JSON Learning Objectives

The objective of this lesson is to familiarize you with the JSON format and how to serialize to and deserialize from that format.

The learning objectives for this lesson are that you can:

1. Identify and generate valid JSON-formatted strings
2. Use `JSON.parse` to deserialize JSON-formatted strings
3. Use `JSON.stringify` to serialize JavaScript objects
4. Correctly identify the definition of "deserialize"
5. Correctly identify the definition of "serialize"

This lesson is relevant because JSON is the *lingua franca* of data interchange.

Storage Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Write JavaScript to store the value "I <3 falafel" with the key "eatz" in the browser's local storage.
2. Write JavaScript to read the value stored in local storage for the key "paper-trail".

Browser Basics Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Explain the difference between the BOM (browser object model) and the DOM(document object model).
2. Given a diagram of all the different parts of the Browser identify each part. Use the Window API to change the innerHeight of a user's window.
3. Identify the context of an anonymous functions running in the Browser (the window).
4. Given a JS file and an HTML file, use a script tag to import the JS file and execute the code therein when all the elements on the page load (using`DOMContentLoaded`)
5. Given a JS file and an HTML file, use a script tag to import the JS file and execute the code therein when the page loads
6. Identify three ways to prevent JS code from executing until an entire HTML page is loaded
7. Label a diagram on the Request/Response cycle.
8. Explain the Browser's main role in the request/response cycle. (1.Parsing HTML,CSS, JS 2. Rendering that information to the user by constructing a DOM tree and rendering it)
9. Given several detractors - identify which real-world situations could be implemented with the Web Storage API (shopping cart, forms saving inputs etc.)
10. Given a website to visit that depends on cookies (like Amazon), students should be able to go to that site add something to their cart and then delete that cookie using the Chrome Developer tools in order to empty their cart.

Element Selection Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Given HTML that includes `<div id=”catch-me-if-you-can”>HI!</div>`, write a JavaScript statement that stores a reference to the HTMLDivElement with the id “catch-me-if-you-can” in a variable named “divOfInterest”.
2. Given HTML that includes seven SPAN elements each with the class “cloudy”, write a JavaScript statement that stores a reference to a NodeList filled with references to the seven HTMLSpanElements in a variable named “cloudySpans”.
3. Given an HTML file with HTML, HEAD, TITLE, and BODY elements, create and reference a JS file that in which the JavaScript will create and attach to the BODY element an H1 element with the id "sleeping-giant" with the content "Jell-O, Burled!".
4. Given an HTML file with HTML, HEAD, TITLE, SCRIPT, and BODY elements with the SCRIPT's SRC attribute referencing an empty JS file, write a script in the JS file to create a DIV element with the id "lickable-frog" and add it as the last child to the BODY element.
5. Given an HTML file with HTML, HEAD, TITLE, SCRIPT, and BODY elements with no SRC attribute on the SCRIPT element, write a script in the SCRIPT block to create a UL element with no id, create an LI element with the id "dreamy-eyes", add the LI as a child to the UL element, and add the UL element as the first child of the BODY element.
6. Write JavaScript to add the CSS class "i-got-loaded" to the BODY element when the window fires the DOMContentLoaded event.
7. Given an HTML file with a UL element with the id "your-best-friend" that has six non-empty LIs as its children, write JavaScript to write the content of each LI to the console.
8. Given an HTML file with a UL element with the id "your-worst-enemy" that has no children, write JavaScript to construct a string that contains six LI tags each containing a random number and set the inner HTML property of ul#your-worst-enemy to that string.
9. Write JavaScript to update the title of the document to the current time at a reasonable interval such that it looks like a real clock.

Event Handling Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Given an HTML page that includes `<button id="increment-count">I have been clicked <span id="clicked-count">0</span> times</button>`, write JavaScript that increases the value of the content of span#clicked-count by 1 every time`button#increment-count`is clicked.
2. Given an HTML page that includes`<input type="checkbox" id="on-off"><div id="now-you-see-me">Now you see me</div>`, write JavaScript that sets the display of div#now-you-see-me to "none" when input#on-off is checked and to "block" when input#on-off is not checked.
3. Given an HTML file that includes`<input id="stopper" type="text" placeholder="Quick! Type STOP">`, write JavaScript that will change the background color of the page to cyan five seconds after a page loads unless the field input#stopper contains only the text "STOP".
4. Given an HTML page with that includes `<input type=”text” id=”fancypants”>`, write JavaScript that changes the background color of the textbox to #E8F5E9 when the caret is in the textbox and turns it back to its normal color when focus is elsewhere.
5. Given an HTML page that includes a form with two password fields, write JavaScript that subscribes to the forms submission event and cancels it if the values in the two password fields differ.
6. Given an HTML page that includes a div styled as a square with a red background, write JavaScript that allows a user to drag the square around the screen.
7. Given an HTML page that has 300 DIVs, create one click event subscription that will print the id of the element clicked on to the console.
8. Identify the definition of the bubbling principle.

Browser Basics Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Explain the difference between the BOM (browser object model) and the DOM(document object model).
2. Given a diagram of all the different parts of the Browser identify each part. Use the Window API to change the innerHeight of a user's window.
3. Identify the context of an anonymous functions running in the Browser (the window).
4. Given a JS file and an HTML file, use a script tag to import the JS file and execute the code therein when all the elements on the page load (using`DOMContentLoaded`)
5. Given a JS file and an HTML file, use a script tag to import the JS file and execute the code therein when the page loads
6. Identify three ways to prevent JS code from executing until an entire HTML page is loaded
7. Label a diagram on the Request/Response cycle.
8. Explain the Browser's main role in the request/response cycle. (1.Parsing HTML,CSS, JS 2. Rendering that information to the user by constructing a DOM tree and rendering it)
9. Given several detractors - identify which real-world situations could be implemented with the Web Storage API (shopping cart, forms saving inputs etc.)
10. Given a website to visit that depends on cookies (like Amazon), students should be able to go to that site add something to their cart and then delete that cookie using the Chrome Developer tools in order to empty their cart.

Browsers: They're The BOM Dot Com!

If the Internet exists, but there's no way to browse it, does it even really exist? Unless you've been living under a rock for the past couple decades, you should know what a browser is, and you probably use it multiple times a day. Browsers are something most people take for granted, but behind the scenes is a complex structure working to display information to users who browse the Web.

Web developers rely on browsers constantly. They can be your best friend or your worst enemy. (*Yes, we're looking at you, IE!*) Spending some time learning about browsers will help you get a higher-level understanding of how the Web operates, how to debug, and how to write code that works across browsers. In this reading, we'll learn about the BOM (Browser Object Model), how it's structured, and how it differs from the DOM (Document Object Model).

The DOM vs. the BOM

By now, you've learned about the **DOM, or Document Object Model**, and that it contains a collection of nodes (HTML elements), that can be accessed and manipulated. In essence, the `document`object is a Web page, and the DOM represents the object hierarchy of that document.

How do we access a document on the Web? Through a browser, of course! If we took a bird's-eye view of the browser, we would see that the document object is part of a hierarchy of browser objects. This hierarchy is known as the **BOM, or Browser Object Model**.

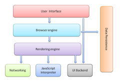
The chief browser object is the `window`object, which contains properties and methods we can use to access different objects within the window. These include:

- `window.navigator`
  - Returns a reference to the navigator object.
- `window.screen`
  - Returns a reference to the screen object associated with the window.
- `window.history`
  - Returns a reference to the history object.
- `window.location`
  - Gets/sets the location, or current URL, of the window object.
- `window.document`, which can be shortened to just `document`
  - Returns a reference to the document that the window contains.

Note how we can shorten `window.document`to `document.`For example, the`document`in `document.getElementById('id')`actually refers to`window.document`. All of the methods above can be shortened in the same way.

The browser diagram

We started in the DOM, and we stepped outside it into the BOM. Now, let's take an even higher view of the browser itself. Take a look at this diagram depicting a high-level structure of the browser, from html5rocks.com:



- **User interface**: This is the browser interface that users interact with, including the address bar, back and forward buttons, bookmarks menu, etc. It includes everything except for the requested page content.
- **Browser engine**: Manages the interactions between the UI and the rendering engine.
- **Rendering engine**: Displays, or renders, the requested page content. If the requested content is HTML, it will parse HTML and CSS and render the parsed content.
- **Networking**: Handles network calls, such as HTTP requests.
- **Javascript interpreter**: Parses and executes JavaScript code.
- **UI backend**: Used for drawing basic widgets like combo boxes and windows; uses operating system user interface methods.
- **Data storage**: The persistence of data stored in the browser, such as cookies.

What we learned:

- Review of the DOM, or Document Object Model
- How the BOM differs from the DOM
- The window object and related methods

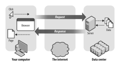
The Request-Response Cycle

Browsing the Web might seem like magic, but it’s really just a series of**requests**and **responses**. When we search for information or navigate to a Web page, we are requesting something, and we expect to get a response back.

We can think about the request-response cycle as the communication pattern between a client, or browser, and a server. Whenever we type a URL into the address bar of a browser, we are making a *request*to a server for information back. The most common of these is an `http request`.

The request-response cycle diagram

Let’s take a look at this diagram of the request-response cycle fromO’Reily:



On the left is the **client**side, or the browser. On the right is the**server**side, with a database where data is stored. The internet, in the middle, is a series of these client requests and server responses. We'll be going into more depth with servers soon, but for right now we are focusing on the client side.

The browser’s role in the request-response cycle

As depicted in the diagram, the browser plays a key role in the request-response cycle. Besides letting the user make the request to the server, the browser also:

1. Parses HTML, CSS, and JS
2. Renders that information to the user by constructing and rendering a DOM tree

When we make a successful request to the server, we are able to view a Web page with content and functionality. Unsuccessful requests prevent the page from loading and displaying information. You've probably seen a 404 page before!

Understanding the request-response cycle is fundamental to developing for the Web. If a server is down, or something is wrong with the request, you’ll most likely see an error on the client side. Learning how to debug these errors and set up error handling is a common task for Web developers.

You can go to the **Network tab** of your browser’s **Developer Tools** to view these requests and responses. Open a new tab, open up the Developer Tools in your browser, and then navigate to `google.com`. Watch the request go through in your Network tab!

