute a function repeatedly with a time delay between calls using `setInterval()`. The ***setInterval()*** method takes two arguments: a function and a time interval in milliseconds (*t*). The browser calls the function every *t* milliseconds until the interval is canceled. The `setInterval()` method returns the interval's unique integer identifier, and the interval identifier can be passed to the ***clearInterval()*** method to cancel the interval.

PARTICIPATION ACTIVITY 7.5.3: Animating a ball with setInterval().

```
<img src="ball.png" id="ball" alt="ball"
   style="position:absolute; left:0">
```

```
let ballImage;
let timerId;

function startMoving() {
   ballImage = document.getElementById("ball");
```

```
    timerId = setInterval(moveBall, 10);
}

function moveBall() {
    let left = parseInt(ballImage.style.left);
    ballImage.style.left = left + 5 + "px";
}
```

Animation content:

Step 1: The following HTML is displayed:

<img src="ball.png" id="ball" alt="ball"
  style="position:absolute; left:0">

A ball is placed on the left side of the screen.

Step 2: The following JavaScript is displayed:

let ballImage;
let timerId;

function startMoving() {
  ballImage = document.getElementById("ball");
  timerId = setInterval(moveBall, 10);
}

function moveBall() {
  let left = parseInt(ballImage.style.left);
  ballImage.style.left = left + 5 + "px";
}

Step 3: The line of code reading "let left = parseInt(ballImage.style.left);" is highlighted, and left is assigned to integer value 0.

Step 4: The line of code reading "ballImage.style.left = left + 5 + "px";" is highlighted, and the ball is moved five pixels to the right.

Step 5: The moveBall() function is highlighted. The ball moves to the right and off of the screen.

Animation captions:

1. The HTML and CSS place a ball image on the left side of the browser screen.
2. When startMoving() is called, setInterval() creates an interval that calls moveBall() every 10 ms.
3. left is assigned the image's left CSS property, converted into an integer.
4. The image's left CSS property is assigned a px value 5 more than before, moving the ball 5 pixels to the right.
5. The interval calls moveBall() every 10 ms, animating the ball to the right. Eventually the ball is no longer on the screen.

PARTICIPATION ACTIVITY 7.5.4: Intervals.

Refer to the animation above.

1) The code below calls `moveBall()` less frequently than the animation does.

```
timerId =
setInterval(moveBall, 20);
```

- True
- False

2) How many times is `moveBall()` called to move the ball 100 pixels?

- 5
- 20

3) How long does the ball take to move 100 pixels?

- 1 second
- 200 ms

4) `moveBall()` is not called anymore after the ball moves off the screen.

- True
- False

5) What is missing to stop `moveBall()` from being called after the ball reaches the browser edge?

```
function moveBall() {
   let left =
parseInt(ballImage.style.left);
   if (left + ballImage.width >
document.body.clientWidth) {
      ____;
   } else {
      ballImage.style.left =
left + 2 + "px";
   }
}
```

- clearInterval(timerId)
- clearTimeout(timerId)

6) The code below moves the ball in the same direction as the animation above.

```
function moveBall(distance) {
   let left =
parseInt(ballImage.style.left);
   ballImage.style.left = left
+ distance + "px";
}

setInterval(function() {
   moveBall(-5);
}, 20);
```

- ○ True
- ○ False

PARTICIPATION ACTIVITY 7.5.5: Intervals.

Modify the JavaScript to create a countdown timer.

1. Add code to `startbutton`'s click event handler to start an interval that calls the `countdown()` function every second.
2. Store the unique identifier returned by `setInterval()` in `countdownTimerId` so the interval can be canceled.
3. Add code to `countdown()` to clear the countdown interval.
4. Add code to `stopbutton`'s click event handler to clear the countdown interval.

HTML JavaScript

```
<body>
   <p>Enter the countdown starting number, then click the start buttor
   <input type="number" id="number" min="0" value="5">
   <input type="button" id="startbutton" value="start">
   <input type="button" id="stopbutton" value="stop" disabled>
</body>

```

Render webpage Reset code

Your webpage

Enter the countdown starting number, then click the start button.

5 start stop

Expected webpage

Enter the countdown starting number, then click the start button.

5 start stop

▶ View solution

CHALLENGE ACTIVITY 7.5.1: Timers.

550544.4142762.qx3zqy7

Start

Write a setTimeout() function that reveals the answer after 2.5 seconds.

HTML JavaScript

```
1 <h1>Algebra question #1</h1>
2 <p>x<sup>2</sup> + 4x + 4 = 0</p>
3 <p class="answer" style="display: none;">Answer: x = -2</p>
```

1 2 3

Check Next

Exploring further:

- Scheduling: setTimeout and setInterval from javascript.info
- JavaScript Timing Events from w3schools

7.6 Modifying CSS with JavaScript

Modifying an element's inline style

JavaScript can manipulate a webpage's CSS to dynamically alter the looks of a webpage. Ex: JavaScript can change a background color when a button is clicked, or change the visibility of an error message when an input field is left blank. The ***CSS Object Model*** (***CSSOM***) is a set of APIs that allow JavaScript to manipulate CSS properties of a webpage.

Every element in the DOM has a ***style*** property that holds the inline styles set on the element. The `style` object implements the CSSOM interface ***CSSStyleDeclaration***, which provides methods for accessing, modifying, and removing CSS properties:

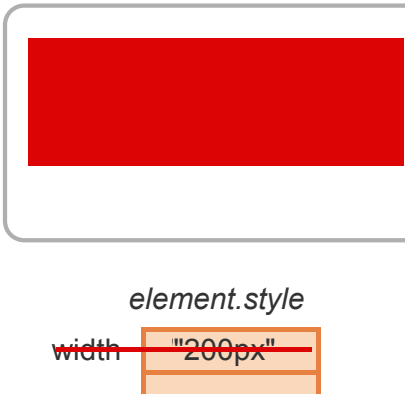
- The ***getPropertyValue()*** method returns the value of an element's CSS property or an empty string if the property is not set. Ex: `elem.style.getPropertyValue("color")` gets the element's color value.
- The ***setProperty()*** method sets the value of an element's CSS property. Ex: `elem.style.setProperty("color", "blue")` sets the element's color to blue.
- The ***removeProperty()*** method removes an element's CSS property. Ex: `elem.style.removeProperty("color")` removes the element's color property.

The `style` CSS properties can alternatively be accessed and modified using JavaScript property names. Ex: `elem.style.color = "blue"` is equivalent to `elem.style.setProperty("color", "blue")`. CSS property names that have dashes are converted into property names that use camel case. Ex: `background-color` becomes the JavaScript property `backgroundColor`.

PARTICIPATION ACTIVITY 7.6.1: Modifying inline CSS style.

```
<div id="mydiv" style="width: 100px; height: 100px;
                       background-color: green">
</div>
```

```
let myDiv = document.getElementById("mydiv");
let width = myDiv.style.getPropertyValue("width");
width = parseInt(width) + 100;
myDiv.style.setProperty("width", width + "px");
myDiv.style.setProperty("background-color", "red");
myDiv.style.removeProperty("width");
```



Animation content:

Step 1: The following HTML is displayed:

```
<div id="mydiv" style="width: 100px; height: 100px;
                background-color: green">
</div>
```

A green square is displayed on the browser.

Step 2: A list called element.style is created as follows.
width = 100 px
weight = 100 px
background-color = green

Step 3: The following JavaScript is displayed:

```
let myDiv = document.getElementById("mydiv");
let width = myDiv.style.getPropertyValue("width");
width = parseInt(width) + 100;
myDiv.style.setProperty("width", width + "px");
myDiv.style.setProperty("background-color", "red");
myDiv.style.removeProperty("width");
```

The line of code reading "let width = myDiv.style.getPropertyValue("width");" is highlighted and set to the value of "100px".

Step 4: The following lines of code are highlighted.

```
width = parseInt(width) + 100;
myDiv.style.setProperty("width", width + "px");
```

"width" is set to 200px, and the green square is expanded to be 200 pixels wide.

Step 5: The line of code reading "myDiv.style.setProperty("background-color", "red");" is

highlighted. "Background-color" is set to red, and the square changes color to red.

Step 6: The line of code reading "myDiv.style.removeProperty("width");" is highlighted. "width" is removed from elementary.style, and the red square becomes as wide as the webpage.

Animation captions:

1. The inline style properties make the a green 100px wide square.
2. The DOM maintains a list of the div's style properties and values.
3. The element.style.getPropertyValue() method returns the 100px width as a string.
4. 100px is added to width, and element.style.setProperty() sets the div's width to 200px. The browser automatically renders the div with the new width.
5. Changing the div's background color to red automatically renders the div red.
6. removeProperty() removes the width property, so the by default spans the entire browser width.

PARTICIPATION ACTIVITY 7.6.2: Modifying inline style.

Refer to the HTML and JavaScript below.

```
<span style="color: green">TEST</span>

<script>
let span = document.querySelector("span");
</script>
```

1) What does the code below output to the console?

```
console.log(span.style.getPropertyValue("color"));
```

- color
- green
- rgb(0, 255, 0)

2) What does the code below output to the console?

```
console.log(span.style.getPropertyValue("width"));
```

- An empty string
- The span's pixel width
- `false`

3) What is needed to set the span's background color?

```
span.____;
```

- `setProperty("background-color", "lightblue")`
- `style.setProperty(background-color, lightblue)`
- `style.setProperty("background-color", "lightblue")`

4) What is equivalent to the following code?

```
span.style.setProperty("font-family", "Arial");
```

- `span.style.font-family = "Arial";`
- `span.style.fontFamily = "Arial";`
- `span.style.fontfamily = "Arial";`

Modifying a stylesheet

The ***document.styleSheets*** object is a list of the stylesheets used in the webpage. Each stylesheet in `document.styleSheets` is a ***CSSStyleSheet*** object, which maintains a list of the stylesheet's CSS rules in the property called ***cssRules***. Two `CSSStyleSheet` methods allow CSS rules to be added or removed:

- The ***insertRule()*** method inserts a new rule into the stylesheet. Ex: `document.styleSheets[0].insertRule("p { color: blue; }")` inserts a new paragraph rule that makes the text color blue.
- The ***deleteRule()*** method deletes a rule at a given index number from the stylesheet. Ex: `document.styleSheets[0].deleteRule(0)` deletes the first CSS rule from the first stylesheet.

The CSS properties from a CSS rule are accessible from the rule's `style` property, which implements the `CSSStyleDeclaration` interface. So a rule's CSS properties can be accessed, modified, and removed with `getPropertyValue()`, `setProperty()`, and `removeProperty()`. Ex: `document.styleSheets[0].cssRules[0].style.setProperty("color", "blue")` sets the stylesheet's first rule's color to blue.

Security issue

For security reasons, some browsers like Chrome restrict the `cssRules` property from being accessed by JavaScript when the stylesheet is loaded off the computer's disk. The JavaScript and stylesheet must be downloaded from a web server for `cssRules` to be accessible.

PARTICIPATION ACTIVITY 7.6.3: Insert, modify, and delete CSS rules.

The webpage below displays a menu of food items with 3 buttons underneath:

1. Insert Rule button - Calls `insertRule()` to add a new paragraph rule that turns the menu items' font color blue.
2. Change Rule button - Calls `changeRule()` to change the paragraph rule's color to red.
3. Delete Rule button - Calls `deleteRule()` to delete the paragraph rule, which turns the font color back to green.

Click the three buttons in order to watch the font color change from green to blue, blue to red, and finally back to green.

Make the following modifications:

1. Add code to `insertRule()` that inserts the rule `.price { font-weight: bold; }` so the prices appear bold.
2. Add code to `changeRule()` that changes the `.price` rule to include the property `font-style` set to `italic` so the prices appear bold and italic.
3. Add code to `deleteRule()` that deletes the `.price` rule so the font weight and style returns to normal.

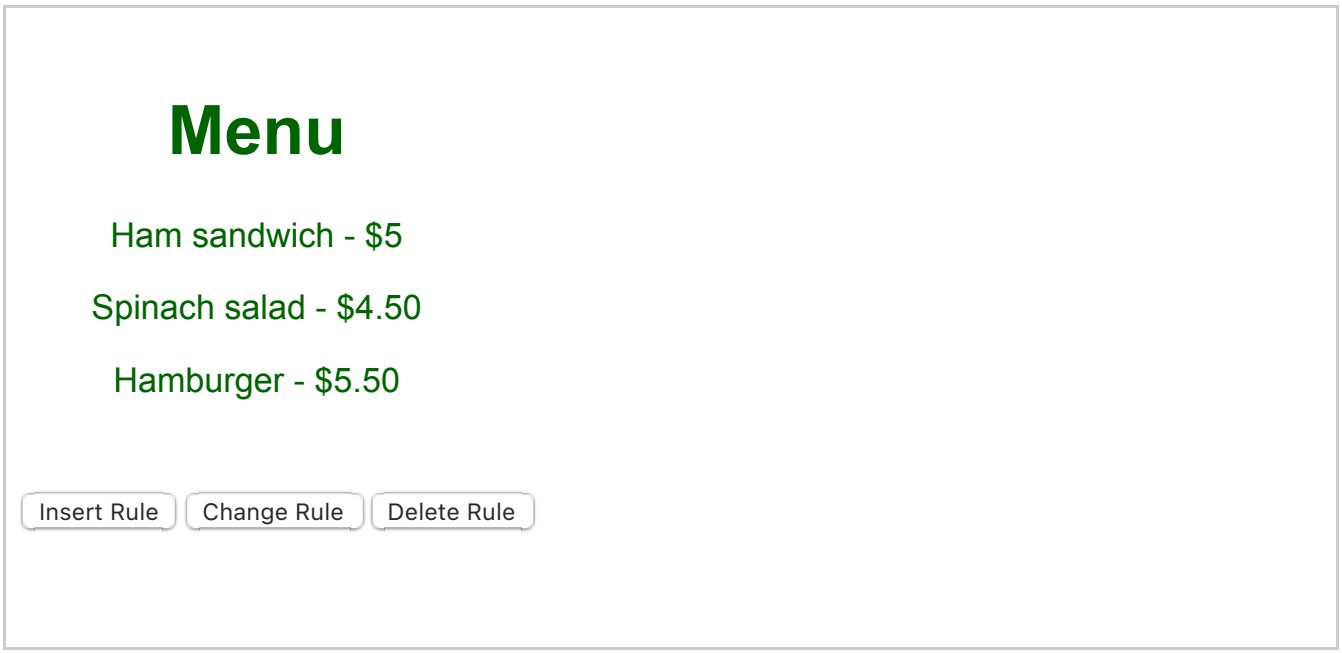
After making the modifications, click the 3 buttons in order to verify the price font changes as expected.

HTML | CSS | JavaScript

```
<body>
   <div id="menu">
      <h1>Menu</h1>
      <p>
         Ham sandwich - <span class="price">$5</span>
      </p>
      <p>
         Spinach salad - <span class="price">$4.50</span>
      </p>
      <p>
         Hamburger - <span class="price">$5.50</span>
      </p>
   </div>

   <p>
      <button id="insertRuleBtn">Insert Rule</button>
```

Render webpage Reset code

Your webpage



▶ View solution

PARTICIPATION ACTIVITY 7.6.4: Modifying stylesheet rules.

Refer to the HTML and CSS below.

```
<!DOCTYPE HTML>
<html lang="en">
   <head>
      <title>Funny Quotes</title>
      <link rel="stylesheet" href="styles.css">
   </head>
   <body>
      <blockquote
cite="https://www.brainyquote.com/quotes/tommy_cooper_189072">
         "I used to think I was indecisive, but now I'm not sure."
         - <span class="attribution">Tommy Cooper</span>
      </blockquote>
   </body>
</html>
```

```
/* styles.css */
body  {
   color: white;
   font-family: Arial, Helvetica, sans-serif;
}

blockquote {
   background-color: darkgreen;
   width: 200px;
   padding: 15px;
   border-radius: 5px;
}
```

1) What is the value of `document.styleSheets.length`?

- 0
- 1
- 2

2) If the webpage is modified with the HTML below, what is the value of `document.styleSheets.length`?

```
<link rel="stylesheet"
href="styles.css">
<style>
div { color: brown; }
</style>
```

- 0
- 1
- 2

3) What is the value of `document.styleSheets[0].cssRules.length`?

