JavaScript for Front-End

Learn by doing step by step exercises.

Includes downloadable class files that work on Mac & PC.

EDITION 4.1

noble desktop

Published by:
Noble Desktop
185 Madison Ave, 3rd Floor
New York, NY 10016
nobledesktop.com

Copyright © 2009–2022 Noble Desktop NYC LLC
Publish Date: 5-32-2022

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Downloading the Class Files

Thank You for Purchasing a Noble Desktop Course Workbook!

These instructions tell you how to install the class files you'll need to go through the exercises in this workbook.

Downloading & Installing Class Files

1. Navigate to the **Desktop**.
2. Create a **new folder** called **Class Files** (this is where you'll put the files after they have been downloaded).
3. Go to **nobledesktop.com/download**
4. Enter the code **js-2205-31**
5. If you haven't already, click **Start Download**.
6. After the **.zip** file has finished downloading, be sure to unzip the file if it hasn't been done for you. You should end up with a **JavaScript Class** folder.
7. Drag the downloaded folder into the **Class Files** folder you just made. These are the files you will use while going through the workbook.
8. If you still have the downloaded .zip file, you can delete that. That's it! Enjoy.

If You've Done Other Noble Desktop Coding Books

If you've setup Visual Studio Code in other Noble Desktop workbooks, there's nothing new for you to do so you're all set up.

Choosing a Code Editor to Work In

You probably already have a preferred code editor, such as **Visual Studio Code**, **Sublime Text**, etc. You can use whatever code editor you want, but we highly recommend Visual Studio Code.

Installing Visual Studio Code

Visual Studio Code is a great code free editor for Mac and Windows.

1. Visit **code.visualstudio.com** and download **Visual Studio Code**.
2. To install the app:
   - Mac users: Drag the downloaded app into your **Applications** folder.
   - Windows users: Run the downloaded installer.
     During installation check on the **Add "Open with Code" ...** options.

Setting Up Visual Studio Code

There are a few things we can do to make Visual Studio Code more efficient.

Setting Preferences

1. Launch **Visual Studio Code**.
2. In Visual Studio Code do the following:
   - Mac users: Go into the **Code** menu and choose **Preferences > Settings**.
   - Windows users: Go into the **File** menu and choose **Preferences > Settings**.
3. In the search field type: **wrap**
4. Find **Editor: Word Wrap** and change its setting from **off** to **on**

   NOTE: This ensures you'll be able to see all the code without having to scroll to the right (because it will wrap long lines to fit to your screen size).

Setting Up Open in Browser

Visual Studio Code does not include a quick way to preview HTML files in a web browser (which you'll do a lot). We can add that feature by installing an extension:

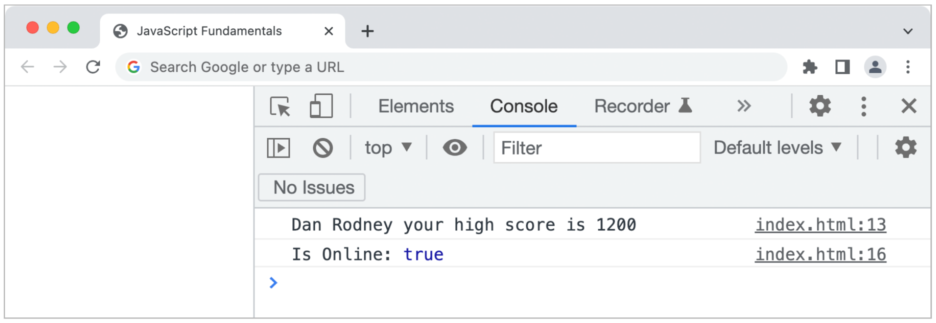
1. On the left side of the Visual Studio Code window, click on the **Extensions** icon.
2. In the search field type: **open in browser**
3. Under the **open in browser** click **Install**.

Defining a Keystroke for Wrapping Code

Visual Studio Code has a feature to wrap a selection with some code. This is very useful so we'll want to use it frequently. There's no keystroke for it by default, so let's define one:

1. In Visual Studio Code do the following:
   - Mac users: Go into the **Code** menu and choose **Preferences > Keyboard Shortcuts**.
   - Windows users: Go into the **File** menu and choose **Preferences > Keyboard Shortcuts**.
2. In the search field type: **wrap with**
3. Double–click on **Emmet: Wrap with Abbreviation** and then:
   - Mac users: Hold **Option** and press **W** then hit **Return**.
   - Windows users: Hold **Alt** and press **W** then hit **Enter**.
4. You're done, so close the **Keyboard Shortcuts** tab.

Exercise Preview



```
Dan Rodney your high score is 1200        index.html:13
Is Online: true                           index.html:16
```

Exercise Overview

Before you see real-life examples of how JavaScript is used, you must first learn some foundational concepts and syntax. These core concepts will be important building blocks for more complex scripts.

Getting Started

1. Launch your code editor (**Visual Studio Code**, **Sublime Text**, etc.). If you are in a Noble Desktop class, launch **Visual Studio Code**.
2. In your code editor, close any files you may have open.
3. For this exercise we'll be working with the **JavaScript-Fundamentals** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
4. In your code editor, open **index.html** from the **JavaScript-Fundamentals** folder.
5. Preview **index.html** in Chrome. (We're using Chrome because we'll be using its DevTools.)

   NOTE: If you're using Visual Studio Code with the **open in browser** extension installed, hit **Option–B** (Mac) or **Alt–B** (Windows) to open the saved HTML document in a browser.

   If you're using another code editor or don't have that keystroke set up, you can open the pages from your browser window to preview it. In your browser, hit **Cmd–O** (Mac) or **Ctrl–O** (Windows). Navigate to the **JavaScript Class** folder and then **JavaScript-Fundamentals** and double–click on **index.html**.
6. Notice this is a blank HTML page. We don't actually need any page content to start learning JavaScript.
7. Leave this page open in the browser as you work, so you can reload it to see the changes you make in the code.

Adding JavaScript to a Webpage

JavaScript code needs to be wrapped in a **script** tag, which we'll put in the **head** section.

1. Switch back to **index.html** in your code editor.
2. Add the following bold code before the closing **`</head>`** tag:

```
   <title>JavaScript Fundamentals</title>
   <script></script>
</head>
```

3. Inside the **script** tag, add the following bold code:

```
<script>
   alert('Hello');
</script>
```

NOTE: **alert(data)** is a JavaScript **method** that displays a dialog containing the data in the parentheses. Methods can be thought of as the verbs of JavaScript.

Methods & Arguments

methods are actions, they do things. The action performed by the `alert()` method is to open a dialog. Methods are written as **methodName** followed by **parentheses**, into which option(s) called **arguments** are often passed.

Code format: **`method(argument)`**

arguments are the data passed into the parentheses of a method. In the **`alert()`** method above, the argument is **`'Hello'`**

4. Save the file and reload the page in your browser.

   You should see a dialog that says **Hello**. Notice that it does **not** have quotes, even though there were quotes in the code. We'll explain that more in a moment.

   Click **OK** to close the alert.

The Console

The Console is a browser developer tool used for testing code and debugging. Let's output our greeting to the Console using the **log()** method on JavaScript's **console** object.

1. Switch back to your code editor.

2. After the alert, add a console.log command as shown below in bold:

```
<script>
   alert('Hello');
   console.log('Hello from console');
</script>
```

3. Save the file and reload the page in your browser.

   The alert still opens, but how do we see the message we logged to the Console?

4. Click OK to close the alert.

5. **Ctrl–click** (Mac) or **Right–click** (Windows) on the page, choose **Inspect**.

6. At the top of the DevTools window, click on the **Console** tab.

   Here you should see the message **Hello from console**

Variables

A **variable** is a container that stores a value. In JavaScript, variables (vars) need to be declared (created) and assigned a **value**. Variables are declared using the reserved word **let** or **var**, although **let** is the preferred, more modern syntax.

JavaScript is case-sensitive, so pay close attention to typing variable names correctly: `myvar`, `myVar`, `Myvar`, `myVar`, and `MYVAR` are all different variables!

1. Switch back to your code editor.

2. Make the following changes shown in bold. Pay close attention to capitalization, because JavaScript is case-sensitive!

```
<script>
   let myMessage = 'Hello';
   alert(myMessage);
   console.log('Hello from console');
</script>
```

Let's break this code down:

- We made up a variable named **myMessage** into which we store the text **`'Hello'`**
- The **let** before the variable name says we're declaring (creating) a new variable.
- The alert is then told to display the contents of that variable. Notice we no longer have quotes around text inside the alert() method.

3. Save and reload the page in the browser.
   - In the alert, you should see the same **Hello** message, but this time it's getting that from the value of the **myMessage** variable.
   - Click **OK** to close the alert.

Strings

1. Back in your code editor, put **single quotes** around **myMessage** in the alert, as shown below:

```
alert('myMessage');
```

2. Save and reload the page in the browser.

   The alert should literally say **myMessage** because the quotes in our code say to take those characters literally. We got the variable's name, instead of its value.

   Click **OK** to close the alert.

Strings, Quotes, & Semi-Colons

string A variable that stores text is called a string. The text goes in quotes, either single or double (both okay). The alert() and log() methods are both taking strings as their arguments.

quotes tell JavaScript that text should be read literally as a **string** of characters. If there are no quotes, the characters are seen as a **variable**.

single or double quotes: either are fine, although issues can arise when there are quotes inside quotes (nested quotes).

semi-colons ; indicate the end of a statement. Semi-colons are optional when used at the end of a line, but we will be using them.

3. Switch back to your code editor.
4. We don't need this code anymore, because we're moving on to something else. We'd like to keep the code for reference though. Add a double slash **//** to each line to comment out the code.

```
// let myMessage = 'Hello';
// alert('myMessage');
// console.log('Hello from console');
```

TIP: To quickly comment out code, select the lines and hit **Cmd–/** (Mac) or **Ctrl–/** (Windows). You can also click anywhere into a single line and use the same keystroke to comment out that line.

5. Save and reload the page in the browser.

   Nothing will happen because all the code is ignored.

Comments

Comments are notes in the code that explain what the code is doing. Comments are ignored at run time and have no effect on the program. Single line comments are preceded by a double-slash: **//**

Commenting out refers to deactivating code by putting it in a comment. A reason to comment out (rather than delete) code is so you can reactivate or refer to it later.

Strings vs. Numbers & Concatenation

1. Switch back to your code editor.
2. Let's further explore how quotes work. Below the commented lines, add the following bold code:

```
   // console.log('Hello from console');
   alert( 2 + 2 );
   alert( '2' + '2' );
</script>
```

3. Save and reload the page in the browser.
   - The first alert is doing basic math, so you should see **4**. The plus is working as an addition sign. Click **OK** to see a second alert.
   - The second alert is doing something different. Quotes indicate a string (characters). The plus sign is now putting one string of characters after another, a process called **concatenation**.

     Concatenation means to connect or join together. Substrings and variables are concatenated into larger strings using the plus sign (+).
4. Click **OK** to close the alert.
5. Back in your code editor, delete everything inside the **script** tag.
6. As shown below, declare two string variables followed by a concatenation line:

```
<script>
   let firstName = 'Dan';
   let lastName = 'Rodney';
   console.log('firstName' + 'lastName');
</script>
```

7. Save and reload the page in the browser.

   In the Console (which should still be open) it should say **firstNamelastName**

   The quotes in our code treat those words as strings (or literal characters). We've concatenated (combined) the two strings, but that's not what we want.

8. Back in your code editor, remove the quotes in the console.log():

```
<script>
   let firstName = 'Dan';
   let lastName = 'Rodney';
   console.log(firstName + lastName);
</script>
```

9. Save and reload the browser.

10. In the Console you should see the message we wanted in the first place: **DanRodney**

    Without quotes, JavaScript treats **firstName** and **lastName** as the variables they are.

    To make this better, let's concatenate a space between the first and last name.

11. Back in your code editor, add the following **' space ' space +** shown in bold:

```
console.log(firstName + ' ' + lastName);
```

    NOTE: Spaces outside the quotes don't matter, but spaces inside the quotes do!

12. Save, reload the browser, and check the Console.

    You should now see the names separated by a space: **Dan Rodney**

13. Back in your code editor, declare a new number variable, and then concatenate it into the greeting:

```
<script>
  let firstName = 'Dan';
  let lastName = 'Rodney';
  let highScore = 1200;
  console.log(firstName + ' ' + lastName + ' your high score is ' + highScore);
</script>
```

    NOTE: Number values can be either **integers** (whole numbers) or **floats** (decimals).

14. Save, reload the browser, and check the Console.

    You should get the greeting: **Dan Rodney your high score is 1200**

Booleans & Multiple Arguments for console.log()

A boolean variable has a value of either **true** of **false** (which need to be lowercase). Booleans are used a lot in **conditional logic**, which you'll get to later.

1. Back in your code editor, declare a new boolean variable, and add a new console.log:

   ```
     console.log(firstName + ' ' + lastName + ' your high score is ' + highScore);

     let online = true;
     console.log('Is Online:', online);
   </script>
   ```

   NOTE: The console.log method accepts multiple arguments, each separated by a comma. This is very useful for identifying or labeling output. In the code above, we log a string **`'Is Online:'`** followed by the value of our new **online** variable. This will remind us in the Console of which variable the value goes with.

2. Save, reload the browser, and check the Console.

   You should now see two lines, the second (new) line should say **Is Online: true**

Error Messages

You'll be getting a lot of error messages during your coding journey, which is normal! Let's see some errors and how to troubleshoot our code.

not defined is an error you'll see if you spell a variable wrong or otherwise attempt to access a variable that does not exist.