What we learned:

- Reviewed diagram of request-response cycle
- The client side vs. the server side
- The role of the browser
- Where to view Network requests in the browser

Running Scripts In The Browser

Timing is everything, in life as well as in code that runs in a browser. Executing a script at the right time is an important part of developing front-end code. A script that runs too early or too late can cause bugs and dramatically affect user experience. After reading this section, you should be able to utilize the proper methods for ensuring your scripts run at the right time.

In previous sections, we reviewed how the DOM and BOM works and used event listeners to trigger script execution. In this lesson, we'll dig deeper into the`window`object and learn multiple ways to ensure a script runs after the necessary objects are loaded.

Using the Window API

The `window`object, the core of the Browser Object Model (BOM), has a number of properties and methods that we can use to reference the window object. Refer to the MDN documentation on the Window API for a detailed list of methods and properties. We'll explore some of these methods now to give you a better grasp on what the `window`object can do for you.

Let's use a Window API method called `resizeTo()`to change the width and height of a user's window in a script.

```
// windowTest.js

// Open a new window
newWindow = window.open("", "", "width=100, height=100");

// Resize the new window
newWindow.resizeTo(500, 500);
```

You can execute the code above in your web browser in Google Chrome by right clicking the page, selecting inspect, and selecting the console tab. Paste the code above into the console. When you do this, make sure you are not in full-screen mode for Chrome, otherwise you won't be able to resize the new window!

Note: You must open a new window using `window.open`before it can be resized. This method won't work in an already open window or in a new tab.

Check out the documentation on Window.resizeTo()and Window.resizeBy()for more information.

Go to wikipediaand try setting the window scroll position by pasting `window.scroll(0,300)`in the developer console (right click, inspect, console like usual). Play around with different scroll values. Pretty neat, huh?

Context, scope, and anonymous functions

Two important terms to understand when you're developing in Javascript are**context**and **scope**. Ryan Morr has a great write-up about the differences between the two here: "Understanding Scope and Context in Javascript".

The important things to note about **context**are:

1. Every function has a context (as well as a scope).
2. Context refers to the object that *owns*the function (i.e. the value of*this*inside a given function).
3. Context is most often determined by how a function is invoked.

Take, for example, the following code:

```
const foo = {
  bar: function() {
    return this;
  }
};
console.log(foo.bar() === foo);
// returns true
```

The anonymous function above is a method of the `foo`object, which means that`this`returns the object itself — the context, in this case.

What about functions that are unbound to an object, or not scoped inside of another function? Try running this anonymous function, and see what happens.

```
(function() {
  console.log(this);
})();
```

When you open your console in the browser and run this code, you should see the`window`object printed. When a function is called in the global scope, `this`defaults to the global context, or in the case of running code in the browser, the `window`object.

Refer to “Understanding Scope and Context in Javascript”for more about the scope chain, closures, and using `.call()`and `.apply()`on functions.

Running a script on DOMContentLoaded

Now you will learn how to run a script on `DOMContentLoaded`, when the document has been loaded without waiting for stylesheets, images and subframes to load.

Let’s practice. Set up an HTML file, import an external JS file, and run a script on `DOMContentLoaded`.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="dom-ready-script.js"></script>
  </head>
  <body></body>
  <html></html>
</html>
```

JS

```
window.addEventListener("DOMContentLoaded", event => {
  console.log("This script loaded when the DOM was ready.");
});
```

Running a script on page load

`DOMContentLoaded`ensures that a script will run when the document has been loaded without waiting for stylesheets, images and subframes to load. However, if we wanted to wait for **everything**in the document to load before running the script, we could instead use the `window`object method `window.onload`.

Let’s practice it here. Set up an HTML file, import an external JS file, and run a script on `window.onload`.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="window-load-script.js"></script>
  </head>
  <body></body>
  <html></html>
</html>
```

JS

```
window.onload = () => {
  console.log(
    "This script loaded when all the resources and the DOM were ready."
  );
};
```

Ways to prevent a script from running until page loads

There are actually multiple ways to prevent a script from running until the page has loaded. We'll review three of them here:

1. Use the `DOMContentLoaded`event in an external JS file
2. Put a script tag importing your external code at the bottom of your HTML file
3. Add an attribute to the script tag, like `async`or `defer`

We've reviewed the first method above. Let's now review numbers **2** and **3**. If you want to make sure that all your HTML has loaded before a script runs, an easy option is to include your script immediately after the HTML you need. This works because HTML builds the DOM tree in the order of how your HTML file is structured. Whatever is on top will be loaded first, such as script tags in the`<head>`. It makes sense, then, to keep your script at the bottom of your HTML, right before the closing `</body>`tag, like below.

```
<html>
  <head></head>
  <body>
    …
    <script src="script.js"></script>
  </body>
</html>
```

If you want to include your script in the `<head>`tags, rather than the`<body>`tags, there is another option: We could use the `async`or `defer`methods in our `<script>`tag. Flavio Copes has a great write-upon using`async`or `defer`with graphics showing exactly when the browser parses HTML, fetches the script, and executes the script.

With `async`, a script is fetched asynchronously. After the script is fetched, HTML parsing is paused to execute the script, and then it's resumed. With`defer`, a script is fetched asynchronously and is executed only after HTML parsing is finished.

You can use the `async`and `defer`methods independently or simultaneously. Newer browsers recognize `async`, while older ones recognize `defer`. If you use`async defer`simultaneously, `async`takes precedence, while older browsers that don't recognize it will default to `defer`. Check `caniuse.com`to see which browsers are compatible with asyncand defer.

```
  <script async src="scriptA.js"></script>

  <script defer src="scriptB.js"></script>

  <script async defer src="scriptC.js"></script>
```

What we learned:

- How to use Window API methods
- The context and scope of a function
- Review of `DOMContentLoaded`and `window.onload`
- How to prevent a script from running until a page loads

Element Selection Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Given HTML that includes `<div id=”catch-me-if-you-can”>HI!</div>`, write a JavaScript statement that stores a reference to the HTMLDivElement with the id “catch-me-if-you-can” in a variable named “divOfInterest”.
2. Given HTML that includes seven SPAN elements each with the class “cloudy”, write a JavaScript statement that stores a reference to a NodeList filled with references to the seven HTMLSpanElements in a variable named “cloudySpans”.
3. Given an HTML file with HTML, HEAD, TITLE, and BODY elements, create and reference a JS file that in which the JavaScript will create and attach to the BODY element an H1 element with the id "sleeping-giant" with the content "Jell-O, Burled!".
4. Given an HTML file with HTML, HEAD, TITLE, SCRIPT, and BODY elements with the SCRIPT's SRC attribute referencing an empty JS file, write a script in the JS file to create a DIV element with the id "lickable-frog" and add it as the last child to the BODY element.
5. Given an HTML file with HTML, HEAD, TITLE, SCRIPT, and BODY elements with no SRC attribute on the SCRIPT element, write a script in the SCRIPT block to create a UL element with no id, create an LI element with the id "dreamy-eyes", add the LI as a child to the UL element, and add the UL element as the first child of the BODY element.
6. Write JavaScript to add the CSS class "i-got-loaded" to the BODY element when the window fires the DOMContentLoaded event.
7. Given an HTML file with a UL element with the id "your-best-friend" that has six non-empty LIs as its children, write JavaScript to write the content of each LI to the console.
8. Given an HTML file with a UL element with the id "your-worst-enemy" that has no children, write JavaScript to construct a string that contains six LI tags each containing a random number and set the inner HTML property of ul#your-worst-enemy to that string.
9. Write JavaScript to update the title of the document to the current time at a reasonable interval such that it looks like a real clock.

Hello, World DOMination: Element Selection And Placement

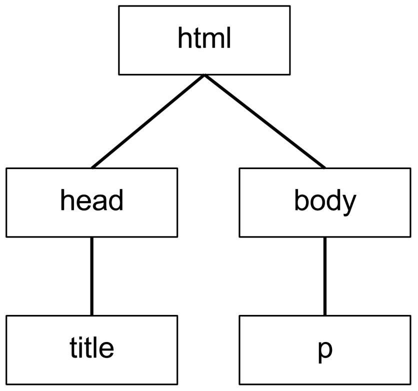
The objective of this lesson is to familiarize yourself with the usage and inner workings of the DOM API. When you finish this lesson, you should be able to:

- Reference and manipulate the DOM via Javascript
- Update and create new DOM elements via Javascript
- Change CSS based on a DOM event
- Familiarize yourself with the console

What is the DOM?

TheDocument Object Model, or DOM, is an object-oriented representation of an HTML document or Web page, meaning that the document is represented as objects, or nodes. It allows developers to access the document via a programming language, like Javascript.

The DOM is typically depicted as a tree with a specific hierarchy. (See the image below.) Higher branches represent parent nodes, while lower branches represent child nodes, or children. More on that later.



Referencing the DOM

The DOM API is one of the most powerful tools frontend developers have at their disposal. Learning how to reference, create, and update DOM elements is an integral part of working with Javascript. We'll start this lesson by learning how to reference a DOM element in Javascript.

Let’s assume we have an HTML file that includes the following `div`:

HTML

```
<div id="”catch-me-if-you-can”">HI!</div>
```

Because we've added the element to our HTML file, that element is available in the DOM for us to reference and manipulate. Using JavaScript, we can reference this element by scanning the document and finding the element by its id with the method document.getElementById(). We then assign the reference to a variable.

Javascript

```
const divOfInterest = document.getElementById(“catch-me-if-you-can”)
```

Now let’s say that our HTML file contains seven `span`elements that share a class name of `cloudy`, like below:

HTML

```
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
<span class="“cloudy”"></span>
```

In Javascript, we can reference all seven of these elements and store them in a single variable.

Javascript

```
const cloudySpans = document.querySelectorAll("span.cloudy");
```

While `getElementById`allows us to reference a single element,`querySelectorAll`references all elements with the class name “cloudy” as a static `NodeList`(*static*meaning that any changes in the DOM do not affect the content of the collection). Note that a NodeList is different from an array, but it is possible to iterate over a NodeList as with an array usingforEach(). Refer to the MDN doc onNodeListfor more information.