- 0
- 1
- 2

4) What is the font color of "Tommy Cooper" after the code below executes?

```
document.styleSheets[0].insertRule(".attribution
{ color: yellow; }");
```

- yellow
- white
- black

5) Assuming the first rule is `body`, what is the quote's font color after the code below executes?

```
document.styleSheets[0].deleteRule(0);
```

- yellow
- white
- black

6) Assuming the second rule is `blockquote`, what is the quote's font color after the code below executes?

```
let quoteRule =
document.styleSheets[0].cssRules[1];
quoteRule.style.setProperty("color",
"orange");
```

- orange
- white
- black

Adding and removing classes

Using the CSSOM to manipulate CSS properties and stylesheets is useful and sometimes necessary, but mixing CSS with JavaScript code blurs the separation between a web application's presentation and functionality. *Good practice is to declare CSS classes that perform the styling and use JavaScript to add and remove classes to and from DOM nodes as needed.*

Every DOM node has a ***classList*** property that lists the classes assigned to the node. Ex: The div node created from `<div class="account warning">` has a `classList` with items "account" and "warning". Methods exist to add and remove `classList` items:

- The ***add()*** method adds a class to the node's `classList`. Ex: `elem.classList.add("mystyle")` adds the class `mystyle` to the element's list of classes.
- The ***remove()*** method removes a class from the node's `classList`. Ex: `elem.classList.remove("mystyle")` removes the class `mystyle` from the element's list of classes.
- The ***toggle()*** method adds the class to the node's `classList` if the class is not present. If the class is already present, the class is removed. Ex: `elem.classList.toggle("mystyle")` toggles the class `mystyle` on or off.

A DOM node's class list can also be modified directly using the ***className*** property, which is a space-delimited list of the classes assigned to the node. Ex: `elem.className = "cat adopted"` assigns the `cat` and `adopted` classes to the element and removes any previously assigned classes from the node. All classes assigned to `className` are also added to the node's `classList`. Adding and removing properties with `classList` is often easier than using `className`.

PARTICIPATION ACTIVITY 7.6.5: Add and remove classes.

The webpage below asks the user to enter a strong password that meets 3 criteria. When the user clicks the Submit button, the `isStrongPassword()` is called with the password entered.

- If the password does not meet all 3 criteria, `isStrongPassword()` returns `false` and an error message is displayed by removing the `hidden` class from the error message.
- If the password meets all 3 criteria, `isStrongPassword()` returns `true` and the `hidden` class is added to the error message to hide the error message.

Enter some passwords that cause the error message to be visible and then hidden. Ex: Enter "abc" and press Submit to see the error message, then "abcdef1" to hide the error message.

Modify the `submitBtnClick()` function to do the following:

1. If `isStrongPassword()` returns `true`, then remove the `error-textbox` class from the password text box.
2. If `isStrongPassword()` returns `false`, then add the `error-textbox` class to the password text box.

After making the modifications, verify the password text box is highlighted in red only when entering an invalid password.

For an extra challenge, add the `error` class to the criteria that is violated when an invalid password is entered. Ex: If the password is not long enough, add the `error` class to the first `<li>` so the item becomes red.

HTML | CSS | JavaScript

```
<body>
   <p>Choose a strong password that meets the following criteria:
   </p>
   <ol>
      <li>At least 6 characters long.</li>
      <li>Contains at least 1 digit.</li>
      <li>Is not "password1".</li>
   </ol>
   <form>
      <label for="password">Password:</label>
      <input type="text" id="password">
      <span class="error hidden" id="errorMsg">Invalid password</spar
      <div>
         <input type="button" id="submitBtn" value="Submit">
      </div>
   </form>
```

Render webpage Reset code

Your webpage

Choose a strong password that meets the following criteria:

1. At least 6 characters long.
2. Contains at least 1 digit.
3. Is not "password1".

Password:
Submit

▶ View solution

PARTICIPATION ACTIVITY 7.6.6: Adding and removing classes.

Refer to the HTML, CSS, and JavaScript below:

```
<style>
   .important { background-color: yellow; }
   .complete { text-decoration: line-through; }
</style>
<body>
   <p>
     To-do list:
   </p>
   <ul>
      <li>Study for history exam</li>
      <li>Get groceries for dinner</li>
      <li>Volunteer at the children's center</li>
      <li>Vacuum and dust room</li>
   </ul>
</body>
```

```
// Add click callback to each <li>
const listItems = document.querySelectorAll("li");
for (let item of listItems) {
   item.addEventListener("click", listItemClick);
}

function listItemClick(e) {
   // Get clicked <li>
   let item = e.target;

}
```

Write the JavaScript code that is inserted into `listItemClick()` to perform the requested operation.

1) Add the `important` class to the clicked `<li>`.

Check Show answer

2) Remove the `complete` class from the clicked `<li>`.

Check Show answer

3) Toggle the `important` class on the clicked `<li>`.

Check Show answer

Exploring further:

- An Introduction and Guide to the CSS Object Model (CSSOM) from css-tricks.com
- Using dynamic styling information from MDN

- Element.classList from MDN
- Working with the new CSS Typed Object Model from Google

CHALLENGE ACTIVITY 7.6.1: Modifying CSS with JavaScript.

550544.4142762.qx3zqy7

Start

Complete the JavaScript code to set the paragraph's inline style to use the blue color and remove the paragraph's text transform property. **SHOW EXPECTED**

HTML JavaScript

```
1 <p id="helloMessage" style="color: gray; text-transform: uppercase">
2    Hello, Olympic contestants!
3 </p>
```

1 2 3

Check Next

7.7 Form validation

Validating data in the web browser

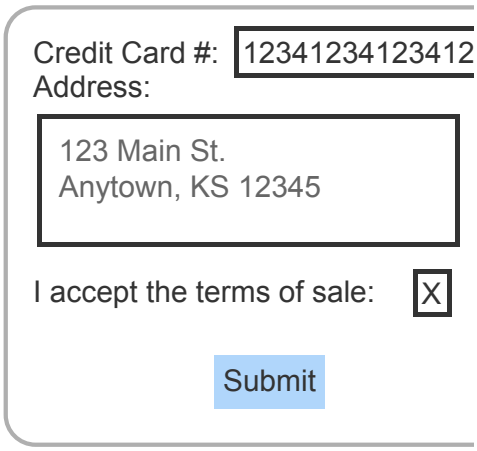
Since data integrity is essential to most applications, many web forms require specific formats for users to enter data. Ex: A credit card must contain 16 digits, a date cannot have a fifteenth month, and only 50 valid names of states exist for the United States of America.

Data validation is checking input for correctness. While a web server must perform data validation on submitted data, a better user experience occurs when the web browser performs the same data validation before submitting. Any invalid data in the webpage can be immediately flagged as needing modifications without waiting for the server to respond.

Data validation can either be performed while the user enters form data by adding a JavaScript function as the change handler for the appropriate field, or immediately prior to submitting the entire form by adding a function as the form's submit handler.

PARTICIPATION ACTIVITY 7.7.1: Validating form input.

```
<!DOCTYPE html>
<html>
<head>
   <title>Purchase Form</title>
   <script src="validate.js" defer></script>
</head>
<body>
   <form>
      <label for="cardNumber">Credit Card #:</label>
      <input type="text" name="cardNumber"
         id="cardNumber">
      <label for="address">Address:</label>
      <textarea name="address" id="address"></textarea>
      <label for="acceptTerms">I accept the terms of
         sale:</label>
      <input type="checkbox" name="acceptTerms"
         id="acceptTerms">
      <input type="submit" value="Submit">
   </form>
</body>
</html>
```



Animation content:

The following code is displayed.

```
<!DOCTYPE html>
<html>
<title>Purchase Form</title>
<script src="validate.js" defer></script>
<body>
  <form>
    <label for="cardNumber">Credit Card #:</label>
    <input type="text" name="cardNumber"
      id="cardNumber">
    <label for="address">Address:</label>
    <textarea name="address" id="address"></textarea>
    <label for="acceptTerms">I accept the terms of
      sale:</label>
    <input type="checkbox" name="acceptTerms"
      id="acceptTerms">
    <input type="submit" value="Submit">
  </form>
</body>
</html>
```

A web form contains an input box labeled "Credit Card #: " and "Address: ". A checkbox is labeled "I accept the terms of sale: " A button labeled "Submit" is at the bottom of the form.

Step 1: The line of code reading "<script src="validate.js" defer></script>" is highlighted.

Step 2: The user enters "12341234ABCD" into the "Credit Card #" field, and "123 Main St. Anytown, KS 12345" into the "Address" field, but does not submit the form.

Step 3: The user presses the "Submit" button, and the "Credit Card #" and "I accept the terms of sale" fields are highlighted in red.

Step 4: The user enters "1234123412341234" into the "Credit Card #" field, and checks the "I accept the terms of sale" field before pressing the "Submit" button. All data is registered as valid.

Animation captions:

1. The webpage uses JavaScript in validate.js to validate the web form.
2. The user enters invalid form data and does not check the checkbox.
3. When the user clicks the submit button, the browser executes code in validate.js to validate the form input and highlights invalid fields in red.
4. The user must correct the form data before the browser submits the form data to the web server. After the user clicks the submit button again, the browser updates the page to reflect that all data is valid.

PARTICIPATION ACTIVITY 7.7.2: Identify why the field is invalid.

1) Enter 5-digit ZIP code:

ZIP

- Some input characters are not digits.
- The input field is empty.
- The input is too long.

2) Enter 5-digit ZIP code:

103

- Some input characters are not digits.
- The input field is empty.
- The input length is incorrect.

3) Enter 5-digit ZIP code:

31M4N

- Some input characters are not digits.
- The input field is empty.
- The input length is incorrect.

Validating form input with JavaScript

Each textual input element in an HTML document has a ***value*** attribute that is associated with the user-entered text. The `value` attribute can be used to validate user-entered text by checking desired properties, such as:

- Checking for a specific length using the `length` property on the `value` attribute
- Checking if entered text is a specific value using `===`
- Checking if the text contains a specific value using the string `indexOf()` method on the `value` attribute
- Checking if the text is a number using `isNaN()`
- Checking that text matches a desired pattern using a regular expression and the string `match()` method

Drop-down menus also have a `value` attribute that is associated with the user-selected menu option.

Checkboxes and radio buttons have a ***checked*** attribute that is a boolean value indicating whether the user has chosen a particular checkbox or radio button. The checked attribute can be used to ensure an input element is either checked or unchecked before form submission. Ex: Agreeing to a website's terms of service.

PARTICIPATION ACTIVITY 7.7.3: Validating the sale price.

```
<input type="text" id="salePrice">
<span id="errorMsg"></span>
```

Sale price: 9 Must be between $10 and $1000.

```
let salePriceWidget = document.querySelector("#salePrice");
let errorMsg = document.querySelector("#errorMsg");

errorMsg.innerHTML = "";

if (salePriceWidget.value.length === 0) {
   errorMsg.innerHTML = "Sale price is missing.";
}
else if (isNaN(salePriceWidget.value)) {
   errorMsg.innerHTML = "Please enter a number.";
}
```

```
else {
   let salePrice = parseFloat(salePriceWidget.value);
   if (salePrice < 10 || salePrice > 1000) {
      errorMsg.innerHTML = "Must be between $10 and $1000.";
   }
}
```

Animation content:

HTML code:
<input type="text" id="salePrice">

JavaScript code:

```
let salePriceWidget = document.querySelector("#salePrice");
let errorMsg = document.querySelector("#errorMsg");

errorMsg.innerHTML = "";

if (salePriceWidget.value.length === 0) {
  errorMsg.innerHTML = "Sale price is missing.";
}
else if (isNaN(salePriceWidget.value)) {
  errorMsg.innerHTML = "Please enter a number.";
}
else {
  let salePrice = parseFloat(salePriceWidget.value);
  if (salePrice < 10 || salePrice > 1000) {
    errorMsg.innerHTML = "Must be between $10 and $1000.";
  }
}
```

Animation captions:

1. The user can enter a sale price into the input field. The span displays validation error messages.
2. The JavaScript code validates the entered price. If nothing is entered, the code displays an error message next to the text box.
3. If the input is "dog", the next if statement calls isNaN() to detect the invalid input and shows

an error message.

4. If the input is a number like 9, parseFloat() converts the "9" string into the number 9.
5. The if statement displays an error message when the price is not between $10 and $1000.

PARTICIPATION ACTIVITY 7.7.4: Using JavaScript to validate input fields.

1) What does the validation function return for `checkGrade("-5")`?

```
function checkGrade(grade) {
   return grade.length > 0 &&
!isNaN(grade);
}
```

- true
- false
- null

2) What does the validation function return for `checkGrade("95.3")`?

```
function checkGrade(grade) {
   return !isNaN(grade) &&
      grade >= 0 && grade <=
10;
}
```

- true
- false

3) What does the validation function return for `checkTemperature("-40")`?

```
function
checkTemperature(temp) {
   return temp.length > 0 &&
!isNaN(temp) &&
      temp >= 0 && temp <=
1000;
}
```

- ○ true
- ○ false

4) What does the validation function return for `checkTemperature("  ")`?

```
function
checkTemperature(temp) {
   return temp.length > 0 &&
!isNaN(temp) &&
      temp >= 0 && temp <=
1000;
}
```

- ○ true
- ○ false

Validating form data upon submission

Validating form data using JavaScript that executes when the user submits the form can be performed by:

1. Register a handler for the form's submit event that executes a validation function.
2. Within the validation function, inspect the form's input fields via the appropriate DOM elements and element attributes.
3. If the form is invalid, call the `preventDefault()` method on the event to cancel the form submission and prevent the form data from being sent to the server.

Figure 7.7.1: Ensuring a checkbox is selected before the form is submitted.

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Terms of Service</title>
      <script src="validate.js" defer></script>
   </head>
   <body>
      <form id="tosForm" action="https://example.com" target="_blank"
method="POST">
         <p>
            <label for="tos">I agree to the terms of service:</label>
            <input type="checkbox" id="tos">
         </p>
         <input type="submit">
      </form>
   </body>
</html>
```

```
// validate.js

function checkForm(event) {
   let tosWidget = document.querySelector("#tos");

   // Cancel form submission if tos not checked
   if (!tosWidget.checked) {
      event.preventDefault();
   }
}

let tosForm = document.querySelector("#tosForm");
tosForm.addEventListener("submit", checkForm);
```

PARTICIPATION ACTIVITY 7.7.5: Practice validating form prior to submission.

Complete the JavaScript `checkForm()` function so that `checkForm()` sets the input `style.backgroundColor` to `LightGreen` for each field that passes the validation check and sets the input field's `style.backgroundColor` to `Orange` if the validation fails.

Validation rules:

- The screen name field must not be empty.
- The ZIP code field must be of length 5.
- The TOS field must contain "yes".

Note that some browsers will override the light green color with another color if the user chooses an autofill option instead of typing a value.

HTML | CSS | JavaScript

```
<form id="tosForm" action="https://wp.zybooks.com/form-viewer.php" me
   <label for="screenName">Screen name:</label>
   <input type="text" id="screenName" name="screenName">
   <label for="zip">ZIP code:</label>
   <input type="text" id="zip" name="zip" placeholder="5-digit ZIP co
   <label for="tos">Type <strong>yes</strong> if you agree to the ter
   <input type="text" name="agreement" id="tos">
   <input type="submit" value="Submit">
</form>

```

Render webpage Reset code

Your webpage	Expected webpage
Screen name:	Screen name:
ZIP code:	ZIP code:
Type **yes** if you agree to the terms of service:	Type **yes** if you agree to the terms of service:
Submit	Submit

▶ View solution

Validating each field as data is entered

Alternatively, form data can be validated as the user enters data in the form by:

1. For each field that should be validated:
    a. Register an input event handler for the field.
    b. Create a global variable to track whether the field is currently valid. In most cases, this global variable should be initialized to false since the form typically starts with the field as invalid.
    c. Modify the global variable as appropriate within the field's event handler.
2. Register a submit event handler for the form that verifies the global variables for each field are true.
3. If one or more of the global variables are false, call the `preventDefault()` method on the submit event to prevent the form from submitting to the server.

The example below uses a regular expression to verify the user enters five digits for the ZIP code. Regular expressions are discussed in more detail elsewhere. The form does not submit unless the ZIP is valid.

Figure 7.7.2: Checking a ZIP code field as the user updates the field.

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Terms of Service</title>
      <script src="validate.js" defer></script>
   </head>
   <body>
      <form id="tosForm">
         <label for="zip">ZIP:</label>
         <input type="text" id="zip">
         <input type="submit">
      </form>
   </body>
</html>
```

```
// validate.js

let zipCodeValid = false;
let zipCodeWidget = document.querySelector("#zip");
zipCodeWidget.addEventListener("input", checkZipCode);

function checkZipCode() {
   let regex = /^\d\d\d\d\d$/;
   let zip = zipCodeWidget.value.trim();
   zipCodeValid = zip.match(regex);
}

let tosForm = document.querySelector("#tosForm");
tosForm.addEventListener("submit", checkForm);

function checkForm(event) {
   if (!zipCodeValid) {
      event.preventDefault();
   }
}
```

PARTICIPATION ACTIVITY 7.7.6: Practice validating form as data is entered.