1. Back in your code editor, at the start of the script tag, declare a new variable and then misspell it in the console.log:

   ```
   <script>
      let veg = 'kale';
      console.log(vex);

      let firstName = 'Dan';
   ```

2. Save, reload the browser, and check the Console.

   - You should see this error: **Uncaught ReferenceError: vex is not defined**.
   - Notice that none of the other JavaScript runs! Unlike HTML and CSS which are very forgiving and continue on, when JavaScript encounters an error it stops.
   - To the far right of the error, notice it lists the file name and line number (so you can find the error in your code).

3. Back in your code editor, delete the equal sign from the variable declaration:

   ```
   let veg 'kale';
   console.log(vex);
   ```

4. Save, reload the browser, and check the Console.

   You should see this error: **Uncaught SyntaxError: Unexpected string**

   This is because `'kale'` is a string, but a string was not expected immediately following the variable name. What is expected is the equal sign.

5. Back in your code editor, put the equal sign back and fix the **veg** spelling in the console.log:

   ```
   let veg = 'kale';
   console.log(veg);
   ```

6. Introduce a new error by removing the opening parentheses from **log**:

   ```
   let veg = 'kale';
   console.logveg);
   ```

7. Save, reload the browser, and check the Console.

   You should see this error: **Uncaught SyntaxError: Unexpected token `')'`**

   logveg is not valid syntax, but the error didn't kick in until `)`.

8. Back in your code editor, put the opening parentheses back so the code is correct:

   ```
   let veg = 'kale';
   console.log(veg);
   ```

9. Save, reload the browser, and check the Console.

   The first line in the Console should say **kale**

10. We're done, so you can close the page in your web browser.

    These examples may seem basic, but they are laying a very important foundation for the more complex scripts you'll soon be writing.

    NOTE: For each exercise in this book, we have the completed code in a folder named **Done-Files**. Go to **Desktop > Class Files > JavaScript Class > Done-Files > JavaScript-Fundamentals** if you want to refer to our final code.

Variable Naming Rules

When naming variables keep the following rules in mind:

- Do not use spaces:
  bad: **`grand total`**
  good: **`grandTotal`**
- Do not start with a number:
  bad: **`1day`**
  good: **`day1`**
- Do not use special characters except for **`$`** and **`_`**
  bad: **`big-city`**
  good: **`bigCity`**
- Do not use reserved words:
  bad: **`console`**
  good: **`reportConsole`**

Variable Naming Best Practices

Rules (like those above) must be followed or else you get an error.
Best practices are optional, but recommended. Disregarding a best practice won't affect your code's performance.

If a variable name consists of more than one word, a best practice is to write it in **camelCase** like **`totalCost`** (instead of **`totalcost`** or **`total_cost`**).

More best practices for naming variables:

- The only names that get uppercased are **constants**, which means their value never changes. **`MOON_DIAMETER`** would be an example of a constant, because the moon's diameter never varies.
- Make variable names short, yet meaningful:

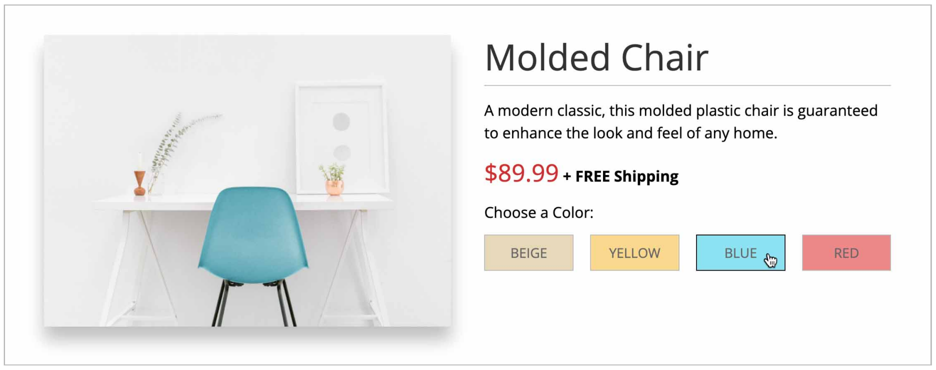
  `let z = 10016;` lacks meaning, because it's too short.

  `let fiveDigitZipCode = 10016;` is meaningful, but needlessly long.

  `let zipcode = 10016;` is meaningful, and short.

The DOM & Getting/Setting Properties

Exercise Preview


Molded Chair
A modern classic, this molded plastic chair is guaranteed to enhance the look and feel of any home.
$89.99 + FREE Shipping
Choose a Color:
BEIGE
YELLOW
BLUE
RED

Exercise Overview

In this exercise you'll learn to work with the DOM (Document Object Model) to target objects in a document (webpage), getting and setting their properties.

When products are available in multiple colors, users need to be able to choose the color. Starting in this exercise (and finishing in later exercises) you'll use JavaScript to make a product color picker that lets a user choose a color and then see a photo of the product in the color they selected.

Getting Started

1. For this exercise we'll be working with the **Product-Chooser-DOM** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
2. In your code editor, open **product.html** from the **Product-Chooser-DOM** folder.
3. Preview **product.html** in Chrome. (We're using Chrome because we'll be using its DevTools.)
4. **Ctrl–click** (Mac) or **Right–click** (Windows) anywhere on the page and choose **Inspect** from the menu.
5. At the top of the DevTools window, click on the **Console** tab.
6. Let's experiment with hiding and showing various objects in this page.

   In the Console, type **document** and hit **Return** (Mac) or **Enter** (Windows).
7. You should see **#document** return in the Console. Hover over the word **#document** and notice that the whole page is highlighted.
8. Click the **arrow** next to **#document** to expand it. This is the whole HTML document, and contains everything on the page.

9. Click the **arrow** next to **`<body>`** to expand it, the expand **`<main>`**
10. Hover over the two div tags inside **`<main>`** and notice that those blocks of content highlight in the browser.

TIP: Adjusting the Console's Text Size

You can adjust the size of the Console text as follows:

- Mac: **Cmd-Plus(+)** enlarges the text. **Cmd-Minus(-)** reduces the text. **Cmd-0** resets to the default size.
- Windows: **Ctrl-Plus(+)** enlarges the text. **Ctrl-Minus(-)** reduces the text. **Ctrl-0** resets to the default size.

Selecting & Working with HTML Elements

We don't typically want to make changes to the whole **document**, but rather specific HTML elements within the document. We can get specific objects using the JavaScript method **document.getElementById()**. This dot syntax will become more familiar as we continue to dig deeper into accessing elements and their properties.

1. At the top left of the DevTools window, click the button.
2. In the page, click on the photo.
3. In the DevTools, notice that the **img** tag with and ID of **product-photo** is highlighted.
4. In the DevTools, click on the **Console** tab.
5. In the Console, type **`document.getElementById('product-photo');`**

About getElementById()

The **document.getElementById('elementID')** method **gets** the element whose **id** is a string that is passed in as the argument (in the parentheses). The element arrives is an object (which is often saved to a variable for future reference). Please note that the capitalization of **ById** in the name of this method must be correct for the code to function.

6. Hit **Return** (Mac) or **Enter** (Windows).

7. You should see **`<img id="product-photo" src="img/chair-beige.jpg">`** in the Console.

   Hover over this and notice that only the image in the page highlights. So far we've selected the whole element whose ID is **product-photo**. We can get more specific and talk to this element's styles by using dot syntax again.

8. To save yourself some typing, press the **Up Arrow** key in the Console. This will show your last command so you don't have to type it again. Very handy!

9. Add the following bold code:

   ```
   document.getElementById('product-photo').style.display = 'none';
   ```

   This says to get the element with the ID **product-photo** and set its **display** property to **none**, which will hide the image. Note that the property value (`'none'`) must be a string.

10. Hit **Return** (Mac) or **Enter** (Windows) to execute the code and notice that the image disappears!

11. Let's get the image to reappear. In the Console, press the **Up Arrow** key again to show your last command.

12. Change the code as shown in bold:

    ```
    document.getElementById('product-photo').style.display = 'block';
    ```

13. Hit **Return** (Mac) or **Enter** (Windows). The image should reappear.

    While changes made in the Console are not permanent, it's a great way to test small pieces of code.

    NOTE: Leave this page open in the browser, so you can reload it to see the changes you make in the code.

Getting & Setting Properties

Notice the product photo is showing the chair in one of its available colors. There are color swatches showing the other colors, but if you click on them they won't do anything yet. When a user clicks one of these color swatches, we want the photo to change to the color they clicked on.

Making something happen when you click requires a function, which you'll learn later. So for now, we'll simply focus on making sure we can change the image. Later we'll make it work when clicking a button (and making all the buttons work).

1. Switch back to **product.html** in your code editor.

2. In your code editor, take note of the following HTML:

```
   <div class="color-swatches">
      <button class="swatch selected" id="beige"></button>
      <button class="swatch" id="yellow"></button>
      <button class="swatch" id="blue"></button>
      <button class="swatch" id="red"></button>
   </div>
</div>
<div class="col photo">
    <img id="product-photo" src="img/chair-beige.jpg">
</div>
```

3. Notice the following:

   - The 4 buttons are "color swatches" a user may click to choose that color product.
   - Each button has an **id** for its color: **beige**, **blue**, **red**, and **yellow**.
   - The **img** tag has an **id** of **product-photo**.
   - The image **src** of **chair-beige.jpg** is what appears when the page first loads.

4. In your code editor's file explorer, open the **img** folder.

   Notice there are 4 product images, one for each color (and the color name exactly matches the **id** we have in HTML):

   - chair-**beige**.jpg
   - chair-**blue**.jpg
   - chair-**red**.jpg
   - chair-**yellow**.jpg

5. In **product.html** add a **script** tag right before the end of the head.

```
   <script></script>
</head>
```

6. Inside the script tag, get the **product-photo** div:

```
<script>
   let productPhoto = document.getElementById('product-photo');
</script>
```

7. Change the image source to the red chair by adding the following bold code:

```
<script>
   let productPhoto = document.getElementById('product-photo');
   productPhoto.src = 'img/chair-red.jpg';
</script>
```

8. Save and reload the page in the browser.

   Hmm, nothing happened. Let's check the Console for errors:

   - **Ctrl–click** (Mac) or **Right–click** (Windows) on the page, choose **Inspect**.
   - At the top of the DevTools window, click on the **Console** tab.
   - You should see an error message. Why?

     Browsers read code from top to bottom. The HTML for the photo is down in the `<body>`. Because the JS that references this object is at the top of the file (before the element has actually been created) our JS code fails! To avoid this problem, it's a best practice to put JS at the bottom of the HTML instead of at the top. This ensures all elements have been created before the JS is executed.

9. Switch back to your code editor.
10. Cut the entire **`<script>`** tag (and its contents) from the head.
11. Paste it just above the closing **`</body>`** tag:

```
   <script>
      let productPhoto = document.getElementById('product-photo');
      productPhoto.src = 'img/chair-red.jpg';
   </script>
</body>
```

12. Save and reload the page in the browser.

    The product photo should now be the **red** chair!

    This hard-coded change does not look at what button the user clicks, which you'll do in a later exercise.

13. Each button has a light gray border that we'll want to change it to another color when the user clicks (so they know which color is currently selected).
14. Back in your code editor, get the red button and change its border color by adding the following bold code:

```
<script>
   let productPhoto = document.getElementById('product-photo');
   let colorButton = document.getElementById('red');

   productPhoto.src = 'img/chair-red.jpg';
   colorButton.style.borderColor = 'lime';
</script>
```

15. Save and reload the page in the browser.

    The **red button** should now have a **lime green** border.

16. **Ctrl–click** (Mac) or **Right–click** (Windows) on the red button and choose **Inspect**.

    In the DevTools Elements tab, notice the button has inline CSS (a **style** attribute) for the lime border.

17. Let's see how to use a class that's already been defined in the CSS. Back in your code editor, change the last line of JavaScript as shown below in bold:

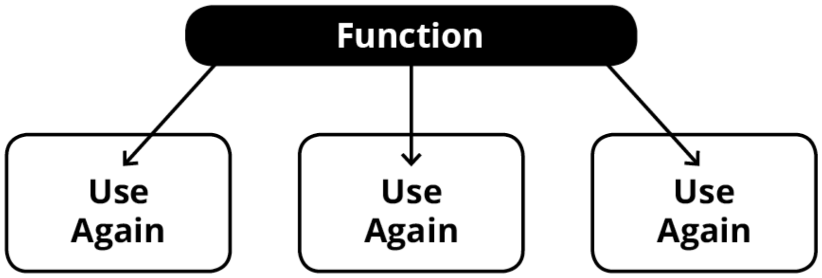
```
<script>
   let productPhoto = document.getElementById('product-photo');
   let colorButton = document.getElementById('red');

   productPhoto.src = 'img/chair-red.jpg';
   colorButton.classList.add('selected');
</script>
```

18. Save and reload the page in the browser.

    - The **red button** should now have a **black** border.
    - Notice in the DevTools Elements tab the button now has a **selected** class and no more **style** attribute. We have a rule for **.selected** in our CSS file.

    NOTE: The advantage of using a class instead of specifying CSS directly in your JS, is it makes the CSS easier to find and edit for other developers (especially those who don't know JS). Also the CSS class can set many CSS properties, keeping your JS clean with just one line of code to add the class.