Using `forEach()`on a NodeList:

Javascript

```
const cloudySpans = document.querySelectorAll("span.cloudy");

cloudySpans.forEach(span => {
  console.log("Cloudy!");
});
```

Creating New DOM Elements

Now that we know how to reference DOM elements, let's try creating new elements. First we'll set up a basic HTML file with the appropriate structure and include a reference to a Javascript file that exists in the same directory in the`head`.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
    <script type="text/javascript" src="example.js"></script>
  </head>
  <body></body>
</html>
```

In our example.js file, we'll write a function to create a new `h1`element, assign it an id, give it content, and attach it to the body of our HTML document.

Javascript

```
const addElement = () => {
  // create a new div element
  const newElement = document.createElement("h1");
  // set the h1's id
  newElement.setAttribute("id", "sleeping-giant");

  // and give it some content
  const newContent = document.createTextNode("Jell-O, Burled!");

  // add the text node to the newly created div
  newElement.appendChild(newContent);

  // add the newly created element and its content into the DOM
  document.body.appendChild(newElement);
};
// run script when page is loaded
window.onload = addElement;
```

If we open up our HTML file in a browser, we should now see the words `Jell-O Burled!` on our page. If we use the browser tools to inspect the page (right-click on the page and select “inspect”, or hotkeys fn + f12), we notice the new `h1` with the id we gave it.

Hello, World DOMination: Inserting Elements in JS File and Script Block

Let's practice adding new elements to our page. We'll create a second element, a `div`with an id of `lickable-frog`, and append it to the `body`like we did the first time. Update the Javascript function to append a second element to the page.

Javascript

```
const addElements = () => {
  // create a new div element
  const newElement = document.createElement("h1");

  // set the h1's id
  newElement.setAttribute("id", "sleeping-giant");

  // and give it some content
  const newContent = document.createTextNode("Jell-O, Burled!");

  // add the text node to the newly created div
  newElement.appendChild(newContent);

  // add the newly created element and its content into the DOM
  document.body.appendChild(newElement);

  // append a second element to the DOM after the first one
  const lastElement = document.createElement("div");
  lastElement.setAttribute("id", "lickable-frog");
  document.body.appendChild(lastElement);
};
// run script when page is loaded
window.onload = addElements;
```

Notice that our function is now called `addElements`, plural, because we're appending two elements to the `body`. Save your Javascript file and refresh the HTML file in the browser. When you inspect the page, you should now see two elements in the `body`, the `h1`and the `div` we added via Javascript.

Referencing a JS File vs. Using a Script Block

In our test example above, we referenced an external JS file, which contained our function to add new elements to the DOM. Typically, we would keep Javascript in a separate file, but we could also write a script block directly in our HTML file. Let's try it. First, we'll delete the script source so that we have an empty script block.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript">
      //Javascript goes here!
    </script>
  </head>
  <body></body>
</html>
```

Inside of our script block, we'll:

- create a `ul`element with no id
- create an `li`element with the id `dreamy-eyes`
- add the `li`as a child to the `ul`element
- add the `ul`element as the first child of the `body`element.

HTML

```html
<!DOCTYPE html>
<html>
  <head>
    <title>My Cool Website</title>
    <script type="text/javascript">
      const addListElement = () => {
        const listElement = document.createElement("ul");
        const listItem = document.createElement("li");
        listItem.setAttribute("id", "dreamy-eyes");
        listElement.appendChild(listItem);
        document.body.prepend(listElement);
      };
      window.onload = addListElement;
    </script>
  </head>
  <body></body>
</html>
```

Refresh the HTML in your browser, inspect the page, and notice the `ul`and `li`elements that were created in the script block.

Hello, World DOMination: Adding a CSS Class After DOM Load Event

In our previous JS examples, we used `window.onload`to run a function after the window has loaded the page, which ensures that all of the objects are in the DOM, including images, scripts, links, and subframes. However, we don't need to wait for stylesheets, images, and subframes to finish loading before our JavaScript runs because JS isn't dependent on them. And, some images may be so large that waiting on them to load before the JS runs would make the user experience feel slow and clunky. There is a better method to use in this case:`DOMContentLoaded`.

According to MDN, "the DOMContentLoaded event fires when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading."

We'll use DOMContentLoaded to add CSS classes to page elements immediately after they are loaded. Let's add the CSS class "i-got-loaded" to the `body`element when the window fires the DOMContentLoaded event. We can do this in the script block or in an external JS file, as we did in the examples above.

Javascript

```
window.addEventListener("DOMContentLoaded", event => {
  document.body.className = "i-got-loaded";
});
```

After adding the Javascript above, refresh the HTML in your browser, inspect the page, and notice that the `body`element now has a class of "i-got-loaded".

Hello, World DOMination: Console.log, Element.innerHTML, and the Date Object

In this section, we'll learn about how to use `console.log`to print element values. We'll also use `Element.innerHTML`to fill in the HTML of a DOM element. Finally, we'll learn about the Javascript Date object and how to use it to construct a clock that keeps the current time.

Console Logging Element Values

Along with the other developer tools, the console is a valuable tool Javascript developers use to debug and check that scripts are running correctly. In this exercise, we'll practice logging to the console.

Create an HTML file that contains the following:

HTML

```html
<!DOCTYPE html>
<html>
  <head> </head>
  <body>
    <ul id="your-best-friend">
      <li>Has your back</li>
      <li>Gives you support</li>
      <li>Actively listens to you</li>
      <li>Lends a helping hand</li>
      <li>Cheers you up when you're down</li>
      <li>Celebrates important moments with you</li>
    </ul>
  </body>
</html>
```

In the above code, we see an id with which we can reference the `ul`element. Recall that we previously used `document.querySelectorAll()`to store multiple elements with the same class name in a single variable, as a NodeList. However, in the above example, we see only one id for the parent element. We can reference the parent element via its id to get access to the content of its children.

Javascript

```javascript
window.addEventListener("DOMContentLoaded", event => {
  const parent = document.getElementById("your-best-friend");
  const childNodes = parent.childNodes;
  for (let value of childNodes.values()) {
    console.log(value);
  }
});
```

In your browser, use the developer tools to open the console and see that the values of each `li`have been printed out.

Using Element.innerHTML

Thus far, we have referenced DOM elements via their id or class name and appended new elements to existing DOM elements. Additionally, we can use the inner HTML property to get or set the HTML or XML markup contained within an element.

In an HTML file, create a `ul`element with the id "your-worst-enemy" that has no children. We'll add some JavaScript to construct a string that contains six `li`tags each containing a random number and set the inner HTML property of`ul#your-worst-enemy`to that string.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="example.js"></script>
  </head>
  <body>
    <ul id="your-worst-enemy"></ul>
  </body>
</html>
```

Javascript

```
// generate a random number for each list item
const getRandomInt = max => {
  return Math.floor(Math.random() * Math.floor(max));
};

// listen for DOM ready event
window.addEventListener("DOMContentLoaded", event => {
  // push 6 LI elements into an array and join
  const liArr = [];
  for (let i = 0; i < 6; i++) {
    liArr.push("<li>" + getRandomInt(10) + "</li>");
  }
  const liString = liArr.join(" ");

  // insert string into the DOM using innerHTML
  const listElement = document.getElementById("your-worst-enemy");
  listElement.innerHTML = liString;
});
```

Save your changes, and refresh your browser page. You should see six new list items on the page, each containing a random number.

Inserting a Date Object into the DOM

We've learned a lot about accessing and manipulating the DOM! Let's use what we've learned so far to add extra functionality involving the Javascript Date object.

Our objective is to update the title of the document to the current time at a reasonable interval such that it looks like a real clock.

We know we'll be starting with an HTML document that contains an empty title element. We've learned a couple of different ways to fill the content of an element so far. We could create a new element and append it to the title element, or we could use `innerHTML`to set the HTML of the title element. Since we don't need to create a new element nor do we care whether it appears last, we can use the latter method.

Let's give our title an id for easy reference.

HTML

```
<title id="title"></title>
```

In our Javascript file, we'll use the Date constructor to instantiate a newDate object.

```
const date = new Date();
```

Javascript

```
window.addEventListener("DOMContentLoaded", event => {
  const title = document.getElementById("title");
  const time = () => {
    const date = new Date();
    const seconds = date.getSeconds();
    const minutes = date.getMinutes();
    const hours = date.getHours();

    title.innerHTML = hours + ":" + minutes + ":" + seconds;
  };
  setInterval(time, 1000);
});
```

Save your changes and refresh your browser. Observe the clock we inserted dynamically keeping the current time in your document title!

What We Learned:

- What the DOM is and how we can access it
- How to access DOM elements using `document.getElementById()` and `document.querySelectorAll()`
- How to create new elements with `document.createElement()` and `document.createTextNode`, and append them to existing DOM elements with `Element.appendChild()`
- The difference between `window.onload` and `DOMContentLoaded`
- How to access children nodes with `NodeList.childNodes`
- Updating DOM elements with `Element.innerHTML`
- The Javascript Date object

Brush Up On Your HTML

You'll be using a lot of HTML in the following days (weeks, months, years), so might as well get a leg up by reacquainting yourself with HTML.

The definitive resource on the Internet for HTML, CSS, and JavaScript is the Mozilla Developer Network. Go there and work through, at a minimum, the following sections:

- The following sections in Introduction to HTML
    - Getting started with HTML
    - What's in the head? Metadata in HTML
    - HTML text fundamentals
    - Creating hyperlinks
- The following section in Multimedia and embedding
    - Images in HTML
- The following section in HTML forms
    - Your first HTML form
    - How to structure an HTML form
    - The native form widgets
    - Client-side form validation
    - Styling HTML forms

Event Handling Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Given an HTML page that includes `<button id="increment-count">I have been clicked <span id="clicked-count">0</span> times</button>`, write JavaScript that increases the value of the content of span#clicked-count by 1 every time`button#increment-count`is clicked.
2. Given an HTML page that includes`<input type="checkbox" id="on-off"><div id="now-you-see-me">Now you see me</div>`, write JavaScript that sets the display of div#now-you-see-me to "none" when input#on-off is checked and to "block" when input#on-off is not checked.
3. Given an HTML file that includes`<input id="stopper" type="text" placeholder="Quick! Type STOP">`, write JavaScript that will change the background color of the page to cyan five seconds after a page loads unless the field input#stopper contains only the text "STOP".
4. Given an HTML page with that includes `<input type=”text” id=”fancypants”>`, write JavaScript that changes the background color of the textbox to #E8F5E9 when the caret is in the textbox and turns it back to its normal color when focus is elsewhere.
5. Given an HTML page that includes a form with two password fields, write JavaScript that subscribes to the forms submission event and cancels it if the values in the two password fields differ.
6. Given an HTML page that includes a div styled as a square with a red background, write JavaScript that allows a user to drag the square around the screen.
7. Given an HTML page that has 300 DIVs, create one click event subscription that will print the id of the element clicked on to the console.
8. Identify the definition of the bubbling principle.

Event Handling: Common Page Events

Event handling is the core of front-end development. When a user interacts with HTML elements on a website, those interactions are known as ***events***. Developers use Javascript to respond to those events. In this reading, we'll go over three common events and do exercises to add functionality based on those events:

- A button click
- A checkbox being checked
- A user typing a value into an input

Handling a button click event

Let's start with a common event that occurs on many websites: a button click. Usually some functionality occurs when a button is clicked -- such as displaying new page elements, changing current elements, or submitting a form.

We'll go through how to set up a click eventlistener and update the click count after each click. Let's say we have a button element in an HTML file, like below:

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js">
  </head>
  <body>
    <button id="increment-count">I have been clicked <span id="clicked-count">0</
  </body>
</html>
```

We'll write Javascript to increase the value of the content of`span#clicked-count`by one each time `button#increment-count` is clicked. Remember to use the `DOMContentLoaded`event listener in an external script to ensure the button has loaded on the page before the script runs.