The JavaScript code defines three boolean variables that must all be true for the form to submit: `screenNameValid`, `zipCodeValid`, and `tosAgreeValid`. Two input event handlers exist:

- The `checkScreenName()` function is called when the screen name changes and assigns `screenNameValid` with true when the field is not empty.
- The `checkZipCode()` function is called when the ZIP code changes and assigns `zipCodeValid` with true when the field contains five digits.

Both functions also make the input widget's background orange if the input is invalid.

Add a `checkTosAgree()` function that is called when the TOS agreement input changes. The function should assign `tosAgreeValid` with true when the user enters "yes". The function should also set the input widget's background to the default color if "yes" is typed and orange for all other input.

HTML | CSS | JavaScript

```
<form id="tosForm" action="https://wp.zybooks.com/form-viewer.php" me
   <label for="screenName">Screen name:</label>
   <input type="text" id="screenName" name="screenName">
   <label for="zip">ZIP code:</label>
   <input type="text" id="zip" name="zip" placeholder="5-digit ZIP co
   <label for="tos">Type <strong>yes</strong> if you agree to the ter
   <input type="text" name="agreement" id="tos">
   <input type="submit" id="validate" value="Submit">
</form>

```

Render webpage Reset code

Your webpage	Expected webpage
Screen name:	Screen name:
ZIP code:	ZIP code:
Type **yes** if you agree to the terms of service:	Type **yes** if you agree to the terms of service:
Submit	Submit

▶ View solution

Using HTML form validation

Some HTML form elements and attributes enable the browser to do form validation automatically, which reduces the need for JavaScript validation.

Note

A browser that does not support a particular HTML input element will transform an unsupported element into a text input, which then requires JavaScript to validate the form data.

Some customized HTML input elements can only contain valid values, such as date or color. Customized elements are automatically checked by the browser and/or filled in by a pop-up input picker in the browser, ensuring the submitted value matches a common specification.

Various element attributes allow the browser to do validation without using JavaScript:

- The ***required*** attribute indicates that the field must have a value (text or selection) prior to submitting the form.

- The ***max*** and ***min*** attributes indicate the maximum and minimum values respectively that can be entered in an input field with ranges, such as a date or number.
- The ***maxlength*** and ***minlength*** attributes indicate the maximum and minimum length of input allowed by an input field.
- The ***pattern*** attribute provides a regular expression that valid input must match.
- The ***title*** attribute can be used to provide a description of valid input when using the pattern attribute.

Figure 7.7.3: Using HTML form validation.

```
<form>
  <input type="range" name="age" min="5" max="120">
  <input type="checkbox" name="agree" required>
  <input type="password" name="password" minlength="10" maxlength="16">
  <input type="text" name="credit" pattern="^\d{16}$" title="exactly 16
digits">
  <input type="submit">
</form>
```

Several CSS pseudo-classes exist to style input and form elements:

- The ***:valid*** pseudo-class is active on an element when the element meets all the stated requirements in field attributes.
- The ***:invalid*** pseudo-class is active on an element when one or more of the attributes in the field are not fully met.
- The ***:required*** pseudo-class is active on an element if the element has the `required` attribute set.
- The ***:optional*** pseudo-class is active on an element if the element does not have the `required` attribute set.

PARTICIPATION ACTIVITY 7.7.7: Practice with CSS pseudo-classes.

The form below requires all three inputs to be supplied. A red rectangle appears around the form, and the rectangle turns green once all three inputs are given valid values. No JavaScript is used, only CSS pseudo-classes that are automatically applied by the browser.

Make the following modifications:

1. Add the following CSS rule to make each input's background red when the input contains invalid data:

```
input:invalid {
   background-color: #ffdddd;
}
```

2. Add the following CSS rule to make each input's background green when the input contains valid data:

```
input:valid {
   background-color: #ddffdd;
}
```

Render the webpage and verify each input is red until valid data is input, then the input turns green.

HTML CSS

```
<form action="https://wp.zybooks.com/form-viewer.php" target="_blank"
   <label for="dob">Date of birth:</label>
   <input type="date" name="dob" id="dob" required>

   <label for="creditCard">Credit card number:</label>
   <input type="text" name="creditCard" id="creditCard" maxlength="16
      title="Exactly 16 digits" required>

   <label for="emailAddr">Email address:</label>
   <input type="email" name="emailAddr" id="emailAddr" required>

   <input type="submit">
</form>

```

Render webpage Reset code

Your webpage	Expected webpage
Date of birth: / / Credit card number: Email address: Submit	Date of birth: / / Credit card number: Email address: Submit

▶ View solution

PARTICIPATION ACTIVITY 7.7.8: Form validation questions.

1) If all the fields in a form have been validated before submitting the form data to a server, does the server need to repeat the field validation?

   - ○ Yes
   - ○ No

2) Is validating input fields as the user fills in each field better than validating the entire form after all the form data has been entered?

   - ○ Yes
   - ○ No

3) If validation shows that a form input value is invalid, should the input value be reset to the initial value?

- Yes
- No

4) Can web developers do all form data validation using HTML input element attributes and not use JavaScript validation?

- Yes
- No

CHALLENGE ACTIVITY 7.7.1: Form validation.

550544.4142762.qx3zqy7

Start

In the validateForm() function, the if statement verifies the username is less than 11 characters long. Display "Username is invalid" in the console log if the username does not meet the requirement. Use the event method preventDefault() to avoid submitting the form when the input is invalid. **SHOW EXPECTED**

HTML | JavaScript

```
<form id="userForm" action="https://learn.zybooks.com" method="POST">
   <p>
      <label for="userName">Username:</label>
      <input type="text" id="userName">
   </p>
   <p>
      <label for="phoneNumber">Phone number:</label>
      <input type="text" id="phoneNumber">
   </p>
   <input type="submit" id="submitBtn">
</form>
```

1 2 3 4 5

Check Next

Exploring further:

- Form data validation from MDN

7.8 JavaScript Object Notation (JSON)

Introduction to JSON

Communicating data between the server and browser is a significant task for modern web applications. Initial attempts to do so included unstructured text documents and heavily structured XML documents, both of which required significant effort to convert to a usable format. ***JavaScript Object Notation***, or ***JSON***, is an efficient, structured format for data based on a subset of the JavaScript language. JSON (pronounced "Jason") is intended to be easily readable by humans and computers. Debugging communication that uses JSON is easy because humans can read JSON. Communication is efficient because computers can transmit and parse JSON quickly. As a result, JSON has rapidly become the dominant format of data transfer between web browsers and servers.

PARTICIPATION ACTIVITY 7.8.1: JSON basics.

1) JSON is only useful for JavaScript programs.
   - ○ True
   - ○ False

2) JSON is the only format for communicating between browser and server.
   - ○ True
   - ○ False

3) JSON is easy for humans to read and write.
   - ○ True
   - ○ False

JSON structure and values

JSON has six basic data types:

1. **String** - Unicode characters enclosed within double quotes (`"`). A few special characters must be escaped with a backslash (`\`). Ex: backslashes (`\\`), double quotes (`\"`), newlines (`\n`), and tabs (`\t`).
2. **Number** - Either an integer or decimal number. Ex: `42`, `3.141`, `-1.1e-5`.
3. **Object** - Unordered list of zero or more name/value pairs separated by commas and enclosed within braces (`{}`). A name in a JSON object must be a string in double quotes. A value can be any legal JSON value. Each name and value is separated by a colon. Ex: `{ "Name": "Joe", "Age": 35 }`
4. **Array** - Ordered list of zero or more JSON values separated by commas and enclosed within brackets (`[]`). Ex: `[]` and `[13,"blue"]`.
5. **Boolean** - Either `true` or `false`.
6. **null** - Represents "nothing".

A ***JSON value*** can be any of the above data types.

The JSON structure is defined recursively so that objects can contain arrays and arrays can contain objects to any arbitrary depth.

A common error when generating JSON programmatically is to include a trailing comma after the list of name/value pairs in a JSON object or after the list of JSON values in a JSON array. Ex: `[0, 1, 2,]`.

Figure 7.8.1: An example JSON data structure.

```
{
   "name": "John Doe",
   "vehicles": [
      {
         "make": "Ford",
         "model": "F-150",
         "color": "white"
      },
      {
         "make": "Toyota",
         "model": "Camry",
         "color": "red"
      }
   ],
   "married": false,
   "previous_customer":
true,
   "known_associates": [],
   "notes": null
}
```

The JSON structure above is an object with six name/value pairs:

1. `name` has the string value `John Doe`.
2. `vehicles` has an array value of two objects. Each object in the vehicles array has three name/value pairs: make, model, and color.
    1. The array's first object's `make` is the string `Ford`, `model` is the string `F-150`, and `color` is the string `white`.
    2. The array's second object's `make` is `Toyota`, `model` is `Camry`, and `color` is `red`.
3. `married` is `false`.
4. `previous_customer` is `true`.
5. `known_associates` is an empty array.
6. `notes` is `null`.

PARTICIPATION ACTIVITY 7.8.2: JSON data types.

Refer to the following JSON structure:

```
[
  { "name": "oreo",
    "type": "cookie",
    "flavors": ["chocolate", "vanilla"],
    "favorite": false,
    "created": 1912
  },
  { "name": "snickers",
    "type": "candy bar",
    "flavors": ["chocolate", "peanuts", "caramel", "nougat"],
    "favorite": true,
    "created": 1930
  },
  { "name": "malt",
    "type": "frozen dairy",
    "flavors": ["vanilla", "chocolate", "strawberry"],
    "favorite": false,
    "created": 1922
  }
]
```

1) What value type does the JSON structure create?

- array
- object
- string

2) How many objects does the JSON structure create?

- 1
- 3
- 4

3) What is the data type of `favorite`?

- array
- boolean
- string

4) What is the data type of `created`?

- number
- object
- string

Working with JSON

JavaScript provides a built-in ***JSON object*** that provides two methods for working with JSON:

1. The ***JSON.parse()*** method creates a JavaScript object from a string containing JSON. Ex: `JSON.parse('[1,"two",null]')` converts the string `'[1,"two",null]'` into the JavaScript array `[1,"two",null]`. Typically, `JSON.parse()` is used with data received from a server.
2. The ***JSON.stringify()*** method creates a string from a JavaScript object. Typically, `JSON.stringify()` is used with data sent to a server. `JSON.stringify()` creates a string representation of any passed object by either calling the object's `toJSON()` method if defined or recursively serializing all enumerable, non-function properties. Ex: `JSON.stringify(new Date('2020-08-06'))` converts the JavaScript Date object to the string `2020-08-06T00:00:00.000Z` by calling the Date object's `toJSON()` method.

Good practice is to use single quotes around JavaScript strings containing JSON notation so that the double quotes for strings and JSON object names do not need to be escaped. Ex: Use `'{"name":"Bob"}'` instead of `"{\"name\":\"Bob\"}"`.

PARTICIPATION ACTIVITY 7.8.3: JSON.parse and JSON.stringify example.

```
let bondStr = '{"name":"James","age":35}';
console.log(bondStr);

let bondObj = JSON.parse(bondStr);
console.log("Happy birthday, " + bondObj.name);

bondObj.age += 1;
bondStr = JSON.stringify(bondObj);
console.log(bondStr);
```

```
{"name":"James","age":35}
Happy birthday, James
{"name":"James","age":36}
```

Animation content:

Step 1: The following code is displayed.

```
let bondStr = '{"name":"James","age":35}';
console.log(bondStr);
```

The following text is shown in the console.

```
{"name":"James","age":35}
```

Step 2: The following code is added.

```
let bondObj = JSON.parse(bondStr);
console.log("Happy birthday, " + bondObj.name);
```

The following text is shown in the console.

```
Happy birthday, James
```

Step 3: The following code is added.

```
bondObj.age += 1;
bondStr = JSON.stringify(bondObj);
console.log(bondStr);
```

The following text is shown in the console.

```
{"name":"James","age":36}
```

Animation captions:

1. bondStr is a string representing a JSON object.
2. The JSON.parse() method parses the JSON string to create a JavaScript object. The JavaScript object's name property is then printed to the console.
3. bondObj's age property is incremented. The JSON.stringify() method then converts bondObj back to a JSON string.

PARTICIPATION ACTIVITY 7.8.4: Using JSON.parse() and JSON.stringify().

1) `JSON.parse(____)` produces an array of three values: 1, 7, and 19.

Check Show answer

2) `JSON.stringify(____)` returns the string `'{"a":true,"b":[],"c":null}'`.

Check Show answer

3) What does the following code display in the console?

```
console.log(JSON.stringify({date: new Date("2001-01-01")}));
```

Check Show answer

Extending and customizing JSON output

The `JSON.parse()` method's second parameter is an optional parameter for a reviver function. A ***reviver function*** is used to modify parsed values before being returned, and is helpful when a JSON string represents a data type not available in JSON. Ex: A reviver function can convert a string representing a date, `"2010-12-30"`, to a JavaScript Date object.

The `JSON.stringify()` method has two optional parameters: a replacer and a spacer. The

replacer enables customization of the generated string. If replacer is a function, `JSON.stringify()` will use the value returned by the function as the string representation. Ex: A replacer can convert a JavaScript type not directly supported in JSON to a string representation of that data type. If replacer is an array, `JSON.stringify()` will filter the returned value by converting only the properties listed in the replacer array. Ex: `JSON.stringify({a:1,b:2,c:3},["a","b"])` returns the string `'{"a":1,"b":2}'`.

The spacer controls the indentation spacing of output JSON string, which indicates the depth of values in the object. When the spacer parameter is specified and not an empty string, the output will also include newlines. Ex: `JSON.stringify({a:1,b:2}, null, "  ")` returns the string below because the spacer parameter is a string with two spaces.

```
'{
  "a": 1,
  "b": 2
}'
```

PARTICIPATION ACTIVITY 7.8.5: Reviver function for JSON.parse().

```
let data = { date:new Date("2010-10-10") };
console.log(data);

let json = JSON.stringify(data);
console.log(json);

console.log(JSON.parse(json));

console.log(JSON.parse(json, function(k,v) {
   if (k == "date") return new Date(v);
   return v;
}));
```

```
Object {date: Sat Oct 09 2010 20:00:00
GMT-0400 (EDT)}

{"date":"2010-10-10T00:00:00.000Z"}

Object {date: "2010-10-10T00:00:00.000Z"}

Object {date: Sat Oct 09 2010 20:00:00
GMT-0400 (EDT)}
```

Animation content:

Step 1: The following lines of code are added.

let data = { date:new Date("2010-10-10") };
console.log(data);

The following text is displayed in the console.

```
Object {date: Sat Oct 09 2010 20:00:00
GMT-0400 (EDT)}
```

Step 2: The following lines of code are added.

```
let json = JSON.stringify(data);
console.log(json);
```

The following text is displayed in the console.

```
{"date":"2010-10-10T00:00:00.000Z"}
```

Step 3: The following code is added.

```
console.log(JSON.parse(json));
```

The following text is displayed in the console.

```
Object {date: "2010-10-10T00:00:00.000Z"}
```

Step 4: The following code is added.

```
console.log(JSON.parse(json, function(k,v) {
  if (k == "date") return new Date(v);
  return v;
}));
```

The following text is displayed in the console.

```
Object {date: Sat Oct 09 2010 20:00:00
GMT-0400 (EDT)}
```

Animation captions:

1. The console displays the date property of the data JavaScript object to be a Date object.
2. JSON.stringify() converts the Date object to a string.
3. JSON.parse() converts the string in json to a JavaScript string.