Exercise Preview



Exercise Overview

In this exercise, you'll learn how to use functions. A **function** is a group of reusable code that performs a specific task (or tasks) at a desired time. The function can be run repetitively, potentially using different information each time it runs. Methods such as **alert()** are similar to functions, but they're predefined. We can do multiple alerts with varying text, but it's essentially doing the same task again. Functions, however, are defined by you, so you decide what they do.

Defining Functions

1. For the first part of this exercise we'll be working with the **Functions** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
2. In your code editor, open **index.html** from the **Functions** folder.
3. Add the following bold code before the closing **`</head>`** tag:

```
   <title>Functions</title>
   <script></script>
</head>
```

4. In the **script** tag, add the following bold code to define a function:

```
<script>
   function displayName() {}
</script>
```

5. In the function's curly braces, add the following bold code which is what the function will do when we call it:

   ```
   function displayName() {
      alert('Dan Rodney');
   }
   ```

   Let's break this down. We just declared a function named **displayName()**. It's good to include a verb in the name, because functions do things.

   The function will run the code in the curly braces **{}**, which is to **alert()** the name **Dan Rodney**. Keep in mind that JavaScript is case-sensitive. We used a capital N for our function's name, **displayName()**. Later we'll need to match that case or else things could break.

6. Save the file.

7. Preview **index.html** (from the **Functions** folder) in a web browser.

8. Notice that nothing happens. Code inside functions is only executed when the function is called. Let's see how to do that.

   NOTE: Leave this page open in the browser, so you can reload it to see the changes you make in the code.

Calling Functions

1. Switch back to your code editor.

2. Ultimately we'll want to run this function when the user clicks a button, but for now let's immediately call the function right after defining it. This means the function will execute immediately on page load.

   ```
   <script>
      function displayName() {
         alert('Dan Rodney');
      }
      displayName();
   </script>
   ```

3. Save and preview the file in a browser.

   Notice the alert is displayed immediately.

4. Click **OK** and return to your code editor.

5. Delete the line of code you just added so you end up with:

```
<script>
   function displayName() {
      alert('Dan Rodney');
   }
</script>
```

6. We want to run this function when the user clicks a button. In the **body** section, type the following text in bold:

```
<body>
   <button>Show a Name</button>
</body>
```

7. Now we need the button to "talk" to JavaScript. HTML elements do not listen for events by default. Instead, we need to tell them to listen for events (such as the click of a button). Here we'll add the **onclick** attribute and set it equal to the function we want to trigger. Add the following bold code:

```
<body>
   <button onclick="displayName();">Show a Name</button>
</body>
```

   NOTE: Adding **onclick** will tell the button to listen for a click event, and only execute the code within the function at that time.

8. Now when the user clicks the button, our **displayName()** function will run. Let's check it out. Save the file and preview in a browser.
   - Click the **Show a Name** button. An alert should appear saying Dan Rodney.
   - Click **OK** to close the alert.
   - Click the button again to see that the alert appears again. We're reusing the code!
   - Click **OK** again.

9. Switch back to your code editor.

Defining Parameters & Passing Arguments

1. An alert that always displays the same thing isn't very flexible. Let's make it work with different names. In order to make a function reusable, we give it **parameters**. Then when we call the function, we can pass information into it. Add these two parameters to your function:

```
function displayName(firstName, lastName) {
   alert('Dan Rodney');
}
```

2. Now edit the alert to use those parameters:

```
function displayName(firstName, lastName) {
   alert(firstName + ' ' + lastName);
}
```

3. While a function's **parameters** ask for information, the pieces of info we pass to it are called **arguments**. Below, change the button to pass the arguments **'Dan'** and **'Rodney'** to the **displayName()** function.

```
<button onclick="displayName('Dan','Rodney');">Show a Name</button>
```

IMPORTANT! Notice that Dan and Rodney are surrounded by single quotes, not double quotes. JavaScript doesn't care which are used, but HTML does. For HTML, the double quotes must wrap the onclick attribute's value. If we used double quotes, HTML would think the onclick attribute would end too early and things would break.

4. Save and preview **index.html** in a browser.
   - Click the **Show a Name** button. Again, it should alert **Dan Rodney**, but this time those names were passed into the function.
   - Click **OK** to close the alert.
5. Switch back to your code editor.
6. Copy and paste the button to make another.

```
   <button onclick="displayName('Dan','Rodney');">Show a Name</button>
   <button onclick="displayName('Dan','Rodney');">Show a Name</button>
</body>
```

7. Alter the second button so it looks as shown below:

```
<button onclick="displayName('Dan','Rodney');">Show a Name</button>
<button onclick="displayName('Trevor','Smith');">Another Name</button>
```

8. Save and preview the file in a browser.
   - Click each of the buttons. They should give you two different alerts. That is one flexible function!

Product Chooser: Functions & Event Handlers

1. In your code editor, close any files you may have open.
2. For the rest of this exercise we'll be working with the **Product-Chooser-Function** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
3. In your code editor, open **product.html** from the **Product-Chooser-Function** folder.

4. Preview **product.html** in a web browser.

   This file is where we left off in a previous exercise for this project. Currently it changes the product photo image and adds a class to one of the buttons (which changes its border color). All this happens on page load, but let's make it so it happens when we click a button.

5. Back in your code editor, wrap a function around the code that changes the image and adds a class to a button:

```
<script>
   let productPhoto = document.getElementById('product-photo');
   let colorButton = document.getElementById('red');

   function changeColor() {
      productPhoto.src = 'img/chair-red.jpg';
      colorButton.classList.add('selected');
   }
</script>
```

6. We want to watch for events involving the **colorButton** (a variable we've already defined). This time we'll do the event a different way, keeping all our code in JavaScript instead of adding it to the HTML. Add the following code shown in bold:

```
      colorButton.classList.add('selected');
   }
   colorButton.addEventListener();
</script>
```

   NOTE: Unobtrusive JavaScript is an approach that separates JS code from HTML. By defining the event handler in JS instead of HTML, we keep all the JS code together, so it's easier to find and edit.

7. Inside addEventListener() add the event we want to look for (in this case a click), and the name of the function we want to execute when that event happens (the **changeColor** function you just defined):

```
colorButton.addEventListener('click', changeColor);
```

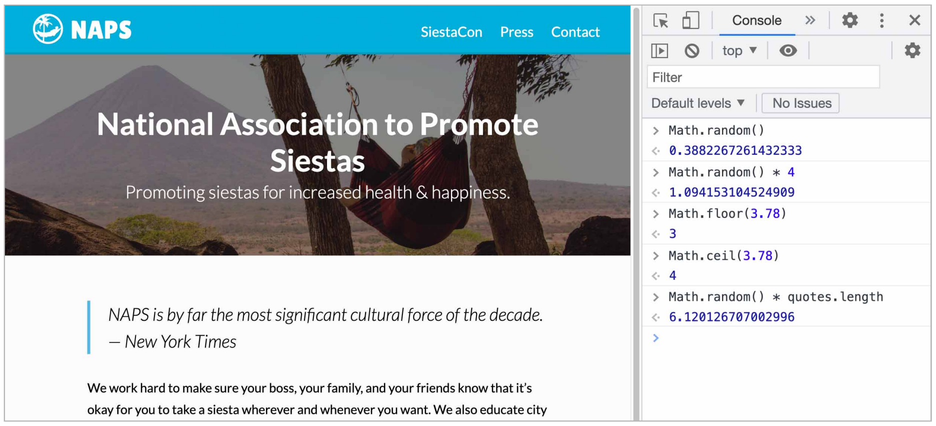
   NOTE: In addEventListener() the function name does NOT have parenthesis after it, because it refers to the function but does not execute it immediately. The event listener will wait until the event is triggered to call the actual function.

8. Save and preview the file in a browser.

9. Click the **red** button to see the image change and the red button get a black border.

   We'll want to reuse this same functionality on all the buttons, not just the red one. To do that we'll have to work with the group of buttons, which is like a list (called an array in JavaScript) and that will also require looping over all the items in that list (array). So before we finish this project, you'll have to learn about those concepts (in the following exercises).

Exercise Preview


NAPS
SiestaCon
Press
Contact
National Association to Promote Siestas
Promoting siestas for increased health & happiness.
Console
top
Filter
Default levels
No Issues

NAPS is by far the most significant cultural force of the decade.
— New York Times

We work hard to make sure your boss, your family, and your friends know that it's okay for you to take a siesta wherever and whenever you want. We also educate city

```
> Math.random()
< 0.3882267261432333
> Math.random() * 4
< 1.094153104524909
> Math.floor(3.78)
< 3
> Math.ceil(3.78)
< 4
> Math.random() * quotes.length
< 6.120126707002996
>
```

Exercise Overview

In this exercise, you'll learn about arrays. You'll also learn about the Math object and how to use it in conjunction with arrays to display the values we want.

Getting Started

1. For this exercise we'll be working with the **Random-Testimonial** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).

2. In your code editor, open **index.html** from the **Random-Testimonial** folder.

3. Preview **index.html** in Chrome. (We'll be using its DevTools.)

4. Notice the testimonial **NAPS is by far the most significant cultural force of the decade. – New York Times**.

   We want to use JavaScript to randomly show a different testimonial each time the page is loaded. In order to accomplish this, we'll be using an array. Think of an **array** as a list. We can store multiple values in a single array. The array stores items in a numbered list, so we can access any specific value by referring to its number (index) within the list. While you typically number lists starting with 1 (then 2, 3, etc.), JavaScript arrays start with 0 (then 1, 2, etc).

Creating an Array

1. Before we start working on this page, let's first learn the fundamentals about arrays by experimenting in the Console. We'll start by create an array. To open Chrome's Console, hit **Cmd–Opt–J** (Mac) or **Ctrl–Shift–J** (Windows).

2. In the Console, type the following but do not hit Return/Enter yet!

   `let myArray = []`

   NOTE: The **`[]`** denotes an array in JavaScript.

3. So far our array is empty. Add some values to the array as shown in bold:

   `let myArray = [`**`'Madrid', 'Paris', 'London'`**`]`

4. Hit **Return** (Mac) or **Enter** (Windows) to apply it.
5. The Console will print **undefined**, but do not be concerned.

   NOTE: Why does it say undefined? Some things in JavaScript return a result when they are executed. Creating an array does not return a value, so the return is undefined. In this case we don't care about a returned value, so you can proceed.

6. Type **`myArray`** and hit **Return** (Mac) or **Enter** (Windows).
7. You should see **`['Madrid', 'Paris', 'London']`** print out in the Console.
8. How do we get a specific value from this array? To get the first value, type:

   `myArray[0]`

   NOTE: Remember that arrays are **zero-indexed**, which means they start numbering with 0.

9. Hit **Return** (Mac) or **Enter** (Windows) to apply it and the string **`'Madrid'`** will print.
10. To get the third value, type:

    `myArray[2]`

    TIP: You can hit your **Up Arrow** key to reload the previous Console command.

11. Hit **Return** (Mac) or **Enter** (Windows) and the string **`'London'`** will print.

Editing an Array

1. To change a value in an array, type:

   `myArray[2] = 'Beijing'`

2. Hit **Return** (Mac) or **Enter** (Windows) and **'Beijing'** will print.
3. Arrays have a variety of methods and we want to test out some of the commonly-used methods. Type the following so we can look at these methods:

   `console.dir(myArray)`

   NOTE: **dir** displays an interactive list of an object's properties. It stands for directory, as in an informational directory.

4. Hit **Return** (Mac) or **Enter** (Windows).
5. Expand the array's list by clicking the **arrow** to the left of **Array(3)**.
   - Notice it says the **length** is **3** (this is how many values are in the array).
   - Click the **arrow** next to **[[Prototype]]: Array(0)**.

     Here you see a list of methods we can use. For example, find **push** and **sort**. Let's try some.
6. What if we want to add a value to the array? To do this, type the following code:

   ```
   myArray.push('New York')
   ```

   NOTE: You can put any value you want to add in the parentheses. Here, we're adding the value `'New York'` to the array.
7. Hit **Return** (Mac) or **Enter** (Windows). It prints **4** to show how many values are in the array.
8. Type `myArray` and hit **Return** (Mac) or **Enter** (Windows).

   It will show the values in the array:

   ```
   ['Madrid', 'Paris', 'Beijing', 'New York']
   ```
9. Checking how many values are in an array is very useful when writing dynamic code. To test it out, type the following:

   ```
   myArray.length
   ```
10. Hit **Return** (Mac) or **Enter** (Windows) and **4** will print.
11. Sometimes it's useful to sort an array. JavaScript arrays have a `sort()` method that will work here. To test it out, type the following:

    ```
    myArray.sort()
    ```
12. Hit **Return** (Mac) or **Enter** (Windows) and the Console should print the cities in alphabetical order:

    ```
    ['Beijing', 'Madrid', 'New York', 'Paris']
    ```
13. Leave **index.html** open in Chrome so we can come back to it later.

Creating an Array of Testimonials

Now that we've seen a bit of what arrays can do, let's get to work on replacing the static testimonial on the page with an array of various press quotes.

1. Switch back to **index.html** in your code editor.

2. Start adding a new array before the closing **</body>** tag:

```
   </footer>
   <script>
      let quotes = [];
   </script>
</body>
```

3. Inside the [] brackets, add some quotes by adding the following bold code:

```
<script>
   let quotes = [
      'first quote',
      'second quote',
      'third quote'
   ];
</script>
```

   NOTE: Our values are strings, so each quote must be wrapped in single quotes. Note that each is separated by a comma, but there's no comma after the last.