Javascript

If you open up the HTML file in a browser, you should see the button. If you click the button rapidly and repeatedly, the value of `span#clicked-count`should increment by one after each click.

```
// script.js

window.addEventListener("DOMContentLoaded", event => {
  const button = document.getElementById("increment-count");
  const count = document.getElementById("clicked-count");
  let clicks = 0;
  button.addEventListener("click", event => {
    clicks += 1;
    count.innerHTML = clicks;
  });
});
```

Using addEventListener() vs. onclick

Adding an event listener to the button element, as we did above, is the preferred method of handling events in scripts. However, there is another method we could use here: GlobalEventHandlers.onclick. Check outcodingrepo.comfor a breakdown of the differences between using`addEventListener()`and `onclick`. One distinction is that `onclick`overrides existing event listeners, while `addEventListener()`does not, making it easy to add new event listeners.

The syntax for `onclick`is: `target.onclick = functionRef;`If we wanted to rewrite the button click event example using `onclick`, we would use the following:

```
let clicks = 0;
button.onclick = event => {
  clicks += 1;
  count.innerHTML = clicks;
};
```

We'll stick to using `addEventListener()`in our code, but it's important for front-end developers to understand the differences between the methods above and use cases for each one.

Handling a checkbox check event

Another common event that occurs on many websites is when a user checks a checkbox. Checkboxes are typically recorded values that get submitted when a user submits a form, but checking the box sometimes also triggers another function.

Let's practice displaying an element when the box is checked and hiding it when the box is unchecked. We'll pretend we're on a pizza delivery website, and we're filling out a form for pizza toppings. There is a checkbox on the page for extra cheese, and when a user checks that box we want to show a `div`with pricing info. Let's set up our HTML file with a `checkbox`and `div` to show/hide, as well as a link to our Javascript file:

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js">
  </head>
  <body>
    <h1>Pizza Toppings</h1>
    <input type="checkbox" id="on-off">
    <label for="on-off">Extra Cheese</label>
    <div id="now-you-see-me" style="display:none">Add $1.00</div>
  </body>
</html>
```

Note that we've added `style="display:none"`to the `div` so that, when the page first loads and the box is unchecked, the `div`won't show.

In our `script.js`file, we'll set up an event listener for `DOMContentLoaded`again to make sure the `checkbox`and `div`have loaded. Then, we'll write Javascript to show `div#now-you-see-me`when the box is checked and hide it when the box is unchecked.

Javascript

```
// script.js

window.addEventListener("DOMContentLoaded", event => {
  // store the elements we need in variables
  const checkbox = document.getElementById("on-off");
  const divShowHide = document.getElementById("now-you-see-me");
  // add an event listener for the checkbox click
  checkbox.addEventListener("click", event => {
    // use the 'checked' attribute of checkbox inputs
    // returns true if checked, false if unchecked
    if (checkbox.checked) {
      // if the box is checked, show the div
      divShowHide.style.display = "block";
      // else hide the div
    } else {
      divShowHide.style.display = "none";
    }
  });
});
```

Open up the HTML document in a browser and make sure that you see the `checkbox`when the page first loads and not the `div`. The `div`should show when you check the box, and appear hidden when you uncheck the box.

The code above works. However, what would happen if we had a whole page of checkboxes with extra options inside each one that would show or hide based on whether the boxes are checked? We would have to call`Element.style.display = "block"`and `Element.style.display = "none"`on each associated `div`.

Instead, we could add a `show`or `hide`class to the `div`based on the checkbox and keep our `display:block`and `display:none`in CSS. That way, we could reuse the classes on different elements, as well as see class names change in the HTML. Here's how the code we wrote above would look if we used CSS classes:

Javascript

```
// script.js
// we need to wait for the stylesheet to load
window.onload = () => {
  // store the elements we need in variables
  const checkbox = document.getElementById("on-off");
  const divShowHide = document.getElementById("now-you-see-me");
  // add an event listener for the checkbox click
  checkbox.addEventListener("click", event => {
    // use the 'checked' attribute of checkbox inputs
    // returns true if checked, false if unchecked
    if (checkbox.checked) {
      // if the box is checked, show the div
      divShowHide.classList.remove("hide");
      divShowHide.classList.add("show");
      // else hide the div
    } else {
      divShowHide.classList.remove("show");
      divShowHide.classList.add("hide");
    }
  });
};
```

CSS

```
.show {
  display: block;
}
.hide {
  display: none;
}
```

HTML (Remove the style attribute, and add the "hide" class)

```
<div id="now-you-see-me" class="hide">Add $1.00</div>
```

Handling a user input value

You've learned a lot about event handling so far! Let's do one more exercise to practice event handling using an input. In this exercise, we'll write JavaScript that will change the background color of the page to cyan five seconds after a page loads unless the field `input#stopper`contains only the text "STOP".

Let's set up an HTML file with the input and a placeholder directing the user to type "STOP": **HTML**

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

```
    <script src="script.js">
  </head>
  <body>
    <input id="stopper" type="text" placeholder="Quick! Type STOP">
  </body>
</html>
```

Now let’s set up our Javascript:

Javascript

```
// script.js
// run when the DOM is ready
window.addEventListener("DOMContentLoaded", event => {
  const stopCyanMadness = () => {
    // get the value of the input field
    const inputValue = document.getElementById("stopper").value;
    // if value is anything other than 'STOP', change background color
    if (inputValue !== "STOP") {
      document.body.style.backgroundColor = "cyan";
    }
  };
  setTimeout(stopCyanMadness, 5000);
});
```

The code at the bottom of our function might look familiar. We used`setInterval`along with the Javascript Date object when we set up our current time clock. In this case we're using `setTimeout`, which runs `stopCyanMadness`after 5000 milliseconds, or 5 seconds after the page loads.

What we learned:

- How to add an event listener on a button click
- How to add an event listener to a checkbox
- Styling elements with Javascript vs. with CSS classes
- How to check the value of an input

Event Handling: Input Focus and Blur

Form inputs are one of the most common HTML elements users interact with on a website. By now, you should be familiar with how to listen for a *click event*and run a script. In this reading, we'll learn about a couple of other events on an input field and how to use them:

- Element: focus event
- Element: blur event

Listening for focus and blur events

According to MDN, the focus eventfires when an element, such as an input field, receives focus (i.e. when a user has clicked on that element).

The opposite of the focus event is the blur event. The blur event fires when an element has lost focus (i.e. when the user clicks out of that element).

Let's see these two events in action. We'll set up an HTML page that includes`<input type="text" id="fancypants">`. Then, we'll write JavaScript that changes the background color of the `fancypants`textbox to `#E8F5E9`when the focus is on the textbox and turns it back to its normal color when focus is elsewhere.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js">
  </head>
  <body>
   <input type="text" id="fancypants">
  </body>
</html>
```

Javascript

```
// script.js

window.addEventListener("DOMContentLoaded", event => {
  const input = document.getElementById("fancypants");

  input.addEventListener("focus", event => {
    event.target.style.backgroundColor = "#E8F5E9";
  });
  input.addEventListener("blur", event => {
    event.target.style.backgroundColor = "initial";
  });
});
```

In the code above, we changed the background color of the input on `focus`and changed it back to its initial value on `blur`. This small bit of functionality signals to users that they've clicked on or off of an input field, which is especially helpful and more user-friendly when there is a long form on the page. Now you can use `focus`and `blur`on your form inputs!

What we learned:

- How to listen for the focus event on inputs
- How to listen for the blur event on inputs

Event Handling: Form Validation

Everyone has submitted a form at some point. Form submissions are another common action users take on a website. We've all seen what happens if we put in values that aren't accepted on a form -- frustrating errors! Those errors prompt the user to input accepted form values before submission and are the first check to ensure valid data gets stored in the database.

Learning how to implement front-end validation before a user submits a form is an important skill for developers. In this reading, we'll learn how to check whether two password values on a form are equal and prevent the user from submitting the form if they're not.