4. By providing a reviver function, JSON.parse() converts the date string to a Date object.

PARTICIPATION ACTIVITY 7.8.6: Customizing JSON.parse and JSON.stringify.

1) Which optional parameter can convert the string representation of a date into a JavaScript Date object?

- replacer
- reviver
- spacer

2) What is the result of the following JSON.stringify() call?

```
JSON.stringify({a:"one",b:"two",c:"three"},
["a","c"])
```

- `'{"a":"one","c":"three"}'`
- `'{"a":"one","b":"two","c":"three"}'`
- `'{"b":"two"}'`

3) What is the result of the following JSON.stringify() call that uses two spaces for the space parameter?

```
JSON.stringify({a:{b:1,c:3}},
null, '  ')
```

- `'{"a":{"b":1,"c":3}}'`
- `'{ "a":{   "b":1,
"c":3 }}'`
-
```
'{
  "a": {
    "b": 1,
    "c": 3
  }
}'
```
-
```
'{
    "a": {
        "b": 1,
        "c": 3,
     }
}'
```

CHALLENGE ACTIVITY 7.8.1: JavaScript and JSON.

550544.4142762.qx3zqy7

Start

Assign objectData with a JSON object with properties: userName (a string), age (a number) and studentSiblings (an array of strings). Note: The content of the properties doesn't matter.

```
1 let objectData;
2 /* Your solution goes here */
3
```

1 2 3 4 5

Check Next

Exploring further:

- JSON from MDN

7.9 XMLHttpRequest (Ajax)

Ajax introduction

A normal HTTP request is triggered by clicking on a hyperlink or submitting a form, after which the browser may appear non-responsive while the browser waits for the server response. When the browser receives the response, the entire webpage is replaced with the HTML in the response. This delay may be undesirable for some web applications and may annoy users if the delay is long.

Ajax (***Asynchronous JavaScript and XML***) is a technique to asynchronously communicate with a server and update a webpage once the response is received, without reloading the whole webpage. An ***asynchronous request*** occurs when the web application sends a request to the server and continues running without waiting for the server response. Although the "x" in Ajax stands for "XML", Ajax is used to transmit plain text, HTML, XML, and JSON.

XMLHttpRequest is an object for communicating with web servers using Ajax. Using the XMLHttpRequest object allows web browsers to hide the communication latency and continue to provide a responsive user interface while waiting for a server response. The XMLHttpRequest object defines handlers for events that occur during the request/response cycle. Ex: A response arrives at the browser, an error occurs during a request, etc. Using event-driven programming, the web application can continue providing a responsive interface and does not need to wait for a response from the server. The web application later updates the page once the response is received.

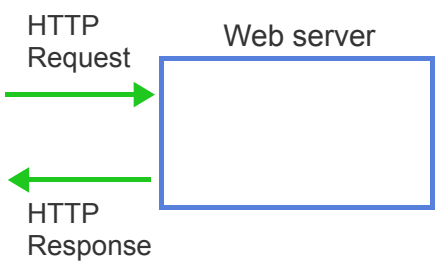
PARTICIPATION ACTIVITY 7.9.1: Asynchronous HTTP request.

```
<h1>Movie Information</h1>
<p id="movieinfo">

   <cite>Star Wars</cite>: Rated PG, released 1977

</p>
```

Movie Information

Star Wars: Rated PG, released 1977

```
let movieinfo = document.getElementById("movieinfo");
let xhr = new XMLHttpRequest();
xhr.addEventListener("load", function() {
   movieinfo.innerHTML = xhr.response;
});
xhr.open("GET", "starwars.html");
xhr.send();
```



Animation content:

Step 1: The following HTML is displayed:

```
<h3>Movie Information</h3>
<div id="movieinfo">
  <!-- to be loaded -->
</div>
```

The following JavaScript is displayed:

```
let movieinfo = document.getElementById("movieinfo");
let xhr = new XMLHttpRequest();
xhr.addEventListener("load", function() {
  movieinfo.innerHTML = xhr.response;
});
xhr.open("GET", "starwars.html");
xhr.send();
```

The browser displays "Movie Information" in a bold font.

Step 2: The JavaScript code is highlighted, and an HTTP request labeled "GET starwars.html" is sent to the webs server.

Step 3: The web server sends back an "HTTP response" containing starwars.html.

Step 4: The following lines of code are highlighted.

```
xhr.addEventListener("load", function() {
  movieinfo.innerHTML = xhr.response;
});
```

The line of code reading "<!-- to be loaded -->" is replaced with "<cite>Star Wars</cite>: Rated PG, released 1977".

Step 5: The web browser displays: Star Wars: Rated PG, released 1977

Animation captions:

1. The browser initially renders the webpage with no movie information.
2. The XMLHttpRequest object sends a GET request for starwars.html to the web server. The request is sent asynchronously.

3. The web server asynchronously responds to the browser with the contents of starwars.html.
4. The load event handler is called when starwars.html is loaded. The paragraph's inner HTML is replaced with the HTML contents of starwars.html.
5. The browser displays the Star Wars information.

Note

For security reasons, browsers limit Ajax requests to the web server from which the JavaScript was downloaded. Ex: JavaScript downloaded from http://instagram.com may only make Ajax requests to instagram.com. A ***cross-origin HTTP request*** *is a request made to another domain. Ex: An Ajax request from JavaScript downloaded from instagram.com to yahoo.com is a cross-origin HTTP request. Browsers can make cross-origin HTTP requests using a number of techniques including proxy servers, Cross-Origin Resource Sharing (CORS), and JSON with Padding (JSONP).*

Using XMLHttpRequest

The steps for using the XMLHttpRequest API are:

1. Create a new XMLHttpRequest object.
2. Assign handlers to the desired events via the `addEventListener()` method. The `addEventListener()` method takes two arguments: the event name and the event handler, code that should execute when the event occurs. If the handlers are not set up prior to calling the `open()` method, the progress events will not execute.
3. Initialize a connection to a remote resource using the `open()` method. The ***open()*** method takes two arguments: the HTTP request type and the URL for the resource. Most browsers only support "GET" and "POST" request types.
4. Modify the default HTTP request headers if needed with the ***setRequestHeader()*** method. Ex: `xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencode` sets the `Content-Type` header so a URL-encoded string may be sent in a POST request.
5. Send the HTTP request via the ***send()*** method. For POST requests, the data to be sent with the request is passed as the argument to the `send()` method.

PARTICIPATION ACTIVITY 7.9.2: Identify steps in making an Ajax request.

Match the JavaScript code with the description.

```
(a)  function responseReceivedHandler() {
       console.log("handling response: " + this.responseText);
     }

(b)  let xhr = new XMLHttpRequest();
(c)  xhr.addEventListener("load", responseReceivedHandler);
(d)  xhr.open("GET", "http://www.example.org/example.html");
(e)  xhr.send();
```

If unable to drag and drop, refresh the page.

Register load event handler **Create load event handler**

Initialize server connection **Create new XMLHttpRequest object**

Send HTTP request

	(a)
	(b)
	(c)
	(d)
	(e)

Reset

XMLHttpRequest result handlers

Good practice is to use a result handler for each specific result to separate functionality for each Ajax

event. Ex: Error handling, progress bars, updating the user interface on success, etc. The XMLHttpRequest result handlers are:

- The ***load*** handler is called when the exchange between the browser and server has completed. From the browser's perspective, the server received the request and responded. However, the request might not have been successful because of a problem such as a non-existent webpage. The HTTP status code must be examined to check which type of response was received. Ex: 200 vs. 404. The load, error, and abort handlers are mutually exclusive and are called after any progress handlers.
- The ***error*** handler is called when the browser does not receive an appropriate response to a request. Ex: The browser is unable to connect to the server, the connection between browser and server is cut in the middle of a response, etc.
- The ***abort*** handler is called when the browser is told to stop a request/response that is still in progress. Ex: The user closes the webpage that made the request.
- The ***timeout*** handler is called if the browser takes too much time to fully receive a response to a request. The timeout is an optional value that can be provided before the request is made. By default, the browser does not provide a timeout for a request.

Note

*The **readystatechange** handler relates to any change in the XMLHttpRequest. When XMLHttpRequest was originally defined, readystatechange was the only handler defined. As a result, many Ajax examples on the Internet only use readystatechange and do not include other handlers. Load, error, abort, and timeout are replacements for readystatechange.*

PARTICIPATION ACTIVITY 7.9.3: Match the event handlers to their descriptions.

If unable to drag and drop, refresh the page.

timeout **load** **error** **abort**

	Response received successfully.
	Sending request failed.
	Browser request was stopped.
	Request took too long to complete.

Reset

XMLHttpRequest progress handlers

XMLHttpRequest progress handlers are:

- The ***loadstart*** handler is called when the browser begins to send a request. The loadstart handler is called before any other XMLHttpRequest handler.
- The ***loadend*** handler is called after the browser receives the response. The loadend handler is called upon both response success and failure, and is called after all other XMLHttpRequest handlers.
- The ***progress*** handler is called one or more times while a response is being received by the client. Progress handlers are called before result handlers. The progress handler can be used to provide a data download progress indicator to the user. A similar handler is available to provide an indicator for uploaded data.

PARTICIPATION ACTIVITY 7.9.4: XMLHttpRequest event handler order.

Arrange the handlers in the order the handlers are called.

If unable to drag and drop, refresh the page.

progress loadend loadstart result handler

	First

	Second
	Third
	Fourth

Reset

Attributes for determining XMLHttpRequest success

The XMLHttpRequest object has attributes for checking the status of a response, which are usually used in the load handler and used to update the DOM.

- The ***status*** attribute is the numeric status code returned in the response.
- The ***statusText*** attribute is the descriptive text describing the status attribute.

Checking the `status` attribute of a response is important because the status code identifies the specific reason for a failure response. Ex: 403 means the requestor does not have permission to access the requested resource, and 404 means the requested resource was not found.

A common error is to assume that a failure response causes the error handler to be called. If the server properly sends the failure response to the browser, the browser will treat the response as successful and call the load handler. The error handler is only called if the response is not fully received by the browser.

Table 7.9.1: Common HTTP response status codes.

Status code	Meaning
200	HTTP request successful
3XX	General form for request redirection errors
301	Resource permanently moved, the new URL is provided
4XX	General form for client errors
400	Bad request. Ex: Incorrect request syntax
401	Unauthorized request. Ex: Not properly authenticated.
403	Request forbidden. Ex: User does not have necessary permissions.
404	Not found. Ex: Requested resource does not exist.
5XX	General form for server error codes
500	Internal server error. Ex: Server-side code crashed.
503	Service unavailable. Ex: Webpage is temporarily unavailable due to site maintenance.

Accessing Ajax response data

The XMLHttpRequest object provides multiple ways to access the response data.

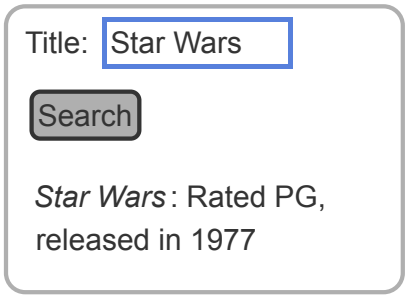
- The ***response*** attribute is the response body, which is parsed by the browser according to the `responseType` attribute.
- The ***responseText*** attribute is the plain text version of the response.
- The ***responseXML*** attribute is the XML DOM version of the response. The `responseXML` attribute is only available as a DOM object if the response is a valid and correctly formatted XML document.

The ***responseType*** attribute is set by the programmer to let the browser know the expected response data format.

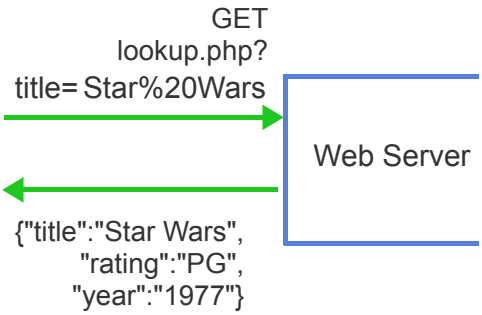
- If the `responseType` attribute is set to `"json"`, then the browser parses the entire response as a JSON object and sets the `response` attribute to the JSON object.
- If the `responseType` attribute is either `""` or `"text"`, the browser leaves the response unprocessed, and the `response` attribute contains the same value as `responseText`.
- If the `responseType` attribute is `"document"`, the browser assumes the response is an XML document, and the `response` attribute contains the same value as `responseXML`.

PARTICIPATION ACTIVITY 7.9.5: Creating a query string and loading JSON.

```
<body>
  <label for="title">Title:</label>
  <input type="text" id="title"><br>
  <button id="search">Search</button>
  <p id="movieinfo">
    <cite>Star Wars</cite>: Rated PG,
    released in 1977
  </p>
</body>
```




```
let searchBtn = document.getElementById("search");
searchBtn.addEventListener("click", function() {
   let xhr = new XMLHttpRequest();
   xhr.addEventListener("load", responseReceivedHandler);
   xhr.responseType = "json";
   let title = document.getElementById("title");
   let queryString = "title=" +
                     encodeURIComponent(title.value);
   xhr.open("GET", "lookup.php?" + queryString);
   xhr.send();
});

function responseReceivedHandler() {
   let movieInfo = document.getElementById("movieinfo");
   if (this.status === 200) {
      let movie = this.response;
      movieInfo.innerHTML = "<cite>" + movie.title +
                  "</cite>: Rated " + movie.rating +
                  ", released in " + movie.year;
   } else {
      movieInfo.innerHTML = "Movie data unavailable.";
   }
}
```



Animation content:

Step 1: The following HTML is displayed:

```
<body>
  <label for="title">Title:</label>
  <input type="text" id="title"><br>
  <button id="search">Search</button>
  <p id="movieinfo">
  </p>
</body>
```

The following JavaScript is displayed:

```
let searchBtn = document.getElementById("search");
searchBtn.addEventListener("click", function() {
   let xhr = new XMLHttpRequest();
   xhr.addEventListener("load", responseReceivedHandler);
   xhr.responseType = "json";
   let title = document.getElementById("title");
   let queryString = "title=" +
                     encodeURIComponent(title.value);
   xhr.open("GET", "lookup.php?" + queryString);
   xhr.send();
});

function responseReceivedHandler() {
   let movieInfo = document.getElementById("movieinfo");
   if (this.status === 200) {
      let movie = this.response;
      movieInfo.innerHTML = "<cite>" + movie.title +
                "</cite>: Rated " + movie.rating +
                ", released in " + movie.year;
   } else {
      movieInfo.innerHTML = "Movie data unavailable.";
   }
}
```

The user enters "Star Wars" into a textbox and presses the "Search" button.

Step 2: The line of code reading "xhr.responseType = "json";" is highlighted,

Step 3: The code encodeURIComponent(title.value) returns "Star%20Wars".

Step 4: The xhr.open() and xhr.send() code sends "GET lookup.php? title=Star%20Wars" to the web server.

Step 5: The web server responds with " {"title":"Star Wars", "rating":"PG", "year":"1977"}".

Step 6: The following lines of code are highlighted.

```
if (this.status === 200) {
    let movie = this.response;
```

Step 7: The following lines of code are highlighted.

```
 movieInfo.innerHTML = "<cite>" + movie.title +
                "</cite>: Rated " + movie.rating +
```

The paragraph is changed to: <cite>Star Wars</cite>: Rated PG, released in 1977

The browser displays "Star Wars: Rated PG, released in 1977".

Animation captions:

1. The user types the title "Star Wars" and presses the Search button, causing the Search button's click handler to execute.
2. xhr.responseType is set to "json" so that the JSON sent to the browser in the Ajax response will be automatically converted into a JavaScript object.
3. A query string is constructed using the text from the text box. encodeURIComponent() converts "Star Wars" into a string with no spaces.
4. An asynchronous HTTP request to lookup.php with a query string is sent to the web server.
5. Web server looks up "Star Wars" in a database and sends back a JSON response with information about the movie.
6. The load handler verifies the response's status code is 200 and accesses the movie object from this.response, which was created from the JSON response.
7. The movie information is placed in the paragraph, and the browser renders the HTML.

PARTICIPATION ACTIVITY 7.9.6: Ajax and JSON.

Refer to the animation above.

1) If the `responseType` attribute is set to "json", what attribute contains the parsed JSON object when the response is received?

   - `response`
   - `responseText`
   - `responseXML`

2) What query string is created for the Ajax request when the user enters *Pride & Prejudice*?