4. Save and reload the page in Chrome.
5. Open the Console if it's not already open.
6. Type **quotes** and hit **Return** (Mac) or **Enter** (Windows). You'll see the quotes printed.
7. Now we need to figure out how to switch out the testimonial on the page. Go back to your code editor.
8. Around line 34, notice that we've given the testimonial **p** tag an ID of **press-quote**.
9. Let's see if we can change the testimonial using the Console. Switch back to Chrome.
10. In the Console, let's grab the **press-quote** element:

```
document.getElementById('press-quote')
```

11. Hit **Return** (Mac) or **Enter** (Windows) to apply it.
12. It should print the HTML element (the entire tag and it's contents). We only want the text inside the element though.
13. Hit your **Up Arrow** key to reload the previous command.
14. At the end, add **.innerHTML** so you end up with the following:

```
document.getElementById('press-quote').innerHTML
```

15. Hit **Return** (Mac) or **Enter** (Windows) to apply it.

16. It should print the text string content:

    ```
    "NAPS is by far the most significant cultural force of the decade. — New York
    Times"
    ```

17. Let's try to change that text. Hit your **Up Arrow** key to reload the previous command.

18. At the end, add **= quotes[2]** so you end up with the following:

    ```
    document.getElementById('press-quote').innerHTML = quotes[2]
    ```

    NOTE: In this case, we're changing the text of the testimonial to the third quote in the array. Remember that JavaScript arrays start counting with 0!

19. Hit **Return** (Mac) or **Enter** (Windows) and notice the testimonial on the page now says **third quote**!

    Now that we know how to change the testimonial text, we need to find a way to choose a random testimonial.

The Math Object

The **Math** object contains a lot of very helpful functions for doing various mathematical operations. Let's investigate to see if it can help us.

1. In the Console, type **`Math`** and hit **Return** (Mac) or **Enter** (Windows).

2. Click the **arrow** next to **Math** (the Math object) to expand it. In it, you can see its properties and functions.

   The list starts off with constants (values that are not meant to be changed), such as **`PI`**. Constants are written in UPPERCASE. So for example, if you ever needed to do a mathematical operation that involves `PI`, type **`Math.PI`** and it will give you the value of `PI`.

3. What we'll be focusing on right now is the **random()** method. Type:

   ```
   Math.random()
   ```

4. Hit **Return** (Mac) or **Enter** (Windows). It'll print a random number between 0 and 1.

5. Hit the **Up Arrow** key to reload **Math.random()**.

6. Hit **Return** (Mac) or **Enter** (Windows) to see it generates a different random number.

   How can we apply this to our project? We can have JS choose a random number within a certain range. Then we use that number to choose an item from the quotes array. If we have 5 testimonials, we'd tell it to pick a number between 0 and 4.

7. To get a random number between 0 and 4, type the following code:

```
Math.random() * 4
```

   NOTE: This multiplies the random number by 4 so instead of getting a number between 0 and 1, it outputs one between 0 and 4.

8. Hit **Return** (Mac) or **Enter** (Windows) to see a random number between 0 and 4.

   One problem is that the number we get has many decimal places, but we need an integer (a whole number).

9. The **Math.floor()** method rounds **down** to the closest whole number. Type:

```
Math.floor(3.78)
```

10. Hit **Return** (Mac) or **Enter** (Windows) and it'll print out **3**.
11. The **Math.ceil()** function rounds **up** to the closest whole number. Type:

```
Math.ceil(3.78)
```

12. Hit **Return** (Mac) or **Enter** (Windows) and it'll print out **4**.

Using the Math Object to Pick Random Testimonials

Now that we know how the **Math** object works, let's figure out how we can customize it for our purposes.

1. Switch back to your code editor.
2. First, let's add the actual press quotes to our array. We've already typed out the quotes for you. Go into the **snippets** folder and open **press-quotes.txt**.
3. Select and copy all the text.
4. Close the file. If you aren't already in **index.html**, switch to it.
5. Select the 3 placeholder quotes.
6. Replace them by pasting the new quotes over them.
7. Save and reload the page in Chrome.
8. In the Console, type the following code but do not hit Return/Enter yet!

```
Math.random() * quotes.length
```

   This says to take a random number between 0 and 1 and multiply it by the **length** of the array. We could specify a certain number to multiply by (such as when we used 4 previously) but it's better to use a dynamic number. That way if the number of items in the array changes, you won't have to rewrite the JavaScript.

9. To round it down, wrap the line in a **Math.floor()** as shown below in bold (don't miss the end parenthesis):

```
Math.floor(Math.random() * quotes.length)
```

10. Hit **Return** (Mac) or **Enter** (Windows) to apply the command. You should get a random integer between 0 and 4.

11. Hit the **Up Arrow** key and then hit **Return** (Mac) or **Enter** (Windows) to get another random integer.

12. Perfect! Now we can add this to our page. Copy the line you typed in the Console:

```
Math.floor(Math.random() * quotes.length)
```

13. Back in your code editor, in the **script** tag near the bottom, make a new line below the array and paste the code, as shown in bold:

```
   ];
   Math.floor(Math.random() * quotes.length);
</script>
```

14. To make this number easier to refer to, let's store it in a variable. Add the following code shown in bold:

```
   ];
   let randomNum = Math.floor(Math.random() * quotes.length);
</script>
```

15. Next we need to grab the testimonial that's currently on the page and replace it with a random one. Add the following bold code (as a single line of code):

```
   let randomNum = Math.floor(Math.random() * quotes.length);
   document.getElementById('press-quote').innerHTML = quotes[randomNum];
</script>
```

16. Save and reload the page in Chrome.

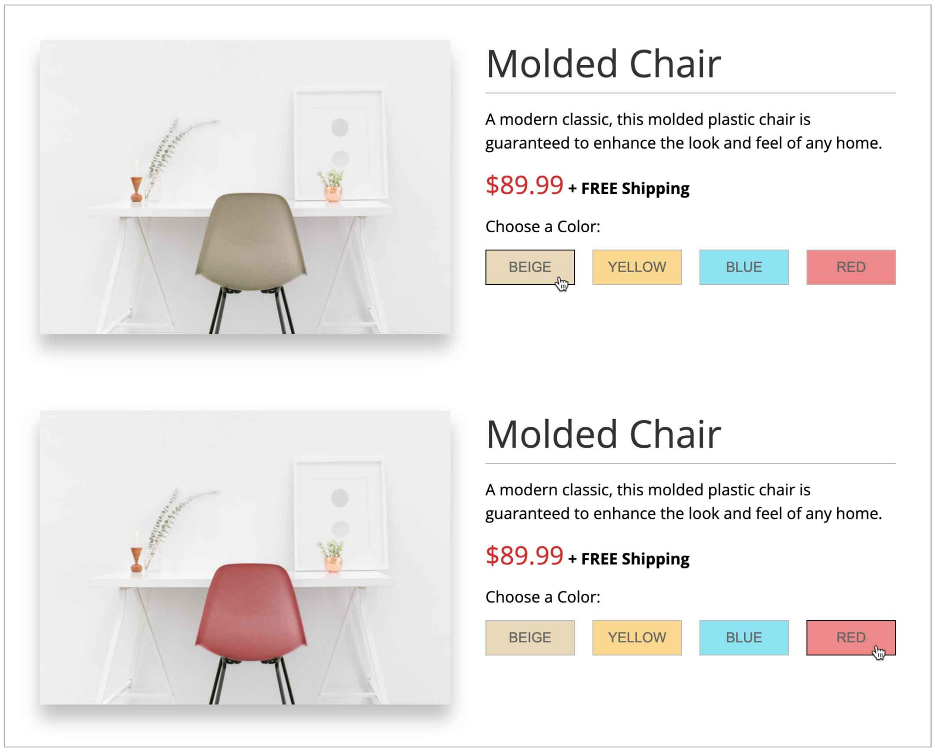
    The page will randomly display one of the testimonials.

17. Reload the page a few more times to see a random testimonial each time. Cool!

    NOTE: We are leaving the static quote in the HTML as a graceful fallback in case a visitor has JavaScript turned off (and for SEO purposes).

For Loops

Exercise Preview


Molded Chair
A modern classic, this molded plastic chair is guaranteed to enhance the look and feel of any home.
$89.99 + FREE Shipping
Choose a Color:
BEIGE
YELLOW
BLUE
RED
Molded Chair
A modern classic, this molded plastic chair is guaranteed to enhance the look and feel of any home.
$89.99 + FREE Shipping
Choose a Color:
BEIGE
YELLOW
BLUE
RED

Exercise Overview

In this exercise, you'll learn how **for loops** are useful for repeating the some code multiple times.

Getting Started

1. For the first part of this exercise we'll be working with the **Loops** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).

2. In your code editor, open **index.html** from the **Loops** folder.

3. Preview **index.html** in Chrome. (We'll be using its DevTools.)

   Notice this is a blank page. We don't actually need any page content because we'll be using the Console.

4. Leave this page open in the browser, so you can reload it later.

Creating a For Loop

1. We've already added a script tag. In it, start writing a **for** loop by adding the following bold code:

   ```
   <script>
      for() {

      }
   </script>
   ```

   - In the `()` we'll add 3 pieces of code controlling the loop.
   - In the `{}` we'll add the code we want to execute every time the loop runs.

2. Let's say we want to count from 0–10. First, we need to add a **counter** to count how many times it's going through the loop. Add the following bold code:

   ```
   for(let i = 0;) {

   }
   ```

   We could name the variable anything but using **i** is a standard convention for incrementing. We also specified that the counter starts at **0**.

3. Every time the loop runs, it will check to see if a certain **condition** is true. If it is true, then we need to tell it what to do… inside the `{}`. If it's false, the loop will stop. Tell the loop to keep going as long as **i is less than 11** by specifying the condition:

   ```
   for(let i = 0; i < 11;) {
   ```

4. Finally, we'll add the **incrementer** itself to specify that every time the loop runs, the value of i will increase by 1. Add:

   ```
   for(let i = 0; i < 11; i++) {
   ```

   NOTE: Alternately, you could write **i = i + 1** or **i += 1** to achieve the same results as **i++** (but with more characters).

5. Now we need to specify what happens each time the loop runs:

   ```
   for(let i = 0; i < 11; i++) {
      console.log('The value of i is: ' + i);
   }
   ```

6. Save the file and reload the page in Chrome.

7. Open the Console by hitting **Cmd–Opt–J** (Mac) or **Ctrl–Shift–J** (Windows).

8. Notice that the loop ran immediately upon page load and printed out the incrementing of i until it got to 10. Perfect!

9. We're done with this page, so close the browser window.

Using a For Loop In the Product Chooser

1. In your code editor, close any files you may have open.
2. For the rest of this exercise we'll be working with the **Product-Chooser-Loops** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
3. In your code editor, open **product.html** from the **Product-Chooser-Loops** folder.
4. Preview **product.html** in Chrome.

   This file is where we left off in a previous exercise for this project. When you click the red button it changes the photo and the button gets an black border, but we want all buttons to have this same functionality.

   All the buttons have a class of **swatch**. Let's get all the buttons (which will be an array), and loop over them (adding an event listener to each one).
5. Back in your code editor, add an **s** to **colorButton** to make it plural:

   ```
   let colorButtons = document.getElementById('red');
   ```
6. Change **getElementById** to **querySelectorAll**:

   ```
   let colorButtons = document.querySelectorAll('red');
   ```

   NOTE: **querySelectorAll()** accepts values like you'd use when targeting elements with CSS selectors, as you'll see next. It finds all of the elements, so you'll get an array instead of a single element as we get with **getElementById()**.
7. Change **red** to the **.swatch** class (which all the buttons have in common):

   ```
   let colorButtons = document.querySelectorAll('.swatch');
   ```
8. Now that we have an array of all the buttons, we need to loop over them, adding an event listener to each button. Wrap a loop around the addEventListener:

   ```
      }
      for(i = 0; i < colorButtons.length; i++) {
         colorButton.addEventListener('click', changeColor);
      }
   </script>
   ```

9. Our addEventListener code is still targeting the single button we had set up. Now that we're using querySelectorAll it will find all the elements with a class of swatch, so we're getting an array.

   As we loop over each button, we'll want to target the specific button using the index of our loop. Add **s[i]** as shown below, being sure not to miss the adding an **s** to **colorButton** to make it plural:

```
for(i = 0; i < colorButtons.length; i++) {
   colorButtons[i].addEventListener('click', changeColor);
}
```

10. Save the file and reload the page in Chrome.

    - Click the **yellow** button to see it changes the image (we know it's to the wrong red image, but we'll fix that next).
    - Reload the page.
    - Click the **blue** button to see it also changes the image (again we know it's the wrong red image, but we'll fix that next).
    - We've successfully made each button change the image (when we started only one button worked). What's left is to make each button set the correct color image, and to add the border around the button being clicked.

Targeting the Current Element

We need to look at the specific individual element being clicked so we can grab it's ID for the color, and use that to change the image to the appropriate image.

1. Back in your code editor, in the **changeColor()** function, add the following:

```
function changeColor() {
   console.log(this);
   productPhoto.src = 'img/chair-red.jpg';
   colorButton.classList.add('selected');
```

The JavaScript Keyword This

In JavaScript, we use the keyword **this** much the way we use the "this" in everyday natural language. In JS (as in natural language) you must be careful to think about how "this" is determined. In the global context (outside of any function), **this** refers to the global object, but an event property is owned by the HTML element it belongs to, therefore in that context **this** refers to the HTML element. You can learn more about the JS keyword **this** at tinyurl.com/javascript-keyword-this

2. Temporarily comment out the colorButton line because that will create an error now that we changed that variable to colorButtons (plural).