Validate passwords before submitting a form

In order to validate passwords, we need a form with two password fields: a password field and a confirmation field. We'll also include two other fields that are common on a signup page: a name field and an email field. See the example below:

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js">
  </head>
  <body>
    <form class="form form--signup" id="signup-form">
      <input class="form__field" id="name" type="text" placeholder="Name" style="
      <input class="form__field" id="email" type="text" placeholder="Email" style
      <input class="form__field" id="password" type="text" placeholder="Password"
      <input class="form__field" id="confirm-password" type="text" placeholder="
      <button class="form__submit" id="submit" type="submit">Submit</button>
    </form>
  </body>
</html>
```

Now, we'll set up our `script.js`file with code that will:

- Listen for a form submission event
- Get the values of the two password fields and check for a match
- Alert the user if there's not a match, or submit the form

Javascript

```
// script.js
window.addEventListener("DOMContentLoaded", event => {
  // get the form element
  const form = document.getElementById("signup-form");

  const checkPasswordMatch = event => {
    // get the values of the pw field and pw confirm field
    const passwordValue = document.getElementById("password").value;
    const passwordConfirmValue = document.getElementById("confirm-password")
      .value;
    // if the values are not equal, alert the user
    // otherwise, submit the form
    if (passwordValue !== passwordConfirmValue) {
      // prevent the default submission behavior
      event.preventDefault();
      alert("Passwords must match!");
    } else {
      alert("The form was submitted!");
    }
  };
  // listen for submit event and run password check
  form.addEventListener("submit", checkPasswordMatch);
});
```

In the code above, we prevented the form submission if the passwords don't match using Event.preventDefault(). This method stops the default action of an event if the event is not explicitly handled. We then alerted the user that the form submission was prevented.

What we learned:

- Front-end form validation prevents invalid data from being recorded in the database.
- We use `Event.preventDefault()` to stop form submission.
- Users are typically notified when default behavior is prevented.

Event Handling: HTML Drag-And-Drop API

Dragging and dropping a page element is a fun and convenient way for users to interact with a Web page! HTML drag-and-drop interfaces are most commonly used for dragging files, such as PDFs or images, onto a page in a specified area known as a *drop zone*.

While less typical than button clicks and form submission events, HTML drag-and-drop is relevant to event handling because it uses the DOM event modeland drag eventsinherited from mouse events.

This reading will go over how to use the HTML Drag and Drop API to create "draggable" page elements and allow users to drop them into a drop zone.

Basic drag-and-drop functions

Let's go over how to set up basic drag-and-drop functionality, according to theMDN documentation. You'll need to mark an element as "draggable". Then, to do something with that dragging, you need

- A *dragstart*handler -- occurs when the user clicks the mouse and starts dragging
- A *drop*handler -- occurs when the user releases the mouse click and "drops" the element

You will also see how to use the `dragenter`and `dragleave`events to do some nice interactions.

The example

Here's an HTML document that you can copy and paste into a text editor if you want to follow along.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Red Square is a Drag</title>
  <style>
    #red-square {
      background-color: red;
      box-sizing: border-box;
      height: 100px;
      width: 100px;
    }
  </style>
</head>
<body>
  <div id="red-square"></div>
</body>
</html>
```

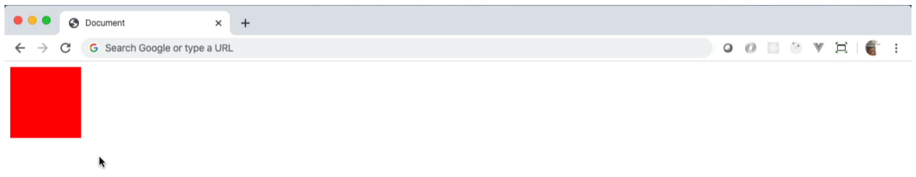
You don't need to know what `box-sizing`does. That'll be covered in future lessons. It's in this example to make sure the box looks ok when it's dragged.

The first step of drag and drop

The first step to making an element draggable is to add that attribute to the element itself. Change the red square `div` to have the `draggable`attribute with a value of "true".

```
<div id="red-square" draggable="true"></div>
```

Now, if you refresh your page, you can start dragging the red square. When you release it, it will "snap" back to it's original position.



Handling the start of a drag

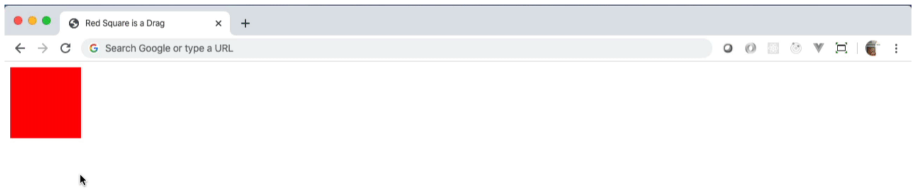
Now that the element is draggable, you need some JavaScript to handle the event of when someone starts dragging an element. This is there so that your code knows what's being dragged! Otherwise, how will it know what to do when the dragging ends?

The following code creates a handler for the `dragstart`event. Then, it subscribes to the red square's `dragstart`event with that event handler. The event handler, in this case, will add a class to the red square to make it show that it's being dragged. Then, it adds the value of the `id`of the element to the "data transfer" object. The "data transfer" object holds all of the information that will be needed when the dragging operation ends.

The `classList`object for the HTML element is just a way to add and remove CSS classes to and from a DOM object. You're not going to be tested over that bit of information for some weeks, so don't worry about remembering it. Just understand that using the `add`method on it *adds* the CSS class to the HTML element.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Red Square is a Drag</title>
  <script type="text/javascript">
    const handleDragStart = e => {
      e.target.classList.add('is-being-dragged');
      e.dataTransfer.setData('text/plain', e.target.id);
      e.dataTransfer.dropEffect = 'move';
    }
    window.addEventListener('DOMContentLoaded', () => {
      document
        .getElementById('red-square')
        .addEventListener('dragstart', handleDragStart);
    });
  </script>
  <style>
    #red-square {
      background-color: red;
      box-sizing: border-box;
      height: 100px;
      width: 100px;
    }

    .is-being-dragged {
      opacity: 0.5;
      border: 2px dashed white;
    }
  </style>
</head>
<body>
  <div id="red-square"></div>
</body>
</html>
```

If you update your version of the code, you can now see that the square's border gets all dashy when you drag it!



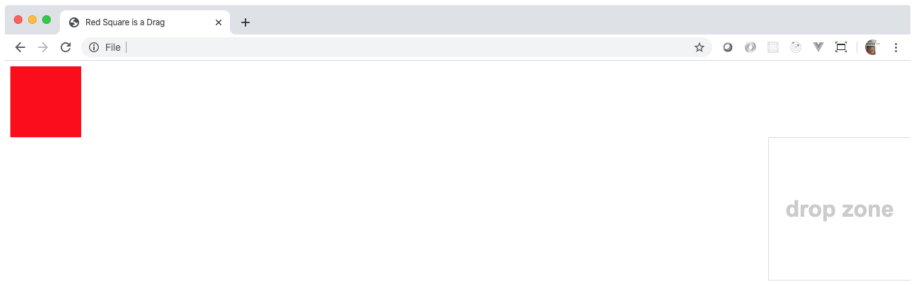
The drop zone is all clear!

A drop zone is just another HTML element. Put another `div` in your HTML and give it an id of `drop-zone`. It could, conceivably, look like this.

```
<body>
  <div id="red-square"></div>

  <div id="drop-zone">drop zone</div>
</body>
```

To make it visible, add some CSS. Note that most of that in there is to make it look pretty. You should understand the simple things like "background-color" and "color" and "font-size", "height" and "width". You won't yet be tested on any of those other properties. But, feel free to play around with them!

```
<style>
  #drop-zone {
    align-items: center;
    border: 1px solid #DDD;
    color: #CCC;
    display: flex;
    font-family: Arial, Helvetica, sans-serif;
    font-size: 2em;
    font-weight: bold;
    height: 200px;
    justify-content: center;
    position: absolute;
    right: 0;
    width: 200px;
  }

  #red-square {
    background-color: red;
    box-sizing: border-box;
    height: 100px;
    width: 100px;
  }

  .is-being-dragged {
    opacity: 0.5;
    border: 2px dashed white;
  }
</style>
```



Handle when the red square gets enters the drop zone (and leaves)

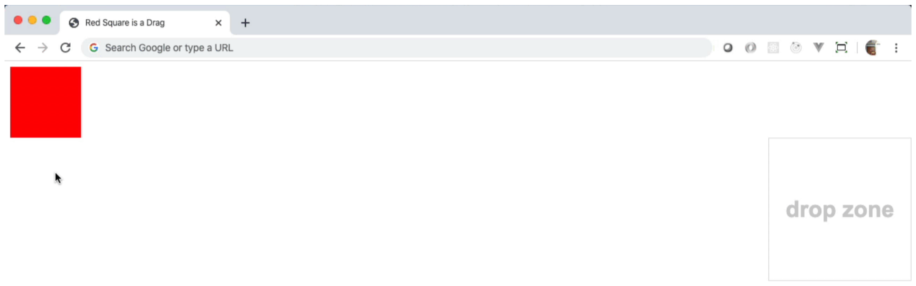
This is another couple of JavaScript event handlers, but this time, it will handle the `dragenter` and `dragleave` events on the drop zone element. You can replace the `<script>` tag in your HTML with this version, here, to handle those events. Note that the `handleDragEnter` event handler merely adds a CSS class to the drop zone. The `handleDragLeave` removes the CSS class. Then, in the `DOMContentLoaded` event handler, the last three lines gets a reference to the drop zone element and attaches the event handlers to it.

```
<script type="text/javascript">
  const handleDragStart = e => {
    e.target.classList.add('is-being-dragged');
    e.dataTransfer.setData('text/plain', e.target.id);
    e.dataTransfer.dropEffect = 'move';
  };

  const handleDragEnter = e => {
    e.target.classList.add('is-active-drop-zone');
  };

  const handleDragLeave = e => {
    e.target.classList.remove('is-active-drop-zone');
  };

  window.addEventListener('DOMContentLoaded', () => {
    document
      .getElementById('red-square')
      .addEventListener('dragstart', handleDragStart);

    const dropZone = document.getElementById('drop-zone');
    dropZone.addEventListener('dragenter', handleDragEnter);
    dropZone.addEventListener('dragleave', handleDragLeave);
  });
</script>
```

The CSS to make the item change looks like this. Just add the class to the `<style>` tag in the HTML.