   - `title=Pride & Prejudice`
   - `title=Pride%20&%20Prejudice`
   - `title=Pride%20%26%20Prejudice`

3) What does the webpage display if lookup.php was accidentally misspelled lookup.html, and no lookup.html file exists?

   - Nothing.
   - The movie information for the given movie title.
   - "Movie data unavailable."

4) If lookup.php expects the movie title to be POSTed, the calls to `open()` and `send()` must be modified, and one more line of code must be added. What is the missing line of code?

```
xhr.open("POST", "lookup.php");
// Missing line
xhr.send(queryString);
```

- `xhr.responseType = "text";`
- `xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");`
- `xhr.addEventListener("error", errorHandler);`

PARTICIPATION ACTIVITY 7.9.7: Ajax practice.

The URL `https://wp.zybooks.com/weather.php?zip=XXXXX`, where XXXXX is a five digit ZIP code, returns JSON containing a randomly produced forecast for the given ZIP code. If the ZIP code is not given or is not five digits, the JSON response indicates the ZIP code is not found.

Successful request	Unsuccessful request
`{ "success": true, "forecast": [ { "high": 90, "low": 72, "desc": "sunny" }, { "high": 92, "low": 73, "desc": "mostly sunny" }, { "high": 87, "low": 64, "desc": "rain" }, { "high": 88, "low": 65, "desc": "cloudy" }, { "high": 90, "low": 68, "desc": "partly cloudy" } ] }`	`{ "success": false, "error": "ZIP code not found" }`

Enter any ZIP code in the webpage below, and press the Search button. When Search is pressed, an Ajax request is made to the URL above using the ZIP code entered in the form. The raw JSON response is displayed in the webpage, which is not ideal.

Make the following changes:

1. Modify `getForecast()` to specify that a JSON response is expected before calling `xhr.send()`:

```
xhr.responseType = "json";
```

2. Replace the code that appends the raw JSON to the `html` string with code that loops through the `forecast` array and produces a numbered list with each day's forecast:

```
//html += this.response;
html += "<ol>";
for (let day of this.response.forecast) {
   html += `<li>${day.desc}: high is ${day.high}, low is ${day.low}
</li>`;
}
html += "</ol>";
```

3. Render the webpage, and verify the changes you have made work correctly to show the weather for the ZIP code you enter.

4. Modify the code to display an appropriate error message if the Ajax response indicates the ZIP code is not found.

```
let html = "";
if (this.response.success) {
   html += "<h1>Forecast</h1>";
   html += "<ol>";
   for (let day of this.response.forecast) {
      html += `<li>${day.desc}: high is ${day.high}, low is
${day.low}</li>`;
   }
   html += "</ol>";
}
else {
   html = `<h1>Error: ${this.response.error}</h1>`;
}
```

5. Render the webpage, and verify that entering a bad ZIP code like "abc" produces an error message.

HTML JavaScript

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>Weather Forecast</title>
</head>
<body>
   <p>
      <label for="zip">ZIP code:</label>
      <input type="text" id="zip" maxlength="5">
      <button id="search">Search</button>
   </p>
   <div id="forecast"></div>
</body>
</html>

```

Render webpage Reset code

Your webpage

ZIP code: Search

Expected webpage

ZIP code: Search

▶ View solution

PARTICIPATION ACTIVITY 7.9.8: Updating the DOM via an Ajax request.

Match the JavaScript code with the description.

```
// Called when Ajax response is received
    function responseReceivedHandler() {
(a)     if (this.status === 200) {
            // code assumes response returns an ordered list that can be
inserted
            // into the list named myList, overwriting any previous
contents
            let list = document.getElementById("myList");
(b)         list.innerHTML = this.response;
        } else {
(c)         console.log("The request failed, status: " + this.status + " "
+ this.statusText);
        }
    }

    let xhr = new XMLHttpRequest();
    xhr.addEventListener("load", responseReceivedHandler);
(d) xhr.responseType = "text";
    xhr.open("GET", "http://www.example.org/example.html");
    xhr.send();
```

If unable to drag and drop, refresh the page.

Print debugging information **Check request success** **Access response's data**

Indicate how to interpret response data

	(a)
	(b)
	(c)
	(d)

Reset

CHALLENGE ACTIVITY 7.9.1: XMLHttpRequest (Ajax).

550544.4142762.qx3zqy7

Start

Store a new XMLHttpRequest object in the xhr variable, then assign responseHandler function as the "error" event listener.

```
1 function responseHandler() {
2    console.log("handling response: " + this.responseText);
3 }
4
5 /* Your solution goes here */
6
```

1 2 3 4

Check Next

Monitoring uploads

The XMLHttpRequest object's ***upload*** attribute is an object for monitoring the status of the request being sent to the server. The `upload` attribute has the same handlers as the XMLHttpRequest object, but the progress handler is the only handler typically used for the `upload` attribute. The progress handler can be used to monitor the status of uploading large files, such as attaching a document to a Gmail message.

Example 7.9.1: Monitoring the progress of an uploaded file.

```
function uploadProgressHandler(event) {
  if (event.lengthComputable) {
    console.log(event.loaded + " bytes uploaded out of " + event.total +
" bytes total.");
  }
}

let file = document.getElementById("file_widget").files[0];
let xhr = new XMLHttpRequest();
xhr.upload.addEventListener("progress", uploadProgressHandler);
xhr.open("POST", "http://www.example.org/example.html");
xhr.setRequestHeader("Content-Type", file.type);
xhr.send(file);
```

PARTICIPATION ACTIVITY 7.9.9: Uploading files using XMLHttpRequest.

1) `xhr.addEventListener("progress", handler)` tracks the status of a request to the server.
   - True
   - False

2) The XMLHttpRequest object can use a GET or POST request to upload files to a server.
   - True
   - False

3) When uploading files to a server, the Content-Type header indicates the type of file being uploaded.
   - True
   - False

4) The XMLHttpRequest object's load handler takes an event object as an argument.

- True
- False

5) The XMLHttpRequest object's progress handler takes an event object as an argument.

- True
- False

Exploring further:

- XMLHttpRequest from MDN
- HTTP access control (CORS) from MDN

7.10 Using third-party web APIs (JavaScript)

Introduction

Many organizations have created public web APIs that provide access to the organization's data or the user's data that is stored by the organization. Ex: The Google Maps API provides applications information about geographic locations, and the Instagram API allows applications access to photos shared on Instagram. Public APIs on GitHub.com lists thousands of free, public web APIs.

A ***third-party web API*** is a public web API used by a web application to access data provided by a third party. "Third-party" refers to a person or organization that is neither the web application using the API nor the user using the web application, which are the "first" and "second" parties. Websites rely on third-party web APIs to integrate with social media, obtain maps and weather data, or access collections of data.

To use a third-party web API, a developer usually registers with the third party to obtain an ***API key***.

Third parties require API keys for several reasons:

- The API key identifies who or what application is using the web API.
- The API key helps the third party limit the number of requests made to the API in a fixed time period or may be used to charge a developer a fee for additional requests.
- To obtain an API key, developers must agree to restrictions the third party places on data obtained from the web API.

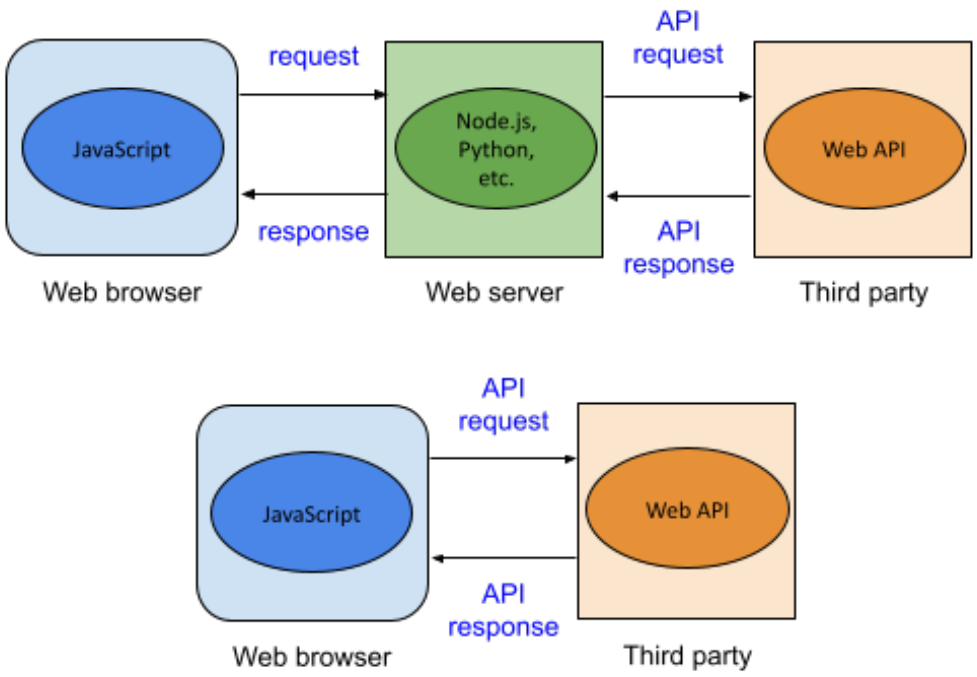
Most third-party web APIs are RESTful. A ***RESTful web API*** is a web API that is called with a URL that specifies API parameters and returns JSON or XML containing the API data. Ex: The URL http://linkedin.com/api/article?id=123 specifies the article ID 123, so the article would be returned formatted in JSON.

Third-party web APIs may be called from the web server or the web browser. This material shows how to call web APIs from the web browser using JavaScript.

SOAP

*A **SOAP-based web API** is another type of web API that relies heavily on XML and is in general more complex to use than RESTful web APIs. See the "Exploring further" section for more information on SOAP.*

Figure 7.10.1: Calling third-party web API from the web server or web browser.



PARTICIPATION ACTIVITY 7.10.1: Third-party web APIs.

1) Information from a third-party web API reaches the browser faster if the browser calls the web API directly instead of the web server calling the web API.

   - True
   - False

2) For a third-party web API requiring an API key, the API key must be transmitted with every API request.

   - True
   - False

3) When the browser makes an API request to a third-party web API, the web API key can be kept secret from prying eyes.

- True
- False

4) Many web APIs charge a fee to the developer after a limited number of requests have been made in a 24-hour period.

- True
- False

5) RESTful web APIs only return XML.

- True
- False

Weather API

OpenWeatherMap provides a free Weather API providing current weather data, forecasts, and historical data. Developers must register at openweathermap.org for an API key that must be transmitted in all API requests.

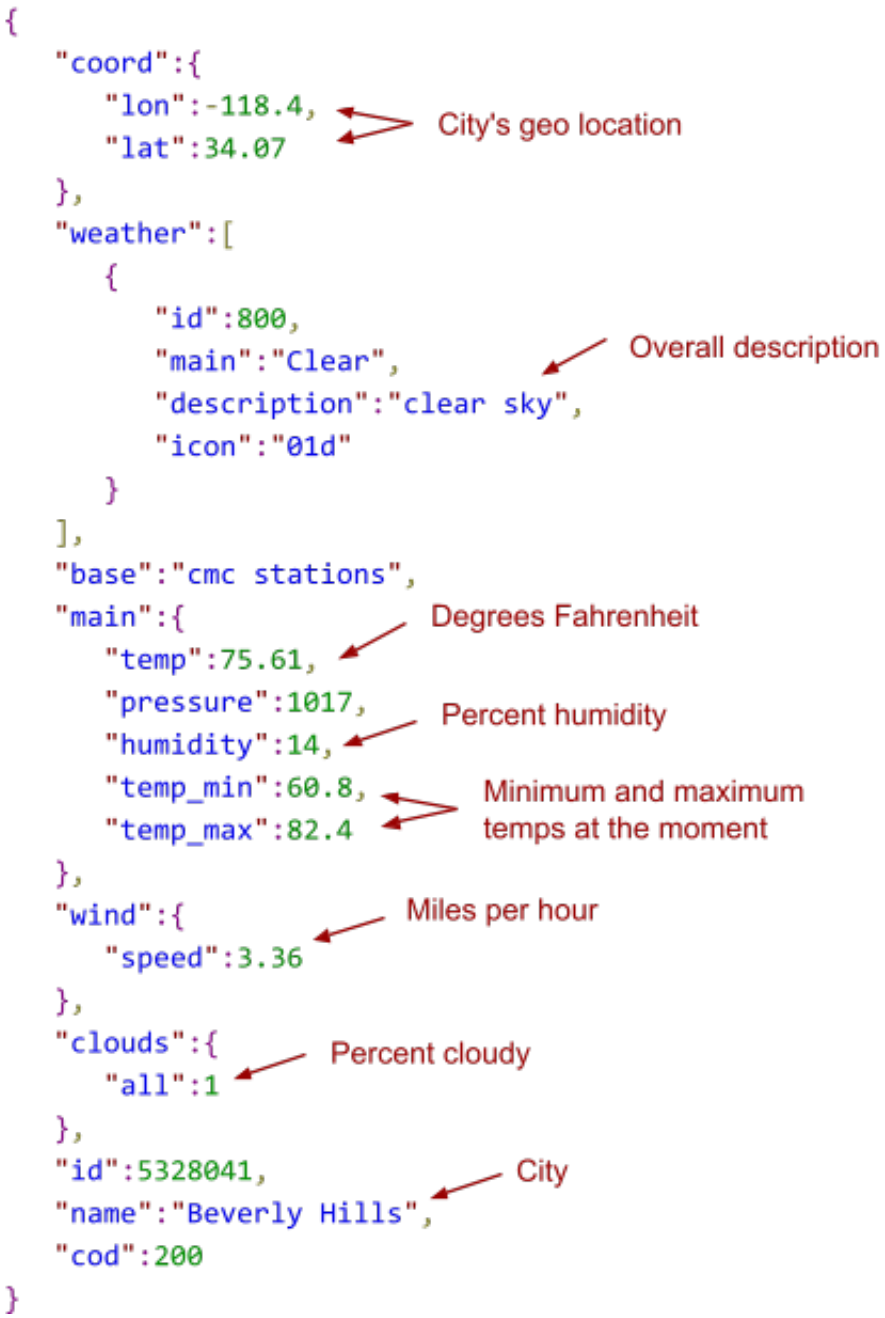
The OpenWeatherMap website provides documentation explaining how to use the Weather API using GET requests with various query string parameters. The API endpoint `http://api.openweathermap.org/data/2.5/weather` returns the current weather based on the following query string parameters:

- zip - Five digit US ZIP code
- units - Standard, metric, or imperial units to use for measurements like temperature and wind speed
- appid - Developer's API key

Other parameters are documented in the OpenWeatherMap website. The Weather API returns weather data in JSON format by default.

Figure 7.10.2: GET request to obtain the current weather for ZIP 90210.

http://api.openweathermap.org/data/2.5/weather?
zip=90210&**units**=imperial&**appid**=APIKEY



Try 7.10.1: Try OpenWeatherMap's API in your web browser.

1. Go to openweathermap.org.
2. Sign up for an account to obtain an API key.
3. When your API key is ready, try the link: http://api.openweathermap.org/data/2.5/weather?zip=90210&units=imperial&appid=APIKEY to make an API request for the weather with ZIP 90210. The page should indicate an invalid API key was used.
4. Replace APIKEY in the URL's query string with your API key, and reload the webpage. The JSON-encoded weather information for 90210 should be displayed.
5. Change the ZIP code in the URL's query string to your ZIP code, and reload the URL to see the weather in your ZIP code.

PARTICIPATION ACTIVITY 7.10.2: The Weather API.

1) What does the Weather API return when an invalid API key is used in a request?

- A blank webpage
- Weather for the 90210 ZIP
- An error message formatted in JSON

2) In the figure above, what does the Weather API return as the current humidity in the 90210 ZIP code?

- 75.61
- 14
- 3.36

3) What "units" parameter value would make the Weather API return the temperature in Celsius?

- imperial
- metric
- standard

4) Does the Weather API support finding the current weather by city name?

- Yes
- No

Cross-origin requests

Calling a third-party web API from the web browser requires a cross-origin HTTP request, since the web API is not hosted on the local website's web server. Two main techniques are used to make cross-origin requests:

- ***Cross-Origin Resource Sharing*** (***CORS***) is a W3C specification for how web browsers and web servers should communicate when making cross-origin requests.
- ***JSON with Padding*** (***JSONP***) is a technique to circumvent cross-origin restrictions by injecting `<script>` elements dynamically into a webpage. Script elements have no cross-origin restrictions.

CORS is the more common of the two techniques and, for the web API user, the easiest to use. CORS requires the web browser to send an `Origin` header in a web API request to indicate the scheme and domain making the API request. If the API accepts the request, the API responds with an `Access-Control-Allow-Origin` header indicating the same value in the `Origin` request header or "`*`", which indicates that requests are allowed from any origin. CORS uses other headers that begin with `Access-Control-*` to support other interactions with the API.

CORS allows the browser to send GET, POST, PUT, and DELETE requests. JSONP limits the browser to sending only GET requests.

Figure 7.10.3: Making a request to the Weather API with CORS.

HTTP request	HTTP response

`GET /data/2.5/weather?zip=90210&units=imperial&appid=APIKEY HTTP/1.1`
`Host: api.openweathermap.org`
`Origin: http://mywebsite.com`
`User-Agent: Mozilla/5.0 Chrome/48.0.2564`
	`HTTP/1.1 200 OK`
	`Access-Control-Allow-Origin: *`
	`Content-Type: application/json; charset=utf-8`
	`Content-Length: 431`
	`Date: Mon, 28 Mar 2016 16:09:48 GMT`
	`Server: openresty`
	
	`{"coord": {"lon":-118.4,"lat":34.07},"weather": [{"id":500, "main":"Rain","description":"light rain","icon":"10d"}], etc...}`

PARTICIPATION ACTIVITY 7.10.3: Cross-origin requests.

1) What HTTP header must the web browser send in every CORS request?
   - Access-Control-Allow-Origin
   - Origin
   - User-Agent

2) The web browser knows to send the `Origin` header in the HTTP request when the requested URL's domain name and the requesting script's domain name are _____.
   - the same
   - different

3) When a third-party web API does not support CORS, what is Access-Control-Allow-Origin set to in the web API's response?

- Access-Control-Allow-Origin is set to *.
- Access-Control-Allow-Origin is set to the Origin value.
- Access-Control-Allow-Origin is not present.

4) Does JSONP support POST or PUT request methods?

- Yes
- No

Calling the Weather API from JavaScript

The Weather API may be called from JavaScript using the `XMLHttpRequest` object, which makes asynchronous HTTP requests. The OpenWeatherMap implements CORS, and API requests can come from any origin.

The animation below shows how to retrieve weather information for a given ZIP code. For the JavaScript code to work in a web browser, the "APIKEY" string needs to be replaced with an actual API key.

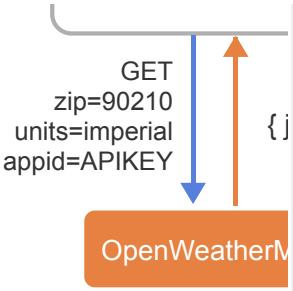
PARTICIPATION ACTIVITY 7.10.4: Calling the Weather API with JavaScript.