   ```
   function changeColor() {
      console.log(this);
      productPhoto.src = 'img/chair-red.jpg';
      // colorButton.classList.add('selected');
   ```

3. Save the file and reload the page in Chrome.

4. Open the Console by hitting **Cmd–Opt–J** (Mac) or **Ctrl–Shift–J** (Windows).

   Click each button and in the Console see it's outputting the button you're clicking!

   Notice the ID of each button is their color. We'll need that to know which image src to use. Let's get that programmatically.

5. Back in your code editor, add **.id** after this:

   ```
   console.log(this.id);
   ```

6. Save the file and reload the page in Chrome.

   Click each of the buttons and in the Console see it's outputting the color of the button you're clicking.

7. Now that we know how to get the current color, we need to use that the grab the correct image. Back in your code editor, replace **red** with the following bold code:

   ```
   function changeColor() {
      console.log(this.id);
      productPhoto.src = 'img/chair-' + this.id + '.jpg';
   ```

8. Save the file and reload the page in Chrome.

   Click each button and notice the image changes to the correct color photo. Nice!

   We'll almost done. We also need change the class of the button that's being clicked, so the user will know which color is currently selected.

9. Back in your code editor, remove the comment from the beginning of the line, and replace **colorButton** with **this**

   ```
   function changeColor() {
      console.log(this.id);
      productPhoto.src = 'img/chair-' + this.id + '.jpg';
      this.classList.add('selected');
   }
   ```

10. Save the file and reload the page in Chrome.

    - Click the **blue** button.

      Notice the image changes to the correct blue photo, and the blue button gets a black border. Perfect.

    - Click the **yellow** button.

      The images changes to the correct photo, and the yellow button gets its border. Those are all great, but the other buttons still have their black border.

      We'll have to remove the borders from all the other buttons before adding the border to the button that is clicked. Because we won't know which other button has the border, we'll loop over them all and remove the class from all of them.

11. Back in your code editor, add the following loop:

```
function changeColor() {
   console.log(this.id);
   productPhoto.src = 'img/chair-' + this.id + '.jpg';

   for(i = 0; i < colorButtons.length; i++) {
      colorButtons[i].classList.remove('selected');
   }
   this.classList.add('selected');
}
```

12. Save the file and reload the page in Chrome.

    Test out all 4 buttons and everything should work perfectly: each button you click should get a black border around it, and change the photo to the appropriate color. Only one button should ever have the black border.

13. Back in your code editor, delete the **console.log(this.id)** line of code.

    You should never leave Console logging in your final code that you'll use.

Optional Bonus: Changing the Click Event to Hover

The product photo changes when the user clicks a button, but what if we wanted it to happen on hover?

1. Back in your code editor, change **click** to **mouseover**:

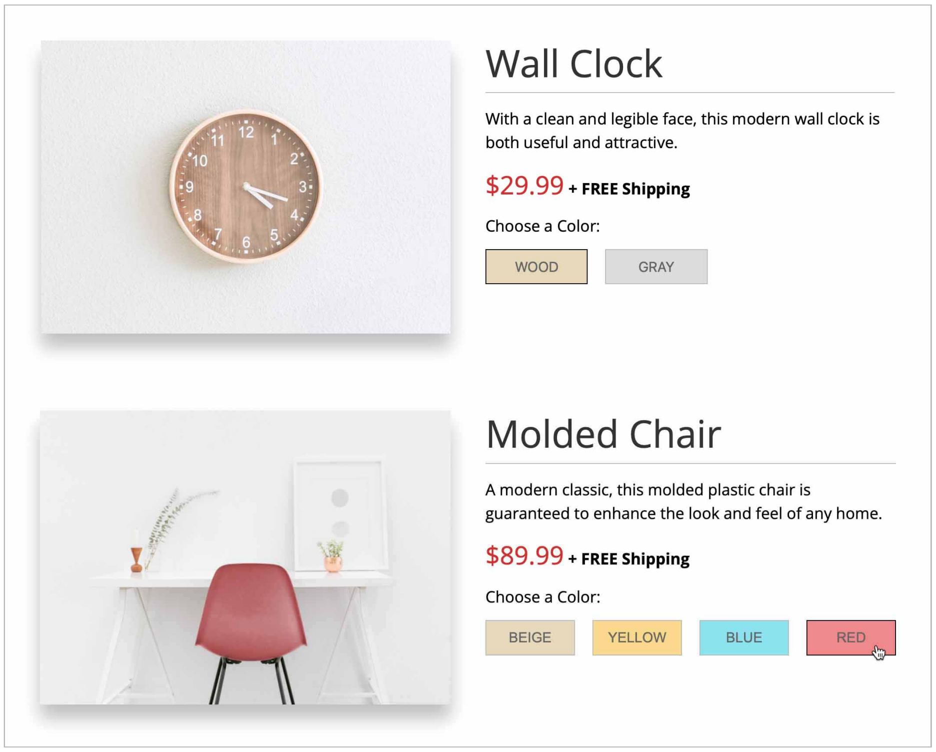
```
colorButtons[i].addEventListener('mouseover', changeColor);
```

2. Save the file and reload the page in Chrome.

   Hover over the color swatches and see you no longer need to click. On mobile devices which don't have the ability to hover, users can tap and it will still work.

External JS Files: Sharing JavaScript Across Pages

Exercise Preview


Wall Clock
With a clean and legible face, this modern wall clock is both useful and attractive.
$29.99 + FREE Shipping
Choose a Color:
WOOD
GRAY
Molded Chair
A modern classic, this molded plastic chair is guaranteed to enhance the look and feel of any home.
$89.99 + FREE Shipping
Choose a Color:
BEIGE
YELLOW
BLUE
RED

Exercise Overview

In this exercise, we'll learn how to externalize JavaScript so it can be shared across multiple webpages.

Getting Started

1. For this exercise we'll be working with the **Sharing-JavaScript** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).

2. In your code editor, open **chair.html** from the **Sharing-JavaScript** folder.

3. Preview **chair.html** in a web browser.

   This is the product color chooser you finished in the previous exercise.

6. Switch back to your code editor.

7. Open **wall-clock.html** from the **Sharing-JavaScript** folder.

8. Preview the page in a browser.

   Notice this page looks the same and also has a color chooser, but it currently does not work because we have not added any JavaScript yet.

   Because we're already written the code to make a color chooser work, let's reuse the code across both of these pages (imagining it could work across any number of product pages).

Externalizing JavaScript

If we copy and paste the JavaScript code into each page, we'll end up with many copies, making updates difficult. Instead we'll move the JavaScript code into a shared .js file, and link multiple HTML pages to it.

1. Switch back to your code editor.
2. Switch back to **chair.html**.
3. Find the **script** tag at the bottom.
4. Select all the code **between** the opening and closing **script** tags. Do not select the **`<script>`** and **`</script>`** tags!

```
<script>
   let productPhoto = document.getElementById('product-photo');
      CODE OMITTED TO SAVE SPACE
   }
</script>
```

5. Cut the code.
6. Create a new file.
7. Paste the code into it.
8. Save the file as **main.js** into the **js** folder in the **Sharing-JavaScript** folder.

Linking to the JavaScript File

We need to link our HTML file to the external JavaScript file we just made.

1. Switch back to **chair.html**.
2. Edit the tags as shown below (make sure to delete any extra lines and spaces between the opening and closing script tags):

```
   <script src="js/main.js"></script>
</body>
```

3. Save and preview **chair.html** in a browser.

   Test out the color buttons to see they should still work.

5. Switch back to your code editor.
6. Switch to **wall-clock.html**.
7. At the bottom of the document, before the closing **`</body>`** tag add the following bold code:

```
   <script src="js/main.js"></script>
</body>
```

8. Save the file.

Making the Code Work for Any Product

Our color chooser JS currently has the the **chair** product name hard-coded into the image path. To make this work with any product, we're going to have to make a minor change.

1. Here in **wall-clock.html**, find the image with the id **product-photo**.

   Notice it has a class of **wall-clock**. (In the other page, it has a class of **chair**.)

   So the class is the product name, which we can grab to use in our JS so we'll know which image to load.

2. Switch to **main.js**.
3. In the **changeColor()** function, replace **chair** as follows:

```
productPhoto.src = 'img/' + productPhoto.className + '-' + this.id + '.jpg';
```

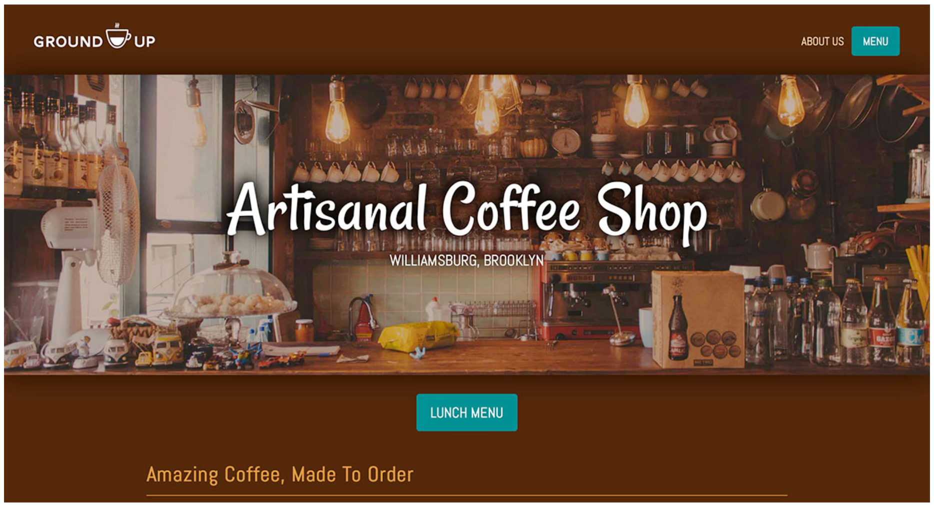
   NOTE: Pay close attention to the single quotes and pluses.

4. Save the file.
5. Preview **wall-clock.html** in a browser.

   Hover the color buttons to see they now work!

6. Preview **chair.html** in a browser and see it still works too. Nice!

Exercise Preview



Exercise Overview

In this exercise you'll learn about if tests, the Date Object, and creating HTML elements via JavaScript.

Conditional Logic

There are times when you only want your JS code to be executed if certain criteria have been met. Programming languages rely on **conditional logic** to make decisions. The usual keywords are **if** and **else**. The logic boils down to:

- **if** a condition is true, do something
- **else** the condition is false, so do something else (or do nothing)

Getting Started

1. For the first part of this exercise we'll be working with the **If-Else-Logic** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
2. In your code editor, open **if-else.html** from the **If-Else-Logic** folder.
3. Preview **if-else.html** in Chrome. (We'll be using its DevTools.)
4. Preview the file in the browser. The page is blank, but we'll be using in the Console.

2D If Else, the Date Object, & Creating Elements

Writing an If Statement

1. Switch back to your code editor.
2. In the script tag, write the if test shown below in bold. The comparison is done with a double equal sign because it's checking something, not setting something.

```
<script>
   let inStock = true;
   if(inStock == true) {
      console.log('Item is in stock');
   }
</script>
```

Single (=) vs. Double (==) Equal Signs

= is an Assignment Operator
A single equal sign **assigns** (sets) a value: **x = 8**
x is assigned a value of 8.

== is a Comparison Operator
A double equal sign **compares** (tests) values without changing them: **x == 8**
x is tested to see if it equals 8. If it does the test is true, otherwise it's false.

3. Save and reload the page in Chrome.
4. Open the Console by hitting **Cmd–Opt–J** (Mac) or **Ctrl–Shift–J** (Windows).

   You should see the message: **Item is in stock**

5. Back in your code editor, change the boolean to **false** as shown below:

```
let inStock = false;
if(inStock == true) {
   console.log('Item is in stock');
}
```

6. Save and reload the page in Chrome.

   There should be no message in the Console this time.

7. Back in your code editor, add an **else** statement with the following bold code:

```
let inStock = false;
if(inStock == true) {
   console.log('Item is in stock');
}
else {
    console.log('Item is out of stock');
}
```

8. Save and reload the page in Chrome.

   In the Console you should see: **Item is out of stock**

More Comparison Operators

You just learned how to check equality. We can also compare numerical values to see if they are less than, equal to, or greater than. Comparisons Operators compare values and the result is either true or false.

Symbol	Comparison Operator	Example & Result
<	less than	3 < 3 is false
>	greater than	3 > 3 is false
==	equal to	3 == 3 is true
<=	less than or equal to	3 <= 3 is true
>=	greater than or equal to	3 >= 3 is true

1. Back in your code editor, below the existing test, add the following if statement using the less-than operator:

```
   }

   let age = 16;
   if (age < 21) {
      console.log('Too young to vote');
   }
</script>
```

2. Save and reload the page in Chrome.

   In the Console you should see: **Too young to vote**

The Date Object

The **Date object** returns an **instance** (a variable object) containing the full date-time from the user's computer.

1. At the bottom of the script tag, instantiate an instance of the Date object as shown below in bold:

   ```
      }

      let dateTime = new Date();
   </script>
   ```

   NOTE: This type of variable declaration, with the **new** keyword, is called the **constructor** method. To use the Date Object, we instantiate it using the constructor method with the resulting **object instance** has access to all the methods and properties of the Date object.

2. Log the **dateTime** object instance to the Console.

   ```
   let dateTime = new Date();
   console.log(dateTime);
   ```

3. Save and reload the page in Chrome.

   In the Console you should get the entire date and time down to the second.