```
.is-active-drop-zone {
  background-color: blue;
}
```

Just look at that drop zone turn blue with glee!



Do something with a "drop"!

Finally, the `drop` event of the drop target handles what happens when you let go of the draggable element over the drop zone. However, there's one small problem. From the MDN documentation on Drag Operations:

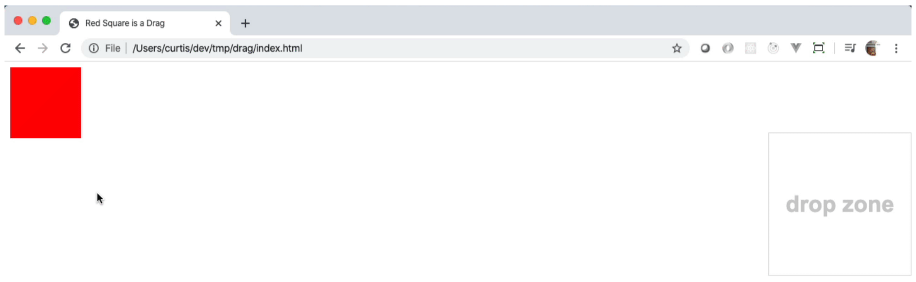
If the mouse is released over an element that is a valid drop target, that is, one that cancelled the last `dragenter` or `dragover` event, then the drop will be successful, and a drop event will fire at the target. Otherwise, the drag operation is cancelled, and no drop event is fired.

For this to work properly, you will *also* have to subscribe to the `drop` event for the drop zone. Then, in both the handlers for the `drop` and `dragenter` events, you'll need to cancel the event. Recall that in the last article you learned about the `preventDefault()` method on the event object. That's what you need to call in both the `drop` and `dragenter` event handlers to make the `drop` event fire.

That's a lot of words! Here is the final HTML for this little dragging example.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Red Square is a Drag</title>
  <script type="text/javascript">
    const handleDragStart = e => {
      e.target.classList.add('is-being-dragged');
      e.dataTransfer.setData('text/plain', e.target.id);
      e.dataTransfer.dropEffect = 'move';
    };

    const handleDragEnter = e => {
      // Needed so that the "drop" event will fire.
      e.preventDefault();
      e.target.classList.add('is-active-drop-zone');
    };

    const handleDragLeave = e => {
      e.target.classList.remove('is-active-drop-zone');
    };

    const handleDragOver = e => {
      // Needed so that the "drop" event will fire.
      e.preventDefault();
    };

    const handleDrop = e => {
      const id = e.dataTransfer.getData('text/plain');
      const draggedElement = document.getElementById(id);
      draggedElement.draggable = false;
      e.target.appendChild(draggedElement);
    };

    window.addEventListener('DOMContentLoaded', () => {
      document
        .getElementById('red-square')
        .addEventListener('dragstart', handleDragStart);
    const dropZone = document.getElementById('drop-zone');
    dropZone.addEventListener('drop', handleDrop);
    dropZone.addEventListener('dragenter', handleDragEnter);
    dropZone.addEventListener('dragleave', handleDragLeave);
    dropZone.addEventListener('dragover', handleDragOver);
  });
</script>
<style>
  #drop-zone {
    align-items: center;
    border: 1px solid #DDD;
    color: #CCC;
    display: flex;
    font-family: Arial, Helvetica, sans-serif;
    font-size: 2em;
    font-weight: bold;
    height: 200px;
    justify-content: center;
    position: absolute;
    right: 0;
    top: 100px;
    width: 200px;
  }

  #red-square {
    background-color: red;
    box-sizing: border-box;
    height: 100px;
    width: 100px;
  }

  .is-being-dragged {
    opacity: 0.5;
    border: 8px dashed white;
  }

  .is-active-drop-zone {
    background-color: blue;
    color:
  }
```

```
  </style>
  </style>
</head>
<body>
  <div id="red-square" draggable="true"></div>
  <div id="drop-zone">drop zone</div>
</body>
</html>
```



What you learned:

There is *a lot*going on, here. Here's a quick review of what you should get from this article.

- HTML has a Drag and Drop API that you can use to do dragging and dropping in an application.
- To make it so that a person can drag an HTML element around, you add the`draggable="true"`attribute/value pair to the element (or elements) that you want to drag.

- You can subscribe to one event on the thing you want to drag, the `dragstart`event. That event will allow you to customize the element and save data to the`dataTransfer`object.
- You can subscribe to four events for the "drop zone" element(s): `dragenter`,`dragover`, and `dragleave`. If you want the `drop`event to work, you *must*subscribe to both `dragenter`and `dragover`and cancel the event using the`preventDefault()`method of the event.

Event Handling: Click Events With Event.target

Imagine a user is viewing a Web page showing 300 different products. The user carefully studies the page, makes a selection, and clicks on one of the 300 products. Could we find out through code which element was clicked on? Yes!

Previously we learned how to handle a click event using an element's ID. However, what if we don't know the ID of the clicked element before it's clicked? There is a simple property we can use to discover on which element the click event occurred: `event.target`.

According to the MDN doc on event.target, "the `target`property of the`Event`interface is a reference to the object that dispatched the event. It is different from event.currentTargetwhen the event handler is called during the bubbling or capturing phase of the event." Essentially:

- `event.target`refers to the element on which the event occurred (e.g. a clicked element).
- `event.currentTarget`refers to the element to which the event handler has been attached, which could be the parent element of the `event.target`element. (*Note: We'll talk about this in more detail in the reading on The Bubbling Principle.*)

It is common practice for developers to use `event.target`to reference the element on which the event occurs in an event handling function. Let's practice using this handy property to get the ID of a clicked element.

Use event.target to console.log the ID of a clicked div

Let's say we had an HTML page with 10 `divs`, each with a unique ID, like below. We want to click on any one of these divs and print the clicked div's ID to the console.

HTML

```html
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="example.css" />
    <script type="text/javascript" src="example.js"></script>
  </head>
  <body>
    <div id="div-1" class="box">1</div>
    <div id="div-2" class="box">2</div>
    <div id="div-3" class="box">3</div>
    <div id="div-4" class="box">4</div>
    <div id="div-5" class="box">5</div>
    <div id="div-6" class="box">6</div>
    <div id="div-7" class="box">7</div>
    <div id="div-8" class="box">8</div>
    <div id="div-9" class="box">9</div>
    <div id="div-10" class="box">10</div>
  </body>
</html>
```

In our linked **example.css**file, we'll add style to the `.box`class to make our `divs`easier to click on:

CSS

```css
.box {
  border: 2px solid gray;
  height: 50px;
  width: 50px;
  margin: 5px;
}
```

Now, we'll write Javascript to print the clicked div's ID to the console. Again, we want to wait for the necessary DOM elements to load before running our script using `DOMContentLoaded`. Then, we'll listen for a click event and `console.log`the clicked element's ID.

Javascript

```
// example.js

// Wait for the DOM to load
window.addEventListener("DOMContentLoaded", event => {
  // Add a click event listener on the document's body
  document.body.addEventListener("click", event => {
    // console.log the event target's ID
    console.log(event.target.id);
  });
});
```

If you open up your HTML in a browser, you should see the 10 `divs`. Click on any one of them. Open up the browser console by right-clicking, selecting*Inspect*, and opening the *Console*tab. The ID of the div you clicked should be printed to the console. Click on the other divs randomly, and make sure their IDs print to the console as well.

What we learned:

- The definition of `event.target`
- How `event.target`differs from `event.currentTarget`
- How to console.log the ID of a clicked element using `event.target`

Event Handling: The Bubbling Principle

Bubbles are little pockets of air that make for an amusing time in the bath. Sometimes, though, bubbles can be annoying -- like when they suddenly pop, or when there are too many and they're overflowing! We can think about Javascript events and their handlers as bubbles that rise up through the murky waters of the DOM until they reach the surface, or the top-level DOM element.

It's important for developers to understand The Bubbling Principle and use it to properly handle events and/or to stop events from bubbling up to outer elements and causing unintended effects.

What is the bubbling principle?

According to this handy bubbling explaineron Javascript.info, The Bubbling Principle means that *when an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors*.Consider the following example HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript">
      window.addEventListener("DOMContentLoaded", event => {
        document.body.addEventListener("click", event => {
          console.log(event.target.id);
        });
      });
    </script>
  </head>
  <body>
    <div onclick="console.log('The onclick handler!')">
      <p id="paragraph">
        If you click on this P, the onclick event for the DIV actually runs.
      </p>
    </div>
  </body>
</html>
```

In the `<script>`, you can see the event listener for `DOMContentLoaded`, and inside it, another listener for a `click`event on the `<body>`element of the document accessed through the special property `document.body`. (You could also use `document.querySelector('body')`, too.) By now, we should be used to listening for click events in our scripts. However, there's another way to run a function on a `click`event as an attribute of the `div` in the body of the HTML, a way that **you should never ever ever use in real production code**!

Check out that `onclick`attribute with some JavaScript code to print out a message about the so-called onclick handler. For almost ever event type like`click`or `change`or `keypress`, you can put an attribute by prefixing the event name with the word "on". However, **you should never ever ever use that in real production code**!

Save the above HTML in a file, and run that file in a browser. Open up the browser console (*right-click* -> *Inspect* -> *Console*), click on the `<p>`element, and observe what happens. The message "The onclick handler" should appear, then you should see the id `paragraph`printed to the console.

What happened here? The `console.log`shows that an event happened on the `<p>`(i.e. the `event.target`), and yet the `onclick`handler on the `<div>`also fired -- meaning that the click event on the `<p>`bubbled up to the `<div>`and fired its `onclick`event!

Once again, here's the deal:

Don't ever use the `on`-event-name attribute version of an event handler. Instead, always use the `addEventListener`method of the DOM object that *you get from something like* `document.getElementById` *or* `document.querySelector`.

An event bubbling example

To visualize event bubbling, it might be helpful to watch this short and fun YouTube video on *bubbles inside bubbles inside bubbles*.

Bubble Inside a Bubble Video

We can think of events that happen on nested DOM elements as these nested bubbles. An event that happens on the innermost element bubbles up to its parent element, and that parent's parent element, and so on up the chain. Let's look at another example that demonstrates bubbling.

HTML

```
<!DOCTYPE html>
<html>
  <body>
    <main>
      <div>
        <p>This is a paragraph in a div in a main in a body in an html</p>
      </div>
    </main>

    <script>
      function handler(e) {
        console.log(e.currentTarget.tagName);
      }
      document.querySelector('main').addEventListener('click', handler);
      document.querySelector('div').addEventListener('click', handler);
      document.querySelector('p').addEventListener('click', handler);
    </script>
```

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

If you save this HTML file, open it in a browser, and click on the `<p>`, three different messages should appear in the console: first "P", second "DIV", and third "MAIN". The click event bubbled upwards from the `<p>`element to the`<div>`and finally to the `<main>`.

We could think of this succession of events as bubbles popping. The innermost bubble (the `<p>`element) *popped*(i.e. displayed an alert), which caused its parent's bubble to pop, which caused its parent's bubble to pop. Since there aren't any `onclick`handlers above the `<main>`nothing else happens on the page, but the bubbles would travel all the way up the DOM until they reached the top (`<html>`) looking for event handlers to run.

Stopping event bubbling with stopPropagation()

As stated in the introduction, event bubbling can cause annoying side effects. This MDN doc on Event bubbling and captureexplains what would happen if a user clicked on a `<video>`element that has a parent `<div>`with a show/hide toggle effect. On a click, the video would disappear along with its parent div!

How can you stop this unintended behavior from occurring? The answer is with theevent.stopPropagation()method which stops the bubbling from continuing up the parent chain. Here's what it would look like on the `<video>`element:

Javascript