```
getWeather(90210);

function getWeather(zip) {
   let endpoint = "https://api.openweathermap.org/data/2.5/weather";
   let apiKey = "APIKEY";
   let queryString = "zip=" + zip + "&units=imperial&appid=" + apiKey;
   let url = endpoint + "?" + queryString;

   let xhr = new XMLHttpRequest();
```

Weather for 902

Current temp: 61 °F
Desc: clear sky
Humidity: 71%

```
    xhr.addEventListener("load", responseReceivedHandler);
    xhr.responseType = "json";
    xhr.open("GET", url);
    xhr.send();
}

function responseReceivedHandler() {
    let weatherInfo = document.getElementById("weather");
    if (this.status === 200) {
        weatherInfo.innerHTML =
         "<p>Current temp: " + this.response.main.temp + " &deg;F</p>" +
         "<p>Desc: " + this.response.weather[0].description + "</p>" +
         "<p>Humidity: " + this.response.main.humidity + "%</p>";
    } else {
        weatherInfo.innerHTML = "Weather data unavailable.";
    }
}
```



Animation content:

The following code is displayed.

```
getWeather(90210);

function getWeather(zip) {
  let endpoint = "https://api.openweathermap.org/data/2.5/weather";
  let apiKey = "APIKEY";
  let queryString = "zip=" + zip + "&units=imperial&appid=" + apiKey;
  let url = endpoint + "?" + queryString;

  let xhr = new XMLHttpRequest();
  xhr.addEventListener("load", responseReceivedHandler);
  xhr.responseType = "json";
  xhr.open("GET", url);
  xhr.send();
}

function responseReceivedHandler() {
  let weatherInfo = document.getElementById("weather");
  if (this.status === 200) {
    weatherInfo.innerHTML =
     "<p>Current temp: " + this.response.main.temp + " &deg;F</p>" +
     "<p>Desc: " + this.response.weather[0].description + "</p>" +
     "<p>Humidity: " + this.response.main.humidity + "%</p>";
```

```
  } else {
    weatherInfo.innerHTML = "Weather data unavailable.";
  }
}
```

"Weather for 90210" is displayed on the web browser.

Step 1: The following lines of code are highlighted.

```
 let endpoint = "https://api.openweathermap.org/data/2.5/weather";
  let apiKey = "APIKEY";
  let queryString = "zip=" + zip + "&units=imperial&appid=" + apiKey;
  let url = endpoint + "?" + queryString;
```

The website url is previewed as "https://api...weather?zip=90210...".

Step 2: The following lines of code are highlighted.

```
let xhr = new XMLHttpRequest();
  xhr.addEventListener("load", responseReceivedHandler);
  xhr.responseType = "json";
  xhr.open("GET", url);
  xhr.send();
```

"GET zip=90210 units=imperial appid=APIKEY" is sent to the Weather API.

Step 3: The Weather API responds with "{json}". The following code is highlighted.

```
function responseReceivedHandler() {
  let weatherInfo = document.getElementById("weather");
  if (this.status === 200) {
    weatherInfo.innerHTML =
     "<p>Current temp: " + this.response.main.temp + " &deg;F</p>" +
     "<p>Desc: " + this.response.weather[0].description + "</p>" +
     "<p>Humidity: " + this.response.main.humidity + "%</p>";
  } else {
    weatherInfo.innerHTML = "Weather data unavailable.";
  }
```

```
}
```

Step 4: The line of code reading "if (this.status === 200)" is highlighted.

Step 5: The following lines of code are highlighted.

```
weatherInfo.innerHTML =
    "<p>Current temp: " + this.response.main.temp + " &deg;F</p>" +
    "<p>Desc: " + this.response.weather[0].description + "</p>" +
    "<p>Humidity: " + this.response.main.humidity + "%</p>";
```

The web browser reads the following.
Weather for 90210
Current temp: 61 °F
Desc: clear sky
Humidity: 71%

Animation captions:

1. getWeather() creates a URL to request the current weather for the 90210 ZIP.
2. The XMLHttpRequest object sends a GET request to the Weather API.
3. OpenWeatherMap responds with JSON containing the current weather for ZIP code 90210. responseReceivedHandler() executes when the browser receives the JSON response.
4. this.status is 200 unless the ZIP code is not found.
5. Weather information is extracted from this.response and displayed in the webpage.

PARTICIPATION ACTIVITY 7.10.5: Calling the Weather API from JavaScript.

Refer to the animation above.

1) What JavaScript variable must be modified for the webpage to correctly access the Weather API?

- `endpoint`
- `apiKey`
- `queryString`

2) What JavaScript variable must be modified if the webpage is to display the temperature in Celsius instead of Fahrenheit?

- `endpoint`
- `apiKey`
- `queryString`

3) What is the expected output for the call below?

```
getWeather("test");
```

- "Weather data unavailable."
- Temperature for "test" ZIP code.
- Exception is thrown.

4) According to the figure above that shows the Weather API's JSON response, what variable in `responseReceivedHandler()` contains the wind speed?

- `this.status.speed`
- `this.response.speed`
- `this.response.wind.speed`

Exploring further:

- Public APIs on GitHub.com
- Understanding SOAP and REST Basics And Differences

7.11 Browser differences: JavaScript

Understanding browser differences

While W3C, WHATWG, and Ecma International define standards for HTML, CSS, and JavaScript, browsers may implement the standards differently for multiple reasons.

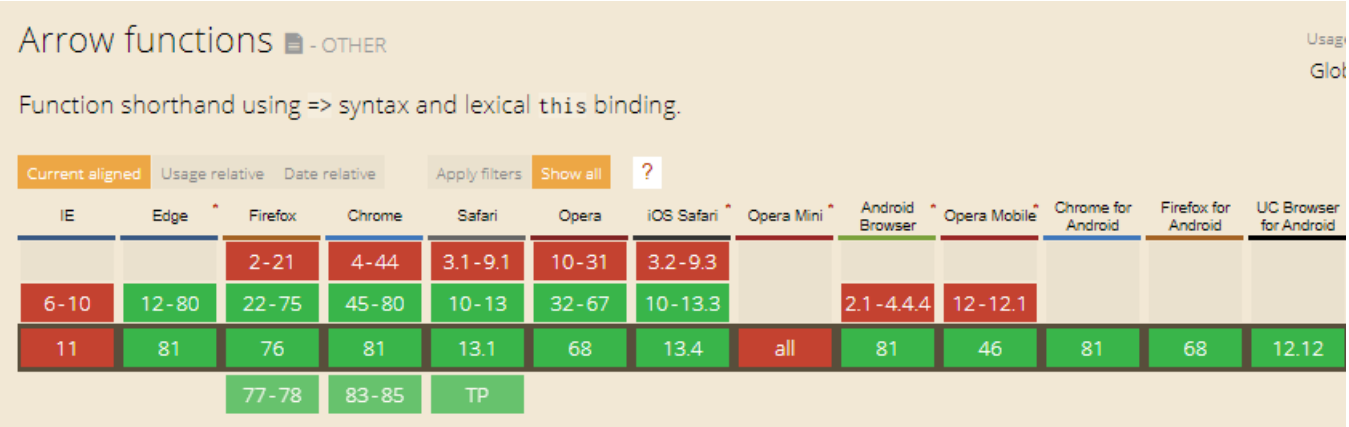
- A browser may have been released before a feature was standardized by the W3C or WHATWG, or the standard may have changed a particular specification. So, the browser vendor may not yet have added the new functionality.
- A browser vendor was helping WHATWG develop the standard API and may have started with a different proposed implementation than the final implementation. Typically, future versions of the browser will implement the final specification, but older versions may support the initial implementation.
- A browser vendor created a new API without input from WHATWG to differentiate the browser from other vendors' browsers. Browser differentiation was a prominent feature of the first browser wars, but non-standard browsers are now much less accepted by web developers and users. Browsers are expected to conform to standards.
- For open source browsers, the browser development team typically consists of volunteers who may not yet have been able to make the browser conform to the latest standards.
- A browser may have a bug causing the browser to interpret the code differently than the standard details.

In the early 2000's, web developers were expected to know the major differences between browsers, so that web applications would look the same in all browsers. Many articles were written to help developers, such as 7 JavaScript Differences Between Firefox and IE by Louis Lazaris. As developers began to expect standards conformance from browser vendors, online resources were created to help developers utilize standards.

CanIUse is a website that tracks which features are actively used on the web. The website also tracks which features are supported by all versions of the major browsers and displays those features in a matrix to show when those features were first supported. CanIUse uses statistics from various sources to indicate the percentage of web users accessing the web with each browser version. CanIUse also determines the percentage of web users with browsers capable of using a particular feature. After the percentage reaches some minimum threshold, a developer can

determine that the new feature is worth using because enough people will benefit and not too many will be "harmed."

Figure 7.11.1: CanIUse: Arrow functions.



Source: CanIUse

PARTICIPATION ACTIVITY 7.11.1: JavaScript support in browsers.

Guess the answer to the following questions to get a feel for JavaScript feature support in browsers. The degree of support is sometimes higher and sometimes lower than might be expected.

1) What percentage of the ECMAScript 5 standard (2009) does Internet Explorer 10 (2012) support?

- 48%
- 99%

2) What percentage of the ECMAScript 6 standard (2015) does Internet Explorer 10 (2012) support?

- 5%
- 60%

3) Which mobile operating system browser supported the highest percentage of ECMAScript 6 features in 2016, one year after ECMAScript 6 was standardized?

- Android 5.1
- iOS 9

Polyfills

A ***polyfill*** is JavaScript code that provides missing standard functionality for older browsers. A polyfill typically checks for the presence of a feature in the browser and uses the built in version if available. Otherwise, the polyfill executes JavaScript code that implements similar functionality. Common polyfill uses include form widgets, video and audio, geo-location, and WebGL.

Note

A polyfill may not behave the same as a native implementation by a browser, but the benefit of being able to use otherwise unsupported features generally outweighs slight differences.

Example 7.11.1: An HTML date polyfill.

If the date input element is not supported by a browser, the browser treats the date input box as an ordinary text input field.

```
<form>
  Select an appointment date:<br>
  <input type="date" name="appointmentDate"><br>
  <input type="submit">
</form>
```

Date input rendered by browser supporting (left) and not supporting (right) date input

A polyfill for supporting the date input can use Modernizr and the jQuery datepicker by:

1. Loading the jQuery and Modernizr libraries.
2. Calling a jQuery function that:
    - Uses Modernizr to determine whether the date input is supported
    - Uses the jQuery datepicker if needed
3. The original input form is unmodified by the polyfill.

```
<link href="https://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.12.0/jquery-ui.min.css" rel="stylesheet">
<script src="https://cdnjs.cloudflare.com/ajax/libs/modernizr/2.8.3/modernizr.min.js"></script>
<script src="https://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="https://code.jquery.com/ui/1.12.0/jquery-ui.min.js"></script>

<script>
$(function() {
   if (!Modernizr.inputtypes['date']) {
      $('input[type=date]').datepicker({
         dateFormat: 'mm-dd-yy'
      });
   }
});
</script>

<form>
   Select an appointment date:<br>
   <input type="date" name="appointmentDate"><br>
   <input type="submit">
</form>
```

Polyfill for date input using Modernizr and the jQuery date picker

Some browsers may implement JavaScript DOM objects differently, such as by not supporting a standard method for the object. Ex: Internet Explorer 8 does not support the Date.now() method. A JavaScript polyfill can be used to add unsupported methods to existing objects.

Example 7.11.2: Creating a polyfill for the Date.now() method.

To support the Date.now() method in an older browser, a web developer can add a now() method to the global Date object.