4. Let's extract parts of the date we want (like the hour, month, year, etc.) by invoking methods of the Date object. Methods that return a property value and include **get** in the method name are known collectively as **getters**.

   Get the current hour, month, and year by adding the following bold code:

   ```
   console.log(dateTime);
   let hour = dateTime.getHours();
   let month = dateTime.getMonth();
   let year = dateTime.getFullYear();
   ```

5. Let's output each part of the date onto a different line in the console. To create a new line we use `\n` as shown below:

   ```
   let year = dateTime.getFullYear();
   console.log( hour + '\n' + month + '\n' + year + '\n' );
   ```

6. Save and reload the page in Chrome.

   In the Console you should see 3 lines of numbers (hours, month, and year). The hour is in 24-hour military format, so **4 pm** would appear as **16**.

Using If Statements in a Page

Now that you've learned how if tests and dates work, let's put them together in an actual webpage. We have a coffee shop website which offers different menus for breakfast, lunch, and dinner. To make it easier for visitors to quickly get to the menu for the current time of day, we'll add a link to the appropriate menu. That will require us to get the current hour and then create a link to the appropriate menu.

1. In your code editor, close any files you may have open.
2. For the rest of this exercise we'll be working with the **GroundUp-Timely-Menu** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
3. In your code editor, open **index.html** from the **GroundUp-Timely-Menu** folder.
4. Preview **index.html** in Chrome. (We'll be using its DevTools.)

   We'll be adding the menu link below the main image (above the first heading).
5. Leave the page open in the browser so we can come back and reload it later.
6. Switch back to your code editor.
7. To get the current hour we'll need to use the Date object, so add the following bold code in the script tag at the bottom of the page.

```
<script>
   let dateTime = new Date();
   let hour = dateTime.getHours();
</script>
```

8. We need to write a conditional statement (which is a bit of logic) to figure out which of the 3 times of the day it is: breakfast (before 11am), lunch (11am up to 4pm), and dinner (4pm and later).

   Add the following test. We're let JavaScript do the math of 12 + 4 to give us the hour in the appropriate 24-hour format. You could enter 16 instead, if you prefer.

```
<script>
   let dateTime = new Date();
   let hour = dateTime.getHours();

   if (hour < 11) {

   }
   else if (hour < (12+4)) {  // 4pm

   }
   else {

   };
<script>
```

9. The link will need an href for the appropriate page and some text for inside the link. Let's store these in variables we'll use later when creating the link.

   If we define variables inside an if statement, they can only be used inside that statement. We'll be creating the link outside the if statement, so add the following bold code to define empty variables outside the if statement. Later we'll change the contents of these variables inside the if statement.

```
let dateTime = new Date();
let hour = dateTime.getHours();
let linkText;
let linkHref;

if (hour < 11) {
```

10. Now we can set the appropriate text and href depending on the test. Add the following bold code:

```
if (hour < 11) {
   linkText = 'Breakfast';
   linkHref = 'breakfast.html';
}
else if (hour < (12+4)) {  // 4pm
   linkText = 'Lunch';
   linkHref = 'lunch.html';
}
else {
   linkText = 'Dinner';
   linkHref = 'dinner.html';
};
```

Creating Elements in a Page with JavaScript

Now that we have the appropriate text and href, we can create a link with those attributes and add it to our page.

1. An HTML link is an **<a>** tag. Below the if statement, add the following bold code to create a new one with JavaScript:

```
   };

   let menuLink = document.createElement('a');
</script>
```

2. This new element has been created in the browser's memory, but we have not told JavaScript where in the page to put it. On line 31 notice this empty element:

   ```
   <div id="current-menu"></div>
   ```

   We want to add our link into this element, so take note if it's ID.

3. Add the following bold code to put our link into that div:

   ```
      let menuLink = document.createElement('a');
      document.getElementById('current-menu').appendChild(menuLink);
   </script>
   ```

4. Save and reload the page in Chrome.

   You won't see anything on the page yet, because the link does not have any text. We can verify the tag has been made using the DevTools though.

5. **Ctrl–click** (Mac) or **Right–click** (Windows) on the heading **Amazing Coffee, Made To Order** and choose **Inspect**.

6. In the DevTools, above the currently selected h3, click the triangle next to **`<div id="current-menu">`** to expand it.

   Inside you should an empty link **`<a></a>`**

7. Switch back to your code editor.

8. To put text into the link, add the following bold code (paying very close attention to the space character inside the single quotes!

   ```
   let menuLink = document.createElement('a');
   menuLink.innerHTML = linkText + ' Menu';
   document.getElementById('current-menu').appendChild(menuLink);
   ```

9. The link also needs an href so it knows where to go when people click it, so add the following bold code:

   ```
   let menuLink = document.createElement('a');
   menuLink.innerHTML = linkText + ' Menu';
   menuLink.href = linkHref;
   document.getElementById('current-menu').appendChild(menuLink);
   ```

10. Save and reload the page in Chrome.

    - You should see a link that says **Dinner Menu** (or whatever time of day you're doing this exercise).
    - Click the link and it should take you to that page (the menu pages are placeholder without much content, just so you can see it works).

    Great, our link is working properly!

11. Switch back to your code editor.

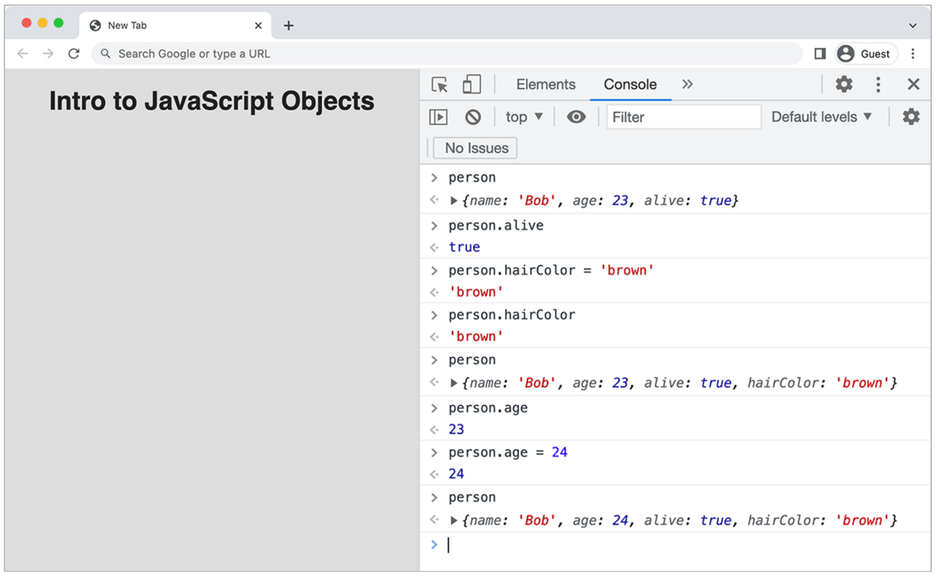
12. One last thing… the link is not styled. We have a **button** class defined in our CSS, so let's assign that to the link by adding the following bold code:

```
menuLink.href = linkHref;
menuLink.className = 'button';
document.getElementById('current-menu').appendChild(menuLink);
```

13. Save and reload the page in Chrome.

    You should now have a nicely styled button that links to the appropriate menu for the current time of day. Nice!

Exercise Preview



Intro to JavaScript Objects

```
> person
< ▸{name: 'Bob', age: 23, alive: true}
> person.alive
< true
> person.hairColor = 'brown'
< 'brown'
> person.hairColor
< 'brown'
> person
< ▸{name: 'Bob', age: 23, alive: true, hairColor: 'brown'}
> person.age
< 23
> person.age = 24
< 24
> person
< ▸{name: 'Bob', age: 24, alive: true, hairColor: 'brown'}
> |
```

Exercise Overview

In this exercise you'll learn the fundamentals of JavaScript objects. You'll create your own object and learn how to access its properties.

JavaScript Objects

JavaScript is an **object-oriented language** because nearly everything is an **object**. An **object** is a collection of properties. Each property is a name (**key**) with a value, which are often called **key-value** pairs. If an object's property value is a function, we'd call it a method.

Objects can store many properties, so they can represent complex, real-world things. For example, a car object might include such properties as doors, color, seats, etc.

Getting Started

1. For this exercise we'll be working with the **Objects** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).

2. In your code editor, open **js-objects.html** from the **Objects** folder.

3. Preview **js-objects.html** in Chrome. (We'll be using its DevTools.)

   This is an almost empty page, but we'll be focusing on the Console.

4. Leave this page open in the browser, so you can reload it later.

Defining an Object

1. Switch back to your code editor.

2. In the script tag, create a variable where we'll save our object:

```
<script>
   let person = {};
</script>
```

   NOTE: **person** is the name of our object. You can name an object anything as long as it doesn't start with a number. The **{}** denotes an object in JavaScript.

3. Inside the **{}** for the object, add the following code shown in bold:

```
let person = {
   name: 'Bob'
};
```

   NOTE: person now contains a single property called **name** that has the string value **Bob**. Properties can be named anything you like (as long as it starts with a lowercase letter). Properties can also be assigned values of any type (string, number, boolean, etc.). Properties and their values are often referred to as **key-value pairs**. In our code, **name** is the **key** and **Bob** is the **value**.

4. Save and reload the page in Chrome.

5. Open the Console by hitting **Cmd–Opt–J** (Mac) or **Ctrl–Shift–J** (Windows).

6. In the Console, type **person** then hit **Return** (Mac) or **Enter** (Windows).

   In the Console you should see **{name: 'Bob'}**

Working with Properties of an Object

1. Let's try using dot syntax to see if we can access the object's **name** property. Type the following:

```
person.name
```

2. Hit **Return** (Mac) or **Enter** (Windows).

   In the Console you should see only the value of name: **'Bob'**

3. Switch back to your code editor.

4. As shown below, add a key with a number value. Multiple key-value pairs in an object are separated by commas, so be sure not to miss the **comma** after **`'Bob'`**:

   ```
   let person = {
      name: 'Bob',
      age: 23
   };
   ```

   NOTE: The additional values don't need to start on a new line but it can help make the code look a little neater and easier to read.

5. Add another key-value pair, this time with a Boolean value:

   ```
   let person = {
      name: 'Bob',
      age: 23,
      alive: true
   };
   ```

   NOTE: A boolean value can only either be **true** or **false**.

6. Save and reload the page in Chrome.
7. The Console should still be open but if you closed it, hit **Cmd–Opt–J** (Mac) or **Ctrl–Shift–J** (Windows) to open it.
8. In the Console, type **`person`** then hit **Return** (Mac) or **Enter** (Windows).

   The Console should show everything in the object (name, age, and alive).

9. Let's check Bob's vitals by looking at just the **alive** value. Type **`person.alive`** then hit **Return** (Mac) or **Enter** (Windows).

   Phew, it should show the boolean value **true**.

10. Let's add another key-value pair dynamically here in the Console. For example, let's say we want to add a hair color value for Bob. Type:

    ```
    person.hairColor = 'brown'
    ```

    NOTE: Remember that any code you write directly to the Console in a browser is only for test purposes and will not be saved in your working document. Make sure not to reload the page, as you will lose your work!

11. Hit **Return** (Mac) or **Enter** (Windows) to apply it.

    The Console should show **`'brown'`**

12. To check if this change has been added to our object, type **`person.hairColor`** then hit **Return** (Mac) or **Enter** (Windows).
13. Type **`person`** then hit **Return** (Mac) or **Enter** (Windows) to see all the object's properties, including the hairColor we just added.

14. Bob just had his birthday so we need to change his age. Type the following then hit **Return** (Mac) or **Enter** (Windows).

    ```
    person.age = 24
    ```

15. Type `person` then hit **Return** (Mac) or **Enter** (Windows) to see that the age property has been updated to **24**. Neat!

JavaScript is Object-Oriented

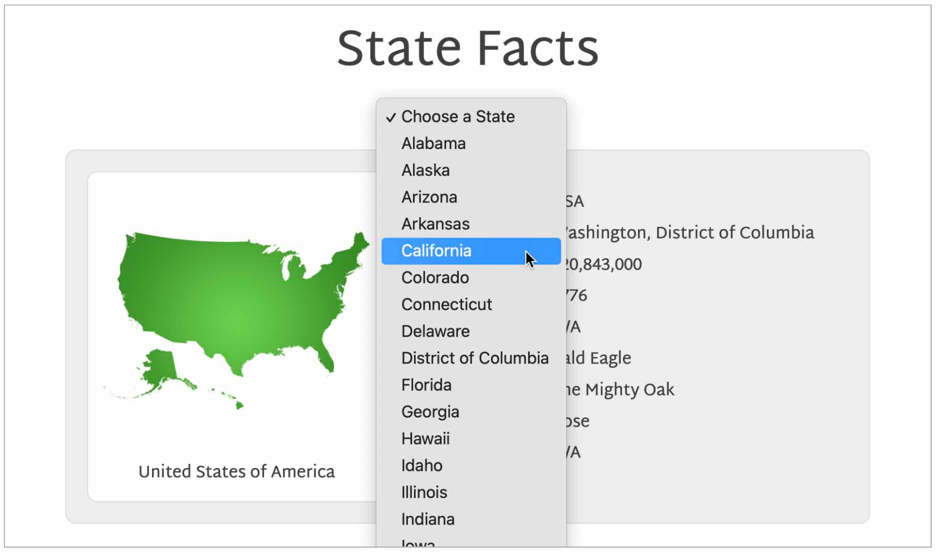
While we can create our own custom objects (as you learned above), keep in mind that just about everything in JavaScript is an object. JavaScript treats HTML elements as objects. Previously, you've seen that you can use **getElementById()** to grab an element. That element is an object in JavaScript, which has properties that you can get and set.