```
document
  .querySelector('video')
  .addEventListener('click', event => {
    event.stopPropagation();
    video.play();
  });
```

Event delegation

While event bubbling can sometimes be annoying, it can also be helpful. The bubbling effect allows us to make use of **event delegation**, which means that we can delegate events to a single element/handler -- a parent element that will handle all events on its children elements.

Say you had an unordered list (`<ul>`) element filled with several list item (`<li>`) elements, and you want to listen for click events on each list item. Instead of attaching a click event listener on each and every list item, you could conveniently attach it to the parent unordered list, like so:

HTML

```
<ul id="my-list">
  <li>This is list item 1.</li>
  <li>This is list item 2.</li>
  <li>This is list item 3.</li>
  <li>This is list item 4.</li>
  <li>This is list item 5.</li>
</ul>
<script>
  document
    .getElementById('my-list')
    .addEventListener('click', e => {
       // will print out "This is list item X"
       // depending on which list item is clicked
      console.log(e.target.innerHTML);

      // always prints "my-list"
      console.log(e.currentTarget.id);
    });
</script>
```

This example is a lot like the first example you saw with the `<p>`inside of a`<div>`, where the click on the `<p>`bubbled up to the `<div>`. In the above example, a click on any `<li>`will bubble up to its parent, the `<ul>`.

When clicked on, a single `<li>`element becomes the event.target-- the object that dispatched the event. The `<ul>`element is theevent.currentTarget-- the element to which the event handler has been attached.

Now that you know how to handle events responsibly, go frolic in the bubbles!

What we learned:

- The definition of The Bubbling Principle
- Examples of event bubbling
- How to stop events from bubbling
- How to use bubbling for event delegation

JSON Learning Objectives

The objective of this lessonis to familiarize you with the JSON format and how to serialize to and deserialize from that format.

The learning objectivesfor this lesson are that you can:

1. Identify and generate valid JSON-formatted strings
2. Use `JSON.parse`to deserialize JSON-formatted strings
3. Use `JSON.stringify`to serialize JavaScript objects
4. Correctly identify the definition of "deserialize"
5. Correctly identify the definition of "serialize"

This lesson is relevantbecause JSON is the *lingua franca* of data interchange.

Storage Lesson Learning Objectives

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Write JavaScript to store the value "I <3 falafel" with the key "eatz" in the browser's local storage.
2. Write JavaScript to read the value stored in local storage for the key "paper-trail".

Cookies and Web Storage

As we've learned in previous sections, most data on the Web is stored in a database on a server, and we use the browser to retrieve this data. However, sometimes data is stored locally for the purposes of persisting throughout an entire session or until a specified expiration date.

In this reading, we'll go over using **cookies**to store data versus using the**Web Storage API** and the use cases for each storage method.

Cookies

Cookies have been around forever, and they are still a widely used method to store information about a site's users.

What is a cookie?

A cookie is a small file stored on a user's computer that holds a bite-sized amount of data, under 4KB. Cookies are included with HTTP requests. The server sends the data to a browser, where it's typically stored and then sent back to the server on the next request.

What are cookies used for?

Cookies are used to store stateful information about a user, such as their personal information, their browser habits or history, or form input information they have filled out. A common use case for cookies is storing a *session cookie*on user login/validation. Session cookies are lost once the browser window is closed. To make sure the cookie persists beyond the end of the session, you could set up a *persistent cookie*with a specified expiration date. A use case for a persistent cookie is an e-commerce website that tracks a user's browsing or buying habits.

How to create a cookie in Javascript:

As we've previously covered, the `document`interface represents the web page loaded in a user's browser. Since cookies are stored on a user's browser, it makes sense that the `document`object also allows us to get/set cookies on a user's browser:

```
const firstCookie = "favoriteCat=million";
document.cookie = firstCookie;
const secondCookie = "favoriteDog=bambi";
document.cookie = secondCookie;
document.cookie; // Returns "favoriteCat=million; favoriteDog=bambi"
```

Using the following syntax will create a new cookie:

```
document.cookie = aNewCookieHere;
```

If you want to set a second cookie, you would assign a new key value pair using the same syntax a second time. Make sure to set the cookie to a string formatted like a key-value pair:

```
const firstCookie = "favoriteCat=million";
document.cookie = firstCookie;
document.cookie; // Returns "favoriteCat=million"
```

Formatting your string like we do in the `firstCookie`variable above sets the cookie `value`with a defined key, known as the cookie's `name`, instead of an empty `name`. Refer to the MDN docs on Document.cookiefor more examples.

You can view all the cookies a website is storing about you by using the Developer Tools. On **Google Chrome**, see the **Application tab**, and on**Firefox**, see the **Storage tab**.

Deleting a cookie:

We can delete our own cookies using JavaScript by setting a cookie's expiration date to a date in the past, causing them to expire:

```
const firstCookie = "favoriteCat=million";
document.cookie = firstCookie;
document.cookie; // Returns "favoriteCat=million"

// specify the cookies "name" (the key) with an "=" and set the  expiration
// date to the past
document.cookie = "favoriteCat=; expires = Thu, 01 Jan 1970 00:00:00 GMT";
document.cookie; // ""
```

We can also delete cookies using the Developer Tools!

Navigate to a website, such as Amazon, and add an item to your cart. Open up the Developer Tools in your browser and delete all the cookies. In Chrome, you can delete cookies by highlighting a cookie and clicking the delete button. In Firefox, you can right-click and delete a cookie. If you've deleted all the cookies in your Amazon cart, and you refresh the page, you should notice your cart is now empty.

Web Storage API

Cookies used to be the only way to store data in the browser, but with HTML5 developers gained access to the Web Storage API, which includes**localStorage**and **Session Storage**. Here are the differences between the two, according to MDN:

`sessionStorage`:

- Stores data only for a *session*, or until the browser window or tab is closed
- Never transfers data to the server
- Has a storage limit of 5MB (much larger than a cookie)

The following example from MDNshows how we can use sessionStorage to autosave the contents of a text field and restore the contents of that text field if the browser is accidentally refreshed.

```
// Get the text field that we're going to track
let field = document.getElementById("field");

// See if we have an autosave value
// (this will only happen if the page is accidentally refreshed)
if (sessionStorage.getItem("autosave")) {
  // Restore the contents of the text field
  field.value = sessionStorage.getItem("autosave");
}

// Listen for changes in the text field
field.addEventListener("change", function () {
  // And save the results into the session storage object
  sessionStorage.setItem("autosave", field.value);
});
```

`localStorage`:

- Stores data with no expiration date and is deleted when clearing the browser cache
- Has the maximum storage limit in the browser (much larger than a cookie)

Like with `sessionStorage`, we can use the `getItem()`and `setItem()`methods to retrieve and set `localStorage`data. The following example from MDNwill:

- Check whether `localStorage`contains a data item called `bgcolor`using`getItem()`.
- If `localStorage`contains `bgcolor`, run a function called `setStyles()`that grabs the data items using `Storage.getItem()`and use those values to update page styles.
- If it doesn't, run a function called `populateStorage()`, which uses`Storage.setItem()`to set the item values, then run `setStyles()`.

```
if (!localStorage.getItem("bgcolor")) {
  populateStorage();
}
setStyles();

const populateStorage = () => {
  localStorage.setItem("bgcolor", document.getElementById("bgcolor").value);
  localStorage.setItem("font", document.getElementById("font").value);
  localStorage.setItem("image", document.getElementById("image").value);
};

const setStyles = () => {
  var currentColor = localStorage.getItem("bgcolor");
  var currentFont = localStorage.getItem("font");
  var currentImage = localStorage.getItem("image");

  document.getElementById("bgcolor").value = currentColor;
  document.getElementById("font").value = currentFont;
  document.getElementById("image").value = currentImage;

  htmlElem.style.backgroundColor = "#" + currentColor;
  pElem.style.fontFamily = currentFont;
  imgElem.setAttribute("src", currentImage);
};
```

When would we use the Web Storage API?

Since web storage can store more data than cookies, it's ideal for storing multiple key-value pairs. Like with cookies, this data can be saved only as a string. With localStorage, the data is stored locally on a user's machine, meaning that it can only be accessed client-side. This differs from cookies which can be read both server-side and client-side.

There are a few common use cases for Web storage. One is storing information about a shopping cart and the products in a user's cart. Another is saving input data on forms. You could also use Web storage to store information about the user, such as their preferences or their buying habits. While we would normally use a cookie to store a user's ID or a session ID after login, we could use localStorage to store extra information about the user.

You can view what's in local or session storage by using the Developer Tools. On**Google Chrome**, see the **Application tab**, and on **Firefox**, see the**Storage tab**.

What we learned:

- What cookies are and when to use them
- Differences between cookies and localStorage
- Use cases for cookies and localStorage

Jason? No, JSON!

Jason is an ancient Greek mythological hero who went traipsing about the known world looking for "the golden fleece".

JSON is an open-standard file format that "uses human-readable text to transmit objects consisting of key-values pairs and array data types."

We're going to ignore Jasonand focus solely on JSON for this reading so that you can, by the end of it, know what JSON is and how to work with it.

JSON is a format!

This is the most important thing that you can get when reading this article. In the same way that HTML is a format for hypertext documents, or DOCX is a format for Microsoft Word documents, JSON is just a format for data. It's just text. It doesn't "run" like JavaScript does. It is just text that contains data that both machines and humans can understand. If you ever hear someone say "a JSON object", then you can rest assured that phrase doesn't make any sense whatsoever.

JSON is just a string. It's just text.

That's so important, here it is, again, but in a fancy quote box.

JSON is just a string. It's just text.

Why all the confusion?

The problem is, JSON *looks* a lot like JavaScript syntax. Heck, it's even named**JavaScript Object Notation**. That's likely because the guy who invented it,Douglas Crockford, is an avid JavaScripter. He's the author of JavaScript: The Good Partsand was the lead JavaScript Architect at Yahoo! back when Yahoo! was a real company.

At that time, like in the late 1990s and early 2000s, there were a whole bunch of competing formats for how computers would send data between one another. The big contender at the time is a format called XML, or the *eXtensible Markup Language*. It looks a lot like HTML, but has far stricter rules than HTML. Douglas didn't like XML because it took a lot of bytes to send the data (and this was a pre-broadband/pre-3G world). Worse, XML is not a friendly format to read if you're human. So, he set out to come up with a new format based on the way JavaScript literals work.

"Remind me about JavaScript literals..."

Just to refresh your memory, a *literal*in JavaScript is a *value that you literally just type in*. If you type `7`into a JavaScript file, when it runs, the JavaScript interpreter will see that character `7` and say to itself, "Hey self, the programmer literally typed the number seven so that must mean they want the value 7."

Here's a table of some literals that you may type into a program.

What you want to type	The JavaScript literal
The value that means "true"	`true`
The number of rows in this table	`6`

What you want to type	The JavaScript literal
A bad approximation of π	`3.14`
An array that contains some US state names	`["Ohio", "Iowa"]`
An object that represents Roberta	`{ person: true, name: "Roberta" }`

Back to Douglas Crockford, inventor of JSON. Douglas thought to himself, *why can't I create a format that has that simplicity so that I can write programs that can send data to each other in that format?*Turns out, he could, and he did.

Boolean, numeric, and null values

The following table shows you what the a JavaScript literal is in the JSON format. Notice that *everything*in the JSON column is actually a string!

JavaScript literal value	JSON representation in a string
`true`	`"true"`
`false`	`"false"`
`12.34`	`"12.34"`
`null`	`"null"`

String literals in JSON

Say you have the following string in JavaScript.