```
if (!Date.now) {
   Date.now = function() {
      return new Date().getTime();
   }
}
```

The code checks whether `Date.now` method is defined. If the now method is not defined, a now attribute is created in the Date object and set to reference a programmer defined function.

Alternatively, the Date.now() polyfill can be written more succinctly as:

```
Date.now = (Date.now || function() {
      return new Date().getTime();
});
```

`Date.now` is set to the result of evaluating `(Date.now || function() {...}`. If `Date.now` is defined and consequently truthy, then `Date.now` is set to the current value of `Date.now`. Otherwise, if the `Date.now` method is not defined and consequently falsy, then a now attribute is created in Date and set to reference the function defined to the right of the || operator.

PARTICIPATION ACTIVITY 7.11.2: Review browser JavaScript differences.

1) A missing JavaScript DOM object property is relatively easy to replace.
   - ○ True
   - ○ False

2) A polyfill is JavaScript code that provides missing standard functionality for older browsers.
   - ○ True
   - ○ False

3) Modernizr is a good polyfill for adding missing functionality to older browsers.

- ○ True
- ○ False

Exploring further:

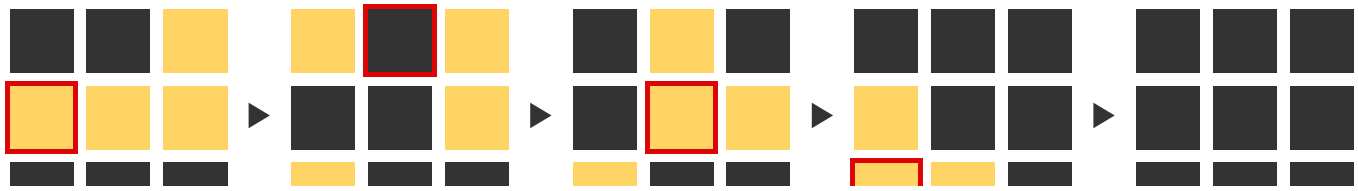
- Modernizr on GitHub
- Polyfills from MDN
- caniuse.com
- JavaScript compatibility table

7.12 Example: Lights Out game

Game overview

This section presents an example implementation of the Lights Out game. Lights Out has a grid of squares, each representing a light that is either on or off/out. The purpose of the game is to turn all the lights off by clicking on the squares. Each click toggles the light in the clicked square and each orthogonally adjacent square. The clicked square has up to four orthogonally adjacent squares: above, below, to the left, and to the right. The game begins with one or more lights on, and is won when all lights are turned off.

PARTICIPATION ACTIVITY 7.12.1: 3x3 Lights Out game example.



Animation content:

Step 1: A 3 by 3 grid is shown. The top right and entire middle row are filled in with yellow; the remaining boxes are filled in with black.

Step 2: The yellow box in the leftmost position of the middle row is clicked. The left column is now also completely filled in with yellow, and the clicked box is now black.

Step 3: The black box in the middle of the top row is clicked. This box is now yellow, and the rest of the top row is reset to black.

Step 4: The yellow box at the center of the grid is clicked. This box changes to black, and its diagonal neighbors swap colors. Now, the whole top row, two leftmost grids of the second row, and the leftmost grid of the last row are black. All other boxes are yellow.

Step 5: The first grid in the last row is clicked, and all of the boxes in the grid are now colored in black.

Animation captions:

1. A 3x3 Lights Out game begins with some lights on. Yellow lights are on and black lights off/out.
2. Clicking the light in the middle row and left column toggles the light and 3 orthogonally adjacent lights.
3. Clicking the light in the top row and middle column toggles the light and 3 orthogonally adjacent lights.
4. Clicking the center light toggles the light and 4 orthogonally adjacent lights.
5. Clicking the light in the bottom row and left column toggles the 3 remaining lights. All lights are out, and the game is won.

PARTICIPATION ACTIVITY 7.12.2: Lights Out game concepts.

1) The game is won when all lights are turned _____.
   - on
   - off

2) The upper-left square is not orthogonally adjacent to the _____ square in the _____ row.
   - center, top
   - leftmost, middle
   - center, middle

3) Each square has _____ orthogonally adjacent squares.
   - exactly 2
   - exactly 4
   - 2, 3, or 4

4) Clicking one square toggles _____ lights.
   - 2, 3, or 4
   - 3, 4, or 5

Page HTML

The LightsOut.html file has a `<div>` for the game grid. The grid is initially empty. JavaScript code will populate the `<div>` with either a 3x3 or 5x5 grid of buttons when a new game is started.

Four CSS classes are defined in the embedded style sheet. `grid3x3` and `grid5x5` are grid layout classes that are applied to the grid `<div>` when a new game starts. The `lightOn` and `lightOff` classes are applied to each grid button based on whether the light represented by the button is on or off.

Also included are a `<div>` to display a message when the game is won, a button to start a new game with a 3x3 grid of lights, and a button to start a new game with a 5x5 grid of lights.

The game's code will be implemented in LightsOut.js, which is referenced by a script tag.

Figure 7.12.1: LightsOut.html file.


```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Lights Out</title>
      <style>
         .grid3x3 {
            display: grid;
            grid-template-columns: 32px 32px 32px;
            grid-template-rows: 32px 32px 32px;
            grid-gap: 2px;
            margin-bottom: 16px;
         }
         .grid5x5 {
            display: grid;
            grid-template-columns: 32px 32px 32px 32px 32px;
            grid-template-rows: 32px 32px 32px 32px 32px;
            grid-gap: 2px;
            margin-bottom: 16px;
         }
         .lightOn {
            border: 1px solid black;
            background-color: yellow;
            width: 100%;
            height: 100%;
         }
         .lightOff {
            border: 1px solid black;
            background-color: black;
            width: 100%;
            height: 100%;
         }
      </style>
      <script src="LightsOut.js"></script>
   </head>
   <body>
      <div id="gameGrid"></div>
      <div id="information"></div>
      <input id="newGame3x3Button" type="button" value="New game
(3x3)">
      <input id="newGame5x5Button" type="button" value="New game
(5x5)">
   </body>
</html>
```

PARTICIPATION ACTIVITY 7.12.3: LightsOut.html file.

1) What is the ID of the element that will contain buttons representing the game's lights?
   - lightGrid
   - gameGrid
   - information

2) What is the ID of the button that is clicked to start a new 3x3 game?
   - newGame3x3Button
   - newGame5x5Button
   - New game (3x3)

3) An element that has `lightOn` assigned as the CSS class will have a _____ background color.
   - white
   - black
   - yellow

JavaScript game object

The game's state is stored in a single JavaScript object with the following properties:

- `rowCount` stores the number of rows in the game grid.
- `columnCount` stores the number of columns in the game grid.
- `lights` is an array of entries, one per light, stored in row-major order. `true` represents a light that is on, and `false` represents a light that is off.
- `startTime` is a `Date` object that stores the game's start time.

Figure 7.12.2: LightsOut.js excerpt: Game object declaration.

```
const currentGame = {
   "rowCount": 3,
   "columnCount": 3,
   "lights": [
      true, true, true,
      true, true, true,
      true, true, true
   ],
   "startTime": new
Date()
};
```

PARTICIPATION ACTIVITY 7.12.4: JavaScript game object.

1) For a 3x3 game, the first 3 entries in the game's `lights` array represent the first column of lights.
   - True
   - False

2) When every entry in the `lights` array is _____, the game is won.
   - true
   - false

3) `startTime` should be set to the current date/time when a new game begins.
   - True
   - False

Game logic: allLightsOut() and toggle() functions

The `allLightsOut()` function returns true if all entries in `game.lights` are false, indicating that all lights are out. The `toggle()` function takes numerical row and column arguments and

toggles the light at the corresponding location, along with each orthogonally adjacent light. Each function operates on a `game` object that is passed as the first argument. Neither function accesses any HTML elements on the page.

Figure 7.12.3: LightsOut.js excerpt: allLightsOut() and toggle() functions.

```
// Returns true if all lights in the game grid are off, false
otherwise.
function allLightsOut(game) {
   for (let i = 0; i < game.lights.length; i++) {
      // Even one light being on implies that not all are out/off
      if (game.lights[i]) {
         return false;
      }
   }

   // All lights were checked and none are on, so lights are out!
   return true;
}

// Toggles the light at (row, column) and each orthogonally adjacent
light
function toggle(game, row, column) {
   const locations = [
      [row, column], [row - 1, column], [row + 1, column],
      [row, column - 1], [row, column + 1]
   ];
   for (let location of locations) {
      row = location[0];
      column = location[1];
      if (row >= 0 && row < game.rowCount &&
         column >= 0 && column < game.columnCount) {
         // Compute array index
         const index = row * game.columnCount + column;

         // Toggle the light
         game.lights[index] = !game.lights[index];
      }
   }
}
```

PARTICIPATION ACTIVITY 7.12.5: allLightsOut() and toggle() functions.

If unable to drag and drop, refresh the page.

5 3 true false

	Returned by `allLightsOut()` if each entry in `game.lights` is set to `false`.
	If each entry in `game.lights` is _____, then the game is won.
	In a 3x3 game, calling `toggle(currentGame, 2, 2)` initializes the `locations` array with _____ entries.
	In a 3x3 game, calling `toggle(currentGame, 2, 2)` toggles _____ lights.

Reset

Game input and UI functions

An event listener for the `DOMContentLoaded` event is added at the start of the code. Five functions implement the remainder of the game:

- `checkForWin()`: Calls `allLightsOut()` to see if the game is won. If so, a message is displayed to the player that includes the time taken to win the game.
- `clickLight()`: Takes row and column parameters and handles a click at the specified location. Toggles lights, updates the grid buttons on the page, and checks to see if the game is won by calling `checkForWin()`.
- `createGameBoard()`: Populates the gameGrid `<div>` with a grid layout of buttons representing the game's lights. Calls `updateGridButtons()` after creating buttons and then clears the information `<div>`.
- `domLoaded()`: Called when the page's DOM content loads. Adds click event listeners for the buttons and starts a new 3x3 game.

- `newGame()`: Reinitializes the game object for either a 3x3 or 5x5 game, based on the `is5x5` parameter's value. The `game.lights` array is rebuilt, as well as the grid of buttons on the page. Also, `game.startTime` is reset to the current time.
- `updateGridButtons()`: Clears and rebuilds the page's grid of buttons based on the contents of `game.lights`. Each button in the grid is assigned a style class name of `lightOn` or `lightOff`.

Figure 7.12.4: LightsOut.js excerpt: input and UI functions.

```
// Add an event listener for the DOMContentLoaded event
window.addEventListener("DOMContentLoaded", domLoaded);

// Checks to see if the game is won. If so, a message is displayed in
the
// information <div> and true is returned. Otherwise false is returned.
function checkForWin(game) {
   if (allLightsOut(game)) {
      // Compute the time taken to solve the puzzle
      const now = new Date();
      const timeTaken = Math.floor((now - game.startTime) / 1000);

      // Display message
      const infoDIV = document.getElementById("information");
      infoDIV.innerHTML = "You win! Solved in " + timeTaken + "
seconds";

      return true; // game is won
   }

   return false; // game is not won
}

// Handles a click at the specified location. Toggles lights, updates
// the HTML grid on the page, and checks to see if the game is won.
function clickLight(game, row, column) {
   // Ignore if the game is already won
   if (allLightsOut(game)) {
      return;
   }

   // Toggle the appropriate lights
   toggle(game, row, column);

   // Update the HTML grid
   updateGridButtons(game);

   // Check to see if the game is won
   checkForWin(game);
}
```

```
}

// Creates the grid of buttons that represents the lights and clears the
// information <div>
function createGameBoard(game, is5x5) {
   // Get the grid <div> and clear existing content
   const gameGrid = document.getElementById("gameGrid");
   gameGrid.innerHTML = "";

   // Set the layout style based on game size
   gameGrid.className = is5x5 ? "grid5x5" : "grid3x3";

   // Create the grid of buttons
   for (let row = 0; row < game.rowCount; row++) {
      for (let column = 0; column < game.columnCount; column++) {
         // Create the button and append as a child to gameGrid
         const button = document.createElement("input");
         button.type = "button";
         gameGrid.appendChild(button);

         // Set the button's click event handler
         button.addEventListener("click", (e) => {
            clickLight(game, row, column);
         });
      }
   }

   // Update button styles from game.lights array
   updateGridButtons(game);

   // Clear the information <div>
   const infoDIV = document.getElementById("information");
   infoDIV.innerHTML = "";
}

// Called when the page's DOM content loads. Adds click event listeners
and
// starts a new 3x3 game.
function domLoaded() {
   // Add click event listeners for the two new game buttons
   const btn3x3 = document.getElementById("newGame3x3Button");
   btn3x3.addEventListener("click", function() {
      newGame(currentGame, false);
   });
   const btn5x5 = document.getElementById("newGame5x5Button");
   btn5x5.addEventListener("click", function() {
      newGame(currentGame, true);
   });

   // Start a new 3x3 game
   newGame(currentGame, false);
}

// Resets to a random, winnable game with at least 1 light on
```

```
function newGame(game, is5x5) {
   // Set the number of rows and columns
   if (is5x5) {
      game.rowCount = 5;
      game.columnCount = 5;
   }
   else {
      game.rowCount = 3;
      game.columnCount = 3;
   }

   // Allocate the light array, with all lights off
   const lightCount = game.rowCount * game.columnCount;
   game.lights = [];
   for (let i = 0; i < lightCount; i++) {
      game.lights.push(false);
   }

   // Perform a series of random toggles, which generates a game grid
   // that is guaranteed to be winnable
   while (allLightsOut(game)) {
      // Generate random lights
      for (let i = 0; i < 20; i++) {
         const randRow = Math.floor(Math.random() * game.rowCount);
         const randCol = Math.floor(Math.random() * game.columnCount);

         // Toggle at the location
         toggle(game, randRow, randCol);
      }
   }

   // Create the UI
   createGameBoard(game, is5x5);

   // Store the start time
   game.startTime = new Date();
}

// Updates the HTML grid's buttons based on game.lights
function updateGridButtons(game) {
   // Get the game grid <div>
   const gameGrid = document.getElementById("gameGrid");

   // Update grid buttons based on the game's light array entries
   for (let i = 0; i < game.lights.length; i++) {
      // Update the button's style based on the light state
      const button = gameGrid.children[i];
      button.className = game.lights[i] ? "lightOn" : "lightOff";
   }
}
```

PARTICIPATION ACTIVITY 7.12.6: Game input and UI functions.

1) After `newGame(currentGame, false)` is called, the length of `currentGame.lights` is ____.
   - 5
   - 9
   - 25

2) Which function checks to see if the game is won?
   - `updateGridButtons()`
   - `clickLight()`

3) Which function makes no alterations to game data or UI if the game is won?
   - `clickLight()`
   - `createGameBoard()`
   - `newGame()`

4) What must be true for `updateGridButtons()` to work properly?
   - The game must not be won.
   - The number of buttons in the gameGrid `<div>` must equal the length of `game.lights`.

5) The `domLoaded()` function adds a click event listener to _____ before calling `newGame()`.

- just the two new-game buttons
- just the buttons representing lights in the grid
- the buttons representing lights in the grid and the two new-game butttons

LightsOut.html and LightsOut.js full source.

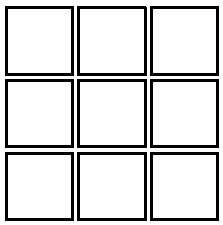
The full source of LightsOut.html and LightsOut.js is provided below, followed by the rendered version of the page. Some additional code is included, providing the option to watch the puzzle solve automatically by clicking the "Solve automatically" button. The solver simply applies the random toggles used to generate the game board in reverse order, so the solution is not likely to be optimal.

LightsOut.html | LightsOut.js

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Lights Out</title>
      <style>
         .grid3x3 {
            display: grid;
            grid-template-columns: 32px 32px 32px;
            grid-template-rows: 32px 32px 32px;
            grid-gap: 2px;
            margin-bottom: 16px;
         }
         .grid5x5 {
            display: grid;
            grid-template-columns: 32px 32px 32px 32px 32px;
```

Render webpage Reset code

Your webpage



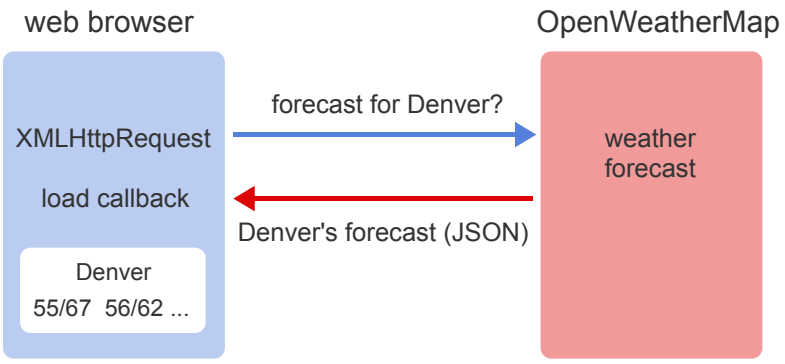
New game (3x3) New game (5x5) Solve automatically

7.13 Example: Weather Comparison (XMLHttpRequest)

Weather Comparison overview

This section presents an example implementation of a Weather Comparison web app that compares the weather forecast between two cities. The OpenWeatherMap's free 5 day weather forecast API is used to obtain weather forecasts using the `XMLHttpRequest` object.

PARTICIPATION ACTIVITY 7.13.1: Using the OpenWeatherMap web API to retrieve a weather forecast.



Animation content:

Step 1: The web browser is shown containing a "XMLHttpRequest". OpenWeatherMap is shown containing a "weather forecast".

The web browser sends a request labeled "forecast for Denver?" to OpenWeatherMap.

Step 2: OpenWeatherMap responds with "Denver's forecast (JSON)".

Step 3: The web browser loads the callback, and the forecast is previewed as follows.

Denver
55/67 56/62 ...

Animation captions:

1. The XMLHttpRequest object requests a city's forecast from the OpenWeatherMap's web API.
2. OpenWeatherMap responds with JSON that includes the city's 5 day forecast.
3. The XMLHttpRequest's load callback receives the JSON response and displays the forecast in the browser.

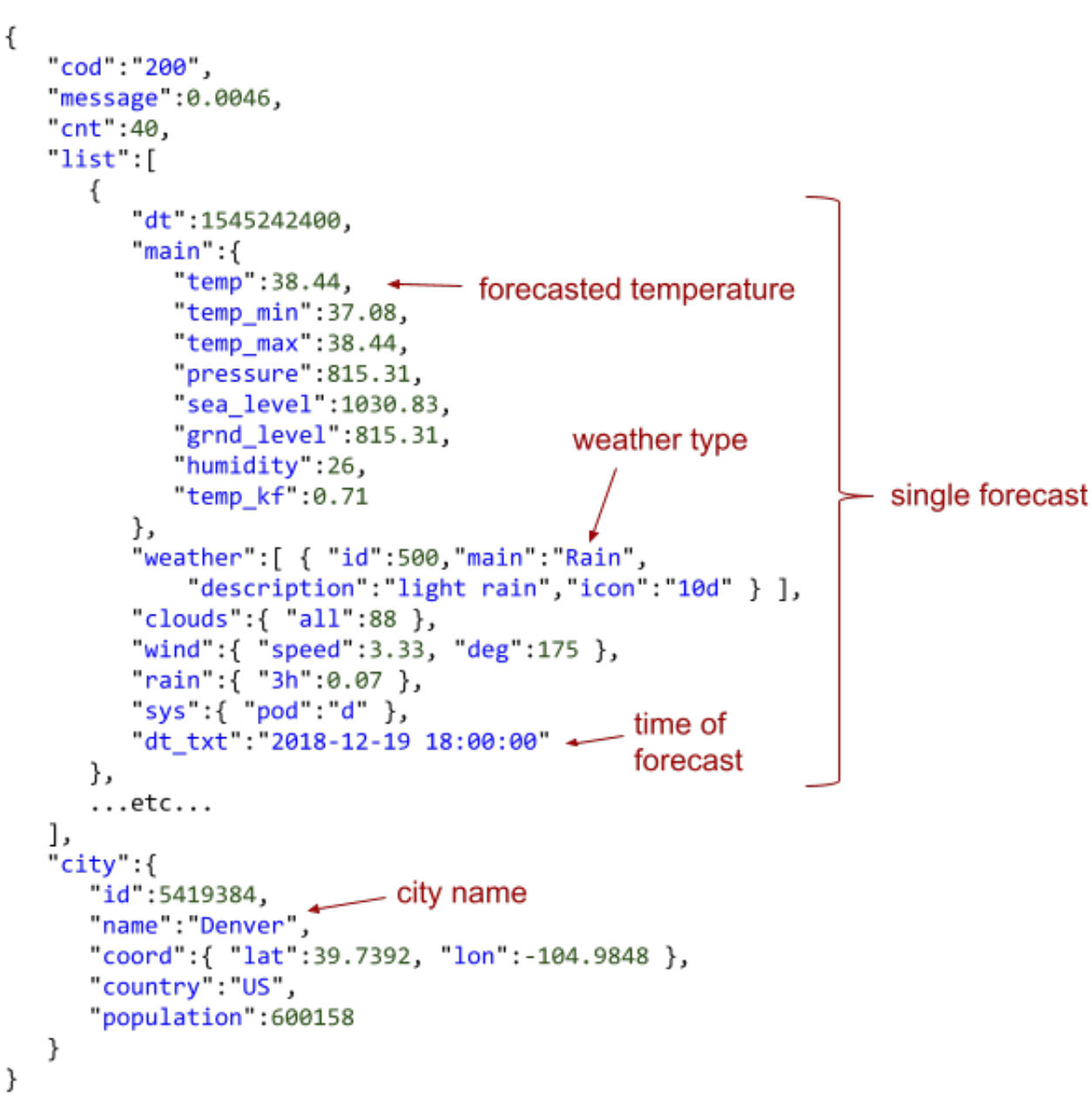
The OpenWeatherMap website provides documentation explaining how to use the forecast API using GET requests with various query string parameters. The API endpoint `https://api.openweathermap.org/data/2.5/forecast` returns the current weather based on the following query string parameters:

- q - City name
- units - Standard, metric, or imperial units to use for measurements like temperature and wind speed
- appid - Developer's API key

Other parameters are documented in the OpenWeatherMap website. The Weather API returns weather data in JSON format by default. The JSON response contains the weather forecast for 5 days with data every 3 hours. The figure below shows the first hour's forecast for Denver in the `list`.

Figure 7.13.1: URL and JSON response containing Denver forecast.

http://api.openweathermap.org/data/2.5/forecast?
q=Denver,US&**units**=imperial&**appid**=APIKEY



PARTICIPATION ACTIVITY 7.13.2: JSON forecast.

Refer to the figure above.

1) The forecast is for 6:00 pm on December 19, 2018.
   - True
   - False

2) The forecast is for a clear sky.
   - True
   - False

3) The forecast temperature is shown in Celsius.
   - True
   - False

4) The next item in the `list` should be 3 hours after the currently shown `list` item.
   - True
   - False

Page HTML and CSS

The HTML for the page has two text inputs and a Compare button so the user can enter the two cities to compare. The 5 day forecast will be displayed below in two tables. The webpage uses an external stylesheet styles.css for styling the page.

HTML and CSS for Weather Comparison app.

The app currently does nothing when Compare is pressed.

weather.html | styles.css

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Weather Comparison</title>
      <link rel="stylesheet" href="styles.css">
<!--   <script src="weather.js"></script>  -->

   </head>
   <body>
      <h1>Weather Comparison</h1>
      <section id="weather-input">
         <p>
            <label for="city1">City 1:</label>
            <input type="text" id="city1">
            <span class="error-msg hidden" id="error-value-city1">Enter a cit
```

Render webpage Reset code

Your webpage

Weather Comparison

City 1:

City 2:

Compare

PARTICIPATION ACTIVITY 7.13.3: HTML and CSS for Weather Comparison app.

1) What CSS class is hiding the "Enter a city" messages and the forecast section?

   - `error-msg`
   - `hidden`
   - `display`

2) What does the webpage display if the `hidden` class is removed from an "Enter a city" `<span>`?

   - Nothing. The text remains hidden.
   - "Enter a city" under the textbox.
   - "Enter a city" next to the textbox.

3) What does the webpage display if the `hidden` class is removed from the forecast `<section>`?

   - Nothing. The section remains hidden.
   - Two empty tables.
   - Two "Loading..." messages and two empty tables.

Handling the Compare button click

When the Compare button is clicked, the JavaScript in weather.js needs to:

1. Extract the cities from the text boxes.
2. If no city is entered in a text box, display an error message next to the text box.

3. If two cities are entered, hide the error messages and show the forecast section and only the loading messages.
4. Send two HTTP requests to the forecast API requesting the forecast for the two cities.

The figure below shows some of the code necessary to implement the above logic.

Figure 7.13.2: Compare button callback and supporting code.

```
// Called when Comapre button is clicked
function compareBtnClick() {
   // Get user input
   const city1 =
document.getElementById("city1").value.trim();
   const city2 =
document.getElementById("city2").value.trim();

   // Show error messages if city fields left blank
   if (city1.length === 0) {
      showElement("error-value-city1");
   }
   if (city2.length === 0) {
      showElement("error-value-city2");
   }

   // Ensure both city names provided
   if (city1.length > 0 && city2.length > 0) {
      showElement("forecast");
      hideElement("error-loading-city1");
      hideElement("error-loading-city2");
      showElement("loading-city1");
      showText("loading-city1", `Loading ${city1}...`);
      showElement("loading-city2");
      showText("loading-city2", `Loading ${city2}...`);
      hideElement("results-city1");
      hideElement("results-city2");

      // Fetch forecasts
      getWeatherForecast(city1, "city1");
      getWeatherForecast(city2, "city2");
   }
}

// Display the text in the element
function showText(elementId, text) {
   document.getElementById(elementId).innerHTML = text;
}

// Show the element
function showElement(elementId) {
```

```
document.getElementById(elementId).classList.remove("hidden");
}

// Hide the element
function hideElement(elementId) {
   document.getElementById(elementId).classList.add("hidden");
}
```

Partially implemented Weather Comparison app.

Press Compare before entering a city name to see error messages. Then enter two city names and press Compare. The `getWeatherForecast()` function has not been implemented yet, so no forecasts are displayed.

weather.html | styles.css | weather.js

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Weather Comparison</title>
      <link rel="stylesheet" href="styles.css">
      <script src="weather.js"></script>
   </head>
   <body>
      <h1>Weather Comparison</h1>
      <section id="weather-input">
         <p>
            <label for="city1">City 1:</label>
            <input type="text" id="city1">
            <span class="error-msg hidden" id="error-value-city1">Enter a cit
         </p>
```

Render webpage Reset code

Your webpage

Weather Comparison

City 1:

City 2:

Compare

PARTICIPATION ACTIVITY 7.13.4: Compare button click.

1) Only one "Enter a city" error message can appear at a time.

   - True
   - False

2) An "Enter a city" error message appears if the user just types a few spaces.

   - True
   - False

3) The "Enter a city" error message only disappears when the user enters a city in the text box next to the message and presses Compare again.

- True
- False

4) Pressing the Compare button causes two HTTP requests to be sent to the forecast API.

- True
- False

Requesting the forecast

The `getWeatherForecast()` function is responsible for making an HTTP request to the forecast API. The `XMLHttpRequest` object requests a URL with the given city and registers a load event callback `responseReceived()`, which is called when the forecast API responds with the JSON forecast.

Figure 7.13.3: getWeatherForecast() function.

```
// Request this city's forecast
function getWeatherForecast(city, cityId) {
   // Create a URL to access the web API
   const endpoint = "https://api.openweathermap.org/data/2.5/forecast";
   const apiKey = "Your API key goes here";
   const queryString =
`q=${encodeURI(city)}&units=imperial&appid=${apiKey}`;
   const url = `${endpoint}?${queryString}`;

   // Use XMLHttpRequest to make http request to web API
   const xhr = new XMLHttpRequest();

   // Call responseReceived() when response is received
   xhr.addEventListener("load", function () {
      responseReceived(xhr, cityId, city)
   });

   // JSON response needs to be converted into an object
   xhr.responseType = "json";

   // Send request
   xhr.open("GET", url);
   xhr.send();
}
```

PARTICIPATION ACTIVITY 7.13.5: Requesting the forecast.

1) How is `getWeatherForecast()` properly called?

- `getWeatherForecast("Denver");`
- `getWeatherForecast("Denver", true);`
- `getWeatherForecast("Denver", "city1");`

2) What does the query string look like if Boston is the city?

- ?q=Boston
- ?q=Boston&appid=APIKEY
- ?q=Boston&units=imperial&appid=APIKEY

3) What happens if the line below is removed?

```
xhr.responseType = "json";
```

- No request is sent.
- The request is sent, but a 404 status is returned.
- The request is sent, but the JSON response is not converted into an object.

Processing the JSON response

The `responseReceived()` function transfers the forecast information into the city's table or shows an error message if the forecast API fails to find the city's forecast. The `responseReceived()` function calls several functions:

- `getSummaryForecast()` - Retrieves a map of objects containing the day's high, low, and weather summary for 5 days.
- `getDayName()` - Converts a date like "2018-12-20" into a day name like "Thu".
- `showImage()` - Displays the weather image matching the given weather type.

Figure 7.13.4: responseReceived() and supporting functions.

```
// Display forecast received from JSON
function responseReceived(xhr, cityId, city) {
   // No longer loading
   hideElement("loading-" + cityId);

   // 200 status indicates forecast successfully received
   if (xhr.status === 200) {
```

```
      showElement("results-" + cityId);

      const cityName = xhr.response.city.name;
      showText(cityId + "-name", cityName);

      // Get 5 day forecast
      const forecast = getSummaryForecast(xhr.response.list);

      // Put forecast into the city's table
      let day = 1;
      for (const date in forecast) {
        // Only process the first 5 days
        if (day <= 5) {
           showText(`${cityId}-day${day}-name`, getDayName(date));
           showText(`${cityId}-day${day}-high`,
Math.round(forecast[date].high) + "&deg;");
           showText(`${cityId}-day${day}-low`,
Math.round(forecast[date].low) + "&deg;");
           showImage(`${cityId}-day${day}-image`,
forecast[date].weather);
        }
        day++;
      }
   } else {
      // Display appropriate error message
      const errorId = "error-loading-" + cityId;
      showElement(errorId);
      showText(errorId, `Unable to load city "${city}".`);
   }
}

// Convert date string into Mon, Tue, etc.
function getDayName(dateStr) {
   const date = new Date(dateStr);
   return date.toLocaleDateString("en-US", { weekday: "short", timeZone:
"UTC" });
}

// Show the weather image that matches the weatherType
function showImage(elementId, weatherType) {
   // Images for various weather types
   const weatherImages = {
      Clear: "clear.png",
      Clouds: "clouds.png",
      Drizzle: "drizzle.png",
      Mist: "mist.png",
      Rain: "rain.png",
      Snow: "snow.png"
   };

   const imgUrl = "https://static-resources.zybooks.com/";
   const img = document.getElementById(elementId);
   img.src = imgUrl + weatherImages[weatherType];
   img.alt = weatherType;
```

```
    imgVar = weatherType;
}
```

The `getSummaryForecast()` function loops through the forecast data retrieved from the forecast API and creates a map of 5 or 6 objects that contain the highest and lowest temperature for each day and weather summary string like "Clear", "Rain", or "Snow".

Figure 7.13.5: getSummaryForecast() function.

```
// Return a map of objects that contain the high temp, low temp, and
weather for the next 5 days
function getSummaryForecast(forecastList) {
  // Map for storing high, low, weather
  const forecast = [];

  // Determine high and low for each day
  forecastList.forEach(function (item) {
     // Extract just the yyyy-mm-dd
     const date = item.dt_txt.substr(0, 10);

     // Extract temperature
     const temp = item.main.temp;

      // Has this date been seen before?
     if (date in forecast) {
        // Determine if the temperature is a new low or high
        if (temp < forecast[date].low) {
           forecast[date].low = temp;
        }
        if (temp > forecast[date].high) {
           forecast[date].high = temp;
        }
     }
     else {
        // Initialize new forecast
        const temps = {
           high: temp,
           low: temp,
           weather: item.weather[0].main
        }

        // Add entry to map
        forecast[date] = temps;
     }
  });

  return forecast;
}
```

Example return value:

```
2019-01-22: {high: 43, low: 38.93, weather: "Rain"},
2019-01-23: {high: 47.02, low: 42.8, weather: "Rain"},
2019-01-24: {high: 47.44, low: 43.43, weather: "Clouds"},
2019-01-25: {high: 45.45, low: 40.39, weather: "Clouds"},
2019-01-26: {high: 42.79, low: 39.57, weather: "Clear"},
2019-01-27: {high: 41.33, low: 37.3, weather: "Clear"}
```

Weather Comparison app.

The complete implementation is provided below. Enter two city names like "Denver" and "Seattle" and press Compare to see the forecasts compared.

The implementation uses zyBook's API key. You may need to replace the API key in `getWeatherForecast()` with your own key due to API query limits.

Weather images from OpenWeatherMap.org

weather.html | styles.css | weather.js

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8">
      <title>Weather Comparison</title>
      <link rel="stylesheet" href="styles.css">
      <script src="weather.js"></script>
   </head>
   <body>
      <h1>Weather Comparison</h1>
      <section id="weather-input">
         <p>
            <label for="city1">City 1:</label>
            <input type="text" id="city1">
            <span class="error-msg hidden" id="error-value-city1">Enter a cit
         </p>
```

Render webpage Reset code

Your webpage

Weather Comparison

City 1:

City 2:

Compare

PARTICIPATION ACTIVITY 7.13.6: Displaying the forecast.

1) When does `responseReceived()` display an error message instead of the forecast?

- When the response status code is anything but 200.
- When the user doesn't type two city names.
- When the Compare button is pressed a second time.

2) What is the likely size of `xhr.response.list` when `getSummaryForecast(xhr.response.list)` is called?

- 1
- 20
- 40

3) The figure above shows an example return value from calling `getSummaryForecast()`. How many `xhr.response.list` forecast objects were likely examined to produce high of 47.44 and low of 43.43 on 2019-01-24?

- 1
- 8
- 40

7.14 LAB: Grade distribution