In **Class Files > JavaScript Class > Objects** there are two **dom** PDF files (one is simple and other more detailed).

Those PDF files show a simplified version of the structure of the **window** object (with just a few example objects).

- Everything is contained inside the topmost node, the **window/global object**. Think of the window object as the browser window.
- Within the window object, notice the **document**. This is where all HTML elements live and where you'll spend most of your time when you're working in JavaScript.

Exercise Preview



Exercise Overview

In this exercise, you'll use JavaScript objects to dynamically update content without having to reload the page. In other words, you'll have a single page where the user makes a selection and the info on the page will update without having to reload.

Getting Started

1. For this exercise we'll be working with the **State-Facts** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).

2. In your code editor, open **index.html** from the **State-Facts** folder.

3. Preview **index.html** in Chrome. (We'll be using its DevTools.)

   When a user goes into the **Choose a State** menu at the top and chooses a state, we want them to see the info for it (without loading a new page). That's the functionality we're going to build.

4. Leave the page open in Chrome so we can come back to it later.

Referencing the Menu

1. Return to **index.html** in your code editor.

2. On line 17, find the **select** tag which defines the menu.
   - Notice there is an **option** with a **value** attribute for each state.
   - We will get the value from the option the user has chosen and we'll use that value to access stored data for each state.
3. Before the closing **`</body>`** tag, add the following bold code to save a reference to that select element (the menu):

```
   </div>
   <script>
      let stateList = document.getElementById('state-list');
   </script>
</body>
```

4. Let's output it to the Console so we can look at it more closely. Add the following bold code:

```
let stateList = document.getElementById('state-list');
console.log(stateList);
```

5. Save and reload the page in Chrome.
6. Open the Console by hitting **Cmd–Opt–J** (Mac) or **Ctrl–Shift–J** (Windows).
   - In the Console, the entire select element will appear.
   - Click the **arrow** to its left of the code to expand it and notice all the options.

   Eventually we want to get the value of what the user selects.
7. Back in your code editor, change **log** to **dir**:

```
console.dir(stateList);
```

8. Save and reload the page in Chrome.
   - In the Console, now you'll see **select#state-list** instead of the HTML.
   - Click the **arrow** to its left of the code to expand it. Scroll down below the list of options. Here you see the many properties, etc. you can use in JS.
   - Scroll down to find **value**. The current value is **`"usa"`** but when the user selects a state the **value** will be a 2-letter abbreviation of whichever state they chose, such as **`"ny"`** or **`"nj"`**. This **value** is the info we need to get.

   TIP: To clear the Console, you can hit **Cmd–K** (Mac) or **Ctrl–L** (Windows).
9. Back in your code editor, add **.value** as shown below:

```
console.dir(stateList.value);
```

10. Save and reload the page in Chrome.

    In the Console you should see **usa** (which will be a 2-letter state abbreviation once the user chooses something from the menu).

Listening For When the *Menu* is Changed & Getting the Chosen Value

Previously we've created event listeners for click or mouseover. Let's create one for when the user changes the menu.

1. Event listeners need a function to run when the event is triggered, so wrap the console.dir in a function:

```
let stateList = document.getElementById('state-list');
function showStateInfo() {
   console.dir(stateList.value);
};
```

2. Now add an event listener that will trigger that function:

```
let stateList = document.getElementById('state-list');
function showStateInfo() {
   console.dir(stateList.value);
};
stateList.addEventListener('change', showStateInfo);
```

3. Outside of a function, **this** refers to the global Window object. But, as we've previously seen, inside a function, **this** refers to the object that called the function. Change **stateList** to **this** as shown below:

```
function showStateInfo() {
   console.dir(this.value);
};
```

4. Save and reload the page in Chrome.

   - Choose a state from the menu.
   - In the Console you should see the value of whatever state you chose.
   - Choose another state and it will again show the value for the state you chose.

   Now that we know the event listener is working, we have to work on what happens in that showStateInfo() function.

Loading in the State Data from an External File

We have an object that we're providing to you, which contains info about each state. When our user chooses a state, we want to provide them with that info. Let's take a look at the object you'll pull the data from.

1. Back in your code editor, in the **js** folder open **state-data.js**.

   In this large file, we've declared an object that we can access by referencing the variable **stateData**. As you can see on line 1, the variable's value is wrapped in **`{}`**, which indicates it's an object.

   Its curly braces wrap its many properties (which are all objects). There is a child object for each state (an object within a parent object), which each contain multiple properties (name, abbr, capitol, etc) with appropriate values.

2. In order to use this object, we must load it into our page by linking to this JS file. In your code editor, switch back to **index.html**.

3. Above the current script tag, add in the following link to that js file:

```
</div>
<script src="js/state-data.js"></script>
<script>
```

4. Now let's see how to access the properties of the **stateData** object by adding the following bold code in the script tag:

```
   stateList.addEventListener('change', showStateInfo);
   console.log(stateData.ny.name);
</script>
```

5. Save and reload the page in Chrome.

   In the Console you should see **New York**

6. Switch back to your code editor.

7. Let's see another way to write this same reference. Change **`ny`** to **`['ny']`** as shown below in bold:

```
console.log(stateData['ny'].name);
```

8. Save and reload the page in Chrome.

   In the Console you again should see **New York**.

   This alternate syntax is useful when you need to pass in a variable into the square brackets `[]` which doesn't work in the other dot syntax. You'll see how this will be useful in just a moment.

Dynamically Changing Info on the Page

We're finally ready to start changing the content on the page. Let's start with changing the state name below the image.

1. Back in your code editor, around line 76, find the **info-name** figcaption. We'll set this to the name of whichever state the user chooses.

2. Change the console.dir as follows:

```
function showStateInfo() {
   let selectedState = this.value;
};
```

3. Cut the **console.log(stateData['ny'].name);** and paste it into the function:

```
function showStateInfo() {
   let selectedState = this.value;
   console.log(stateData['ny'].name);
};
```

4. Change the console.log as follows so it puts the name into the **info-name** paragraph on the page:

```
function showStateInfo() {
   let selectedState = this.value;
   document.getElementById('info-name').innerHTML = stateData['ny'].name;
};
```

5. Currently we've hard coded **ny** as the state. Change that to the **selectedState** as shown below in bold:

```
document.getElementById('info-name').innerHTML = stateData[selectedState].name;
```

NOTE: The value of **selectedState** is **this.value** but the dot syntax **stateData.this.value** would not work. Because **this.value** is a dynamic property, it must go in the square brackets.

6. Save and reload the page in Chrome.

   - Choose a state from the menu and notice the state name below the image changes!
   - Next we'll need to change the rest of the info on the page.

Dynamically Changing the Rest of the Page Info

1. Switch back to your code editor.

2. The easiest way for us to set the other values is by copying/pasting the code we already wrote. Copy the following line:

   ```
   document.getElementById('info-name').innerHTML = stateData[selectedState].name;
   ```

3. Paste a copy as a new line, directly below.
4. The next value is the state abbreviation, so edit the code as shown below:

   ```
   document.getElementById('info-name').innerHTML = stateData[selectedState].name;
   document.getElementById('info-abbr').innerHTML = stateData[selectedState].abbr;
   ```

   NOTE: In **state-data.js**, each state in the parent **stateData** object has a key named **abbr** that stores a value for the state abbreviation.

5. Save and reload the page in Chrome.
   - Choose a state from the menu.
   - Now you should see the text for **Abbreviation** change (as well as the state's name below the image).
6. Return to your code editor.
7. Now that you see how this works, doing the rest would be a lot of copy/paste/edit a name. We'll save you time by providing the code with everything already typed out.

   In the **snippets** folder, open **change-function-code.js**.

8. Copy all the code.
9. Switch back to **index.html** and paste the new code below the 2 lines of document.getElementById you already wrote.
10. Save and reload the page in Chrome.

    Choose a state from the menu and you should see all the text info update!

11. All that's left for us to do is change the state image. In your code editor's file explorer sidebar, open the **img** folder to see all the state images.

    Notice that we specifically named them to match the state abbreviations.

12. In **index.html** in your code editor, add the following bold code to get the state image to change:

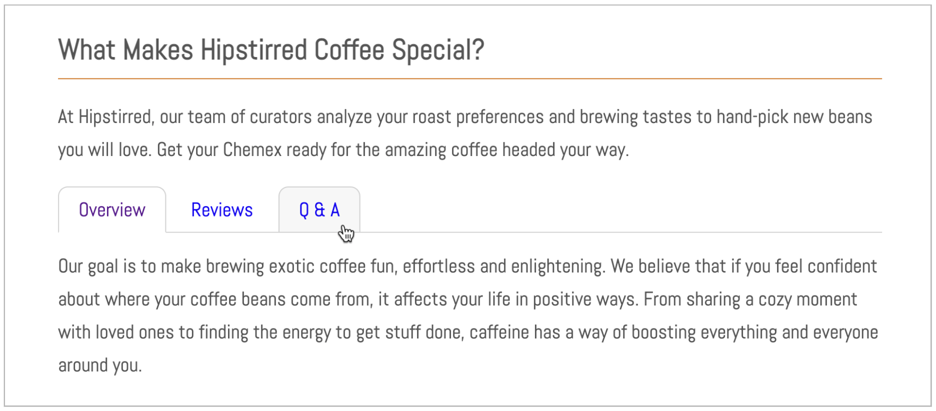
    ```
    let selectedState = this.value;
    document.getElementById('info-pic').src = 'img/' + selectedState + '.jpg';
    document.getElementById('info-name').innerHTML = stateData[selectedState].name;
    ```

13. Save and reload the page in Chrome.

    Choose a state from the menu and you should see the state image change, along with the rest of the info.

Using a JavaScript Library: Tabs

Exercise Preview


What Makes Hipstirred Coffee Special?

At Hipstirred, our team of curators analyze your roast preferences and brewing tastes to hand-pick new beans you will love. Get your Chemex ready for the amazing coffee headed your way.

Our goal is to make brewing exotic coffee fun, effortless and enlightening. We believe that if you feel confident about where your coffee beans come from, it affects your life in positive ways. From sharing a cozy moment with loved ones to finding the energy to get stuff done, caffeine has a way of boosting everything and everyone around you.

Exercise Overview

In this exercise, you'll learn how to use a premade lightweight, accessible vanilla JavaScript called Tabby to create tabbed sections within a webpage.

Premade JavaScript libraries are code some someone else wrote which you can use. It doesn't matter whether or not you know enough JavaScript to create the same functionality, libraries save you time and effort because the hard work is already done. How easy they are to use depends on how well the developer has documented the project so others can understand their work.

Getting Started

1. For this exercise we'll be working with the **Tabs** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
2. In your code editor, open **product.html** from the **Tabs** folder.
3. Preview **product.html** in a browser.

   This page has a lot of content on it, which we want to organize into 3 tabbed sections so users can more easily find what they're looking for.
4. Leave the page open in the browser so we can come back to it later.

Linking to the Script's Files

We've included the **Tabby** files in our class files, but you can also visit github.com/cferdinandi/tabby to download it, read the documentation, and see a demo. Please note that (as with many repositories you'll find) the code you'll use is found in a **dist** folder, which is short for **distribute**.

At a minimum, Tabby requires just one file to run—its JavaScript file. Additionally we can use a CSS file they provide, or use our own CSS instead. We'll use their CSS file, so let's start by linking to it. It's important to link to their CSS **before** our custom CSS. This way our CSS code will override any styling in their CSS should we need to.

1. Back in your code editor, above the link to **main.css**, link to the Tabby CSS file as shown in bold:

```
   <link rel="stylesheet" href="js/tabby/css/tabby-ui.min.css">
   <link rel="stylesheet" href="css/main.css">
</head>
```

   NOTE: Files with **.min** in their name have been minified. That means extra code that users don't need has been removed to make it the smallest possible file size. This can include removing spaces, tabs, line breaks, and comments… all which add file size.

2. Next we'll link to the Tabby JS file. At the bottom, just before the end of the body tag, add the following bold code:

```
   </div>
   <script src="js/tabby/js/tabby.min.js"></script>
</body>
```

3. Save the file.

Adding the HTML

1. From the **snippets** folder, open **tabby-markup.html**.

   This is all the HTML needed to make the tabs (which are the `<ul>`) and the content for each tab (which are the `<div>` tags).

   This code was copied directly from the Tabby GitHub page, which is normally where you'd be copying the code from. The only reason we're providing local files for you to copy/paste from is we never know if someone's website will change.

2. We have already added the content for the tabs to this page, because they contain so much text and we didn't want you to have to do a bunch of mindless copying and pasting.

   So all you need to add is the tabs themselves. Copy the 5 lines of code for the `<ul>` (with the `<li>` tags inside)

3. Switch back to **product.html**.

4. Around line 32 find `<!-- Overview Content -->`.

5. Paste the tab code just **above** that comment.

6. Now we must change the content to make it appropriate for our page. Change the text as shown below:

```
<ul data-tabs>
   <li><a data-tabby-default href="#harry">Overview</a></li>
   <li><a href="#hermione">Reviews</a></li>
   <li><a href="#neville">Q & A</a></li>
</ul>
<!-- Overview Content -->
```

7. Notice the following:

   - The href (links) start with # which means they refer to an ID.
   - Our page already has 3 divs, each with a unique ID (overview, reviews, and questions). You can see them below the tabs you just pasted.

8. We must link each tab to the ID we gave each section of content. Change the links as shown below in bold:

```
<ul data-tabs>
   <li><a data-tabby-default href="#overview">Overview</a></li>
   <li><a href="#reviews">Reviews</a></li>
   <li><a href="#questions">Q & A</a></li>
</ul>
```

9. Save and reload the page in the browser.

   Notice that we have 3 links (Overview, Reviews, and Q & A), but they do not look or work like tabs. That's because we haven't told the Tabby JS to work on them!

Initializing Tabby

We've linked to all the files, so we're ready to initializes Tabby. We'll start with a prepared code snippet we got from the documentation on the Tabby website.

1. Switch back to your code editor.
2. From the **snippets** folder, open **initialize-tabby.js**.
3. Select and copy all the code.

   NOTE: This code creates a new custom event that is defined in the Tabby JS file. It will use event listeners to look for tabs inside a list that will have a **data-tabs** attribute (which you'll see in a moment).

4. Switch back to **product.html**.

5. Paste the code below the link to the Tabby JS file, as shown in bold:

```
   </div>
   <script src="js/tabby/js/tabby.min.js"></script>
   <script>
      var tabs = new Tabby('[data-tabs]');
   </script>
</body>
```

6. Save and reload the page in the browser.

   The tabs now look like tabs! Click the to see they work.

Testing Out the Keyboard Navigation

Tabby adds keystrokes to navigate between tabs, to keep the content accessible to those who can't (or don't want to) use a mouse.

1. Hit your **Tab** key to highlight the first clickable thing in the webpage, which should be the Hipstirred logo.

   NOTE: If Safari you'd have to hit **Option–Tab**.

2. Hit **Tab** several more times until the **Overview** tab is highlighted.
3. While one tab is highlighted, use your **Right Arrow** key to move to the next tab.
4. Then try the **Left Arrow** key to move to the previous tab.

   Cool!

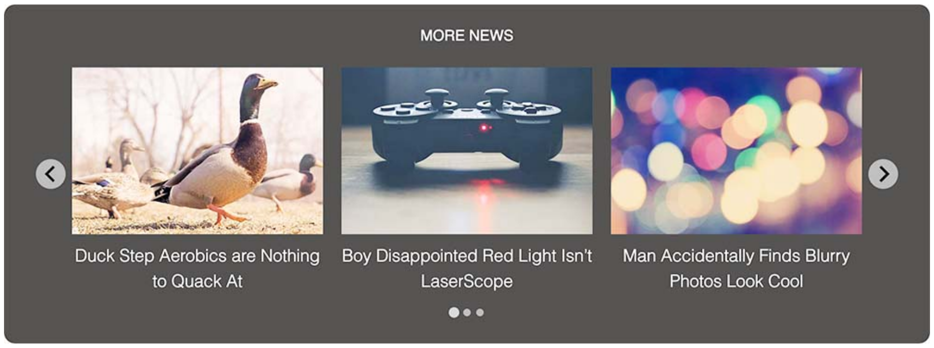
Customizing the Appearance via CSS

When it comes to changing the appearance of premade scripts, as long as they provide a CSS file (instead of baking the CSS into their JS), you can override their CSS by making styles for the elements/classes they define. For this script, you don't even need to link to their CSS file at all. You could start completely from scratch with only your own CSS.

To figure out the elements/classes a script uses, you may be able to refer to the documentation, but typically the best way it to use your browser's DevTools. Inspect an element and start pocking around the HTML, looking at the elements, classes, and styling the script applies. Then write your own CSS targeting those elements/classes.

Using a JavaScript Library: Slideshow/Carousel

Exercise Preview


MORE NEWS
Duck Step Aerobics are Nothing to Quack At
Boy Disappointed Red Light Isn't LaserScope
Man Accidentally Finds Blurry Photos Look Cool

Exercise Overview

In this exercise, you'll learn how to use the premade script called **Splide** to create a carousel or slideshow (also called sliders). These can be a good way to present a series of images and/or text for users to scroll through.

Setting Up the HTML Content

Most JavaScripts assume some specific type of HTML setup (tags with specific class names) which the JS and CSS can target. That markup will be explained in their documentation on their website. The Splide website **splidejs.com** explains all that and more.

On the website they provide some HTML you can copy and paste. Because we never know when a site will change, instead of sending you directly to the site, we pasted their code into a file we provided in the class files.

1. For this exercise we'll be working with the **Carousel** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).

2. From the **snippets** folder open **splide-tags.html**.

   Notice the classes which all refer to splide. Those unique names ensure the Splide provided CSS won't accidentally style objects you may have elsewhere on the page.

   To save you a lot of copying/pasting (or typing of new content), we copied this exact code into our HTML page and added more content in the **splide__slide** li tags.

3. Open **index.html** from the **Carousel** folder.

4. Around line 26 below **`<div id="more-news">`** notice we have the same code. The only things we did was:
   - Change the h2 text (the heading for the slider).
   - For each slide we added a link (to an article) that contains an image and h3. (The links are merely placeholders, so don't bother clicking any of those links in a browser because you'd get a missing page error.)
5. Preview **index.html** in Chrome. (We'll be using its DevTools.)
   - Scroll down to the **More News** section.
   - Currently, all the items are large and you have to scroll down to see them all. This is the unstyled content for our carousel.
6. Leave the page open in the browser so we can come back to it later.

Linking to the Provided Files

1. Switch back to your code editor.
2. Splide provides a CSS to style the slider. After the link to **normalize.css**, add the following bold code:

```
   <link rel="stylesheet" href="css/normalize.css">
   <link rel="stylesheet" href="js/vendor/splide-4.0.0/css/splide.min.css">
   <link rel="stylesheet" href="css/main.css">
</head>
```

   NOTE: We've included the Splide files with the class files, but to download future updates, view examples, and read documentation, you can visit **splidejs.com**

3. Save and reload the page in Chrome.

   In the **More News** section notice that the content is hidden! Don't worry, once we get the carousel working it will come back.

4. We've already linked to a blank **main.js** for you, but you need to link to the Splide script. Near the bottom, add the bold link as show below:

```
<script src="js/vendor/splide-4.0.0/js/splide.min.js"></script>
<script src="js/main.js"></script>
</body>
```

Initializing the Slider/Carousel

1. From the **snippets** folder open **splide-initialize.js**.
2. Copy all the code.

3. From the **js** folder open **main.js**.
4. To initialize the Splide, paste the following bold code:

```
document.addEventListener( 'DOMContentLoaded', function() {
   var splide = new Splide( '.splide' );
   splide.mount();
 } );
```

   NOTE: **var** is another way to declare a variable. In previous exercises we used the more modern **let**, but var has been around longer so you'll it being used sometimes. There's another way using **const**. While there are some differences between **let** and **var**, they both declare variables and you won't see any differences in the way var is being used here.

5. Save and reload the page in Chrome.

   It doesn't look perfect, but we have a working carousel! Try the following:

   - Drag left/right directly on the carousel content (images or text) to move around. Not only does dragging work on mobile devices, it works on desktops too. Cool!
   - Resize the window and notice there's always one item displayed. Soon we'll see how to customize how many items are visible at various screen sizes.

Customizing the Script Using Provided Options

Many premade scripts you'll use will provide a set of options for you to customize the way it works. How many options depends on what the script does and how much work the developer did to allow people to customize it.

These options will be documented on the website for the script. For Splide you can find the complete list of options at splidejs.com/guides/options

1. Switch back to **main.js** in your your code editor.
2. Let's make it display 3 items instead of just one. Add a comma and curly braces (for a JS object) as shown below:

```
var splide = new Splide( '.splide' , {} );
```

3. Inside the object curly braces add hit return and add the following bold setting:

```
var splide = new Splide( '.splide' , {
   perPage: 3
} );
```

4. Save and reload the page in Chrome.

   Notice it displays 3 items, and you can advance to see more sets of 3 items.

5. Let's add some space between the carousel items using a built in setting for that. Switch to **main.js** in your code editor.
6. Add a comma at the end of the first setting, and on a new line add a new setting as shown below in bold:

```
var splide = new Splide( '.splide' , {
   perPage: 3,
   gap: 20
} );
```

7. Save and reload the page in Chrome. Notice the following:
   - There's now some space between each carousel item which looks much better.
   - Click the **next** arrow button on the right until you reach the end and see the carousel stops. We'd like it to rewind back to the beginning.
8. Switch back to **main.js** in your code editor.
9. Add a comma at the end of the last setting, and on a new line add a new setting as shown below in bold:

```
var splide = new Splide( '.splide' , {
   perPage: 3,
   gap: 20,
   rewind: true
} );
```

10. Save and reload the page in Chrome.

    Click the **next** arrow button on the right until you reach the end and see that now it rewinds back to the beginning. Much nicer.

Moving the Prev & Next Arrows Outside the Content

Let's add some padding inside the carousel so the arrow and pager dots at the bottom don't cover over the content. There's no JavaScript option for this, but we can add some padding in our CSS. The section for this carousel has a class of **splide** so we'll target that. You can figure out classes/elements you need to style, either by looking at the HTML you had to add to your page, or using the DevTools inspection features to poke around the code produced by the script you're using.

1. Switch back to your code editor.
2. From the **css** folder open **main.css**.

3. Right above the media query, add the following new rule:

```
.splide {
   padding: 0 55px 35px;
}
@media (min-width: 1100px) {
```

   NOTE: This shorthand sets the padding to 0 on top, 55px on the left/right, and 35px on the bottom.

4. Save and reload the page in Chrome. Notice:

   - It looks much better now that the controls aren't covering the content.
   - Resize the window to see it always displays 3 items, which doesn't look good at smaller screen widths. Let's make it display fewer items on narrower screens.

Controlling How Many Items Are Visible at Various Screen Sizes

We can customize how many items are displayed at specific screens sizes using a **breakpoints** option. We can use this to change the options at any window size.

1. Switch back to **main.js** in your code editor.

2. Add the **breakpoints** option as shown below in bold. Don't miss the comma at the end of the previous line!

```
var splide = new Splide( '.splide' , {
   perPage: 3,
   gap: 20,
   rewind: true,
   breakpoints: {
      850: {
         perPage: 2
      },
      600: {
         perPage: 1
      }
   }
} );
```

3. Save and reload the page in Chrome.

   Resize the browser window to see the carousel changes the number of items displayed from 1 on small screens, to 2, and then 3 as you make the window wider.

   NOTE: When working on responsive sites that change the number of items displayed, be careful of your image sizes. Be sure they are made for the widest possible width, so they will only scale down and not up (which can pixelate them).

Optional Bonus: Lazy Loading Content

Images are typically the largest files in a webpage. They slow down page loading, especially on mobile devices. Splide has an option called lazy loading, which will only load the slider images after the page loads, so the page initially downloads quickly. Let's see how to add lazy loading.

1. Switch to **main.js** in your code editor.
2. Add the following bold code to enable lazy loading:

   ```
   var splide = new Splide( '.splide', {
      lazyLoad: 'nearby',
      perPage: 3,
   ```

3. Save the file.

   There's one more thing we must do. As long as the img tags in our HTML have a **src** attribute, those images will still be downloaded. We need to change that into something HTML won't load, and then Splide will use JavaScript to load the images after the page has downloaded.

4. Switch to **index.html** in your code editor.
5. For every image in the **splide__slide li** tags, you must change the current **src** attribute into **data-splide-lazy**.

   Below is an example of what the final code should look like, with the changes shown in bold. Make this same change to all eight images:

   ```
   <img data-splide-lazy="img/duck.jpg" alt="Ducks. One has a raised leg.">
   ```

   TIP: In many code editors (like Visual Studio Code) you can select the first **img src=** and then hit **Cmd–D** (Mac) or **Ctrl–D** (Windows) to select the following ones (stop after it selects the last one). Then use your Arrow keys to position the text cursor as needed and make the desired change to images all at once!

6. Save and reload the page in Chrome.
7. Hit **Cmd–Opt–J** (Mac) or **Ctrl–Shift–J** (Windows) to bring up the Console.
8. In the DevTools, click on the **Network** tab. If you don't see it, to the right of the **Console** tab click the **>>** and choose **Network** from the menu.
9. Hit **Cmd–R** (Mac) or **Ctrl–R** (Windows) to reload the page.
10. In the list of files, notice the **Initiator** for the **.jpg** images is **setAttribute** (which is JS, so we know those images loaded after the Splide JS did its work).

    Notice the **fronion-logo** Initiator is **index.html** which tells you it loaded normally from HTML.

Exercise Preview

Exercise Overview

In this exercise, you'll learn how to animate something in a webpage using the GreenSock Animation Platform (GSAP), which is the fastest and most robust JavaScript library for animation.

Previewing the Finished Animation

1. For this exercise we'll be working with the **GSAP Intro** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
2. In your code editor, open **done.html** from the **GSAP Intro** folder.
3. Preview **done.html** in a browser.
   - You'll see the Noble Desktop logo appear by scaling in (and moving up slightly). Simple, but cool! This is the finished animation you'll be building.
   - To see the animation again, you can hit **Cmd–R** (Mac) or **Ctrl–R** (Windows) to reload the browser. Feel free to do this until you get a feel for the animation.
4. Close the browser window.
5. Back in your code editor, close **done.html**.
6. Open **intro.html** (from the **GSAP Intro** folder) which is file you'll be coding in.
7. Preview **intro.html** in a browser.

   It's a static logo, which we'll be animating.

8. Back in your code editor, let's get acquainted with