```
'this is "text"'
```

When that gets converted into the JSON format, you will see this:

```
"this is \"text\""
```

First, it's important to notice one thing: JSON always uses double quotes for strings. Yep, that's worth repeating.

JSON always uses double-quotes to mark strings.

Notice also that the quotation marks (") are "escaped". When you write a string surrounded by quotation-marks like "escaped", everything's fine. But, what happens when your string needs to include a quotation mark?

```
// This is a bad string with quotes in it
"Bob said, "Well, this is interesting.""
```

Whatever computer is looking at that string gets really confused because once it reads that first quotation mark it's looking for another quotation mark to show where the string ends. For computers, the above code looks like this to them.

```
"Bob said, "                // That's a good string
Well, this is interesting   // What is THIS JUNK????
""                          // That's a good string
```

You need a way to indicate that the quotation marks around the phrase that Bob says should belong *in*the string, not as a way to show where the string starts or stops. The way that language designers originally addressed this was by saying

If your quotation mark delimited string has a quotation mark in it, put a backslash before the interior quotation mark.

Following that rule, you would correctly write the previous string like this.

```
"Bob said, \"Well, this is interesting.\""
```

Check out all of the so-called JavaScript string escape sequencesover on MDN.

What happens if you had text that spanned more than one line? JSON only allows strings to be on one line, just like old JavaScript did. Let's say you just wrote an American sentence that you want to submit to a contest.

```
She woke him up with
her Ramones ringtone "I Want
to be Sedated"
```

(from American Sentences by Paul E. Nelson)

If you want to format that in a string in JSON format, you have to escape the quotation marks *and* the new lines! The above would look like this:

```
She woke him up with\nher Ramones ringtone \"I Want\nto be Sedated\"
```

The new lines are replaced with "\n".

Array values

The way that JSON represents an array value is using the same literal notation as JavaScript, namely, the square brackets `[]`. With that in mind, can you answer the following question before continuing?

What is the JSON representation of an array containing the numbers one, two, and three?

Well, in JavaScript, you would type `[1, 2, 3]`.

If you were going to type the corresponding JSON-formatted string that contains the representation of the same array, you would type `"[1, 2, 3]"`. Yep, pretty much the same!

Object values

Earlier, you saw that example of an object that represents Roberta as

```
{ person: true, name: "Roberta" }
```

The main difference between objects in JavaScript and JSON is that the keys in JSON *must* be surrounded in quotation marks. That means the above, in a JSON formatted string, would be:

```
"{ \"person\": true, \"name\": \"Roberta\" }"
```

Some terminology

When you have some data and you want to turn it into a string (or some other kind of value like "binary") so your program can send it to another computer, that is the process of **serialization**.

When you take some text (or something another computer has sent to your program) and turn it into data, that is the process of **deserialization**.

Using the built-in JSON object

In modern JavaScript interpreters, there is a `JSON`object that has two methods on it that allows you to convert JSON-formatted strings into JavaScript objects and JavaScript object into JSON-formatted strings. They are:

- `JSON.stringify(value)`will turn the value passed into it into a string.
- `JSON.parse(str)`will turn a JSON-formatted string into a JavaScript object.

So, it shouldn't come as much of a surprise how the following works.

```
const array = [1, 'hello, "world"', 3.14, { id: 17 }];
console.log(JSON.stringify(array));
// prints [1, "hello, \"world\"", 3.14, {"id":17}]
```

It shouldn't surprise you that it works in the opposite direction, too.

```
const str = '[1,"hello, \\"world\\"",3.14,{"id":17}]';
console.log(JSON.parse(str));
// prints an array with the following entries:
//   0: 1
//   1: "hello, \"world\""
//   2: 3.14
//   3: { id: 17 }
```

You may ask yourself, "What's up with that double backslash thing going on in the JSON representation?". It has to do with that escaping thing. When JavaScript reads the string the first time to turn it into a `String`object in memory, it will escape the backslashes. Then, when `JSON.parse`reads it, it will still need backslashes in the string. This is all really confusing, escaped strings and double backslashes. There's an easy solution for that.

You will almost never write raw JSON

Yep. But, you do need to be able to recognize it and read it. What you'll likely end up doing in your coding is creating values and using `JSON.stringify`to create JSON-formatted strings that represent those values. Or, you'll end up calling a data service which will return JSON-formatted content to your code which you will then use `JSON.parse`on to convert the string into a JavaScript object.

Brain teaser

Now that you know JSON is a format for data and is just text, what will the following print?

```
const a = [1, 2, 3, 4, 5];
console.log(a[0]);

const s = JSON.stringify(a);
console.log(s[0]);

const v = JSON.parse(s);
console.log(v[0]);
```

What you just learned

With some more practiced, of course, you will be able to do all of these really well. However, right now, you should be able to

1. Identify and generate valid JSON-formatted strings
2. Use `JSON.parse`to deserialize JSON-formatted strings
3. Use `JSON.stringify`to serialize JavaScript objects
4. Correctly identify the definition of "deserialize"
5. Correctly identify the definition of "serialize"

Using Web Storage To Store Data In The Browser

Like cookies, the Web Storage APIallows browsers to store data in the form of key-value pairs. Web Storage has a much larger storage limit than cookies, making it a useful place to store data on the client side.

In the cookies reading, we reviewed the two main mechanisms of Web Storage:`sessionStorage`and `localStorage`. While `sessionStorage`persists for the duration of the session and ends when a user closes the browser, `localStorage`persists past the current session and has no expiration date.

One typical use case for local storage is caching data fetched from a server on the client side. Instead of making multiple network requests to the server to retrieve data, which takes time and might slow page load, we can fetch the data once and store that data in local storage. Then, our website could read the persisting data stored in localStorage, meaning our website wouldn't have to depend on our server's response - even if the user closes their browser!

In this reading, we'll go over how to store and read a key-value pair in local storage.

Storing data in local storage

Web Storage exists in the window as an object, and we can access it by usingWindow.localStorage. As we previously reviewed, with window properties we can omit the *"window"*part and simply use the property name, `localStorage`.

We can set a key-value pair in local storage with a single line of code. Here are a few examples:

```
localStorage.setItem('eatz', 'I <3 falafel');
localStorage.setItem('coffee', 'black');
localStorage.setItem('doughnuts', '["glazed", "chocolate", "blueberry",
"cream-filled"]');
```

The code above calls the `setItem()`method on the Storage object and sets a key-value pair. Examples: `eatz`(key) and `I <3 falafel`(value), `coffee`(key) and `black`(value), and `doughnut`(key) and `["glazed", "chocolate", "blueberry", "cream-filled"]`(value). Both the key and the value must be strings.

Reading data in local storage

If we wanted to retrieve a key-value pair from local storage, we could use `getItem()`with a key to find the corresponding value. See the example below:

```
localStorage.setItem('eatz', 'I <3 falafel');
localStorage.setItem('coffee', 'black');
localStorage.setItem('doughnuts', '["glazed", "chocolate", "blueberry",
"cream-filled"]');

const eatz = localStorage.getItem('eatz');
const coffee = localStorage.getItem('coffee');
const doughnuts = localStorage.getItem('doughnuts');

console.log(eatz); // 'I <3 falafel'
console.log(coffee); // 'black'
console.log(doughnuts); // '["glazed", "chocolate", "blueberry", "cream-filled"]
```

The above code reads the item with a key of `eatz`, the item with a key of`doughnut`, and the item with a key of `coffee`. We stored these in variables for handy use in any function we write.

Check the MDN docs on localStoragefor other methods on the Storage object to remove and clear all key-value pairs.

JSON and local storage

When we store and read data in local storage, we're actually storing JSONobjects. JSON is text format that is independent from JavaScript but also resembles JavaScript object literal syntax. It's important to note that JSON exists as a *string*.

Websites commonly get JSON back from a server request in the form of a text file with a `.json`extension and a MIME type of `application/json`. We can use JavaScript to parse a JSON response in order to work with it as a regular JavaScript object.

Let's look at the `doughnuts`example from above:

```
localStorage.setItem('doughnuts', '["glazed", "chocolate", "blueberry",
"cream-filled"]');
const doughnuts = localStorage.getItem('doughnuts');
console.log(doughnuts + " is a " + typeof doughnuts);
// prints '["glazed", "chocolate", "blueberry", "cream-filled"] is a string'
```

If we ran the code above in the browser console, we'd see that `doughnuts`is a string value because it's a JSON value. However, we want to be able to store`doughnuts`as an *array*, in order to iterate through it or map it or any other nifty things we can do to arrays.

We can construct a JavaScript value or object from JSON by parsing it:

```
const doughnuts = JSON.parse(localStorage.getItem('doughnuts'));
```

We used JSON.parse()to parse the string into JavaScript. If we printed the parsed value of `doughnuts`to the console, we'd see it's a plain ol' JavaScript array!

See the MDN doc on Working with JSON for more detail about using JSON and JavaScript.

What you learned:

- Why we use local storage
- How to store data in local storage
- How to read data in local storage
- How storage objects are JSON that we need to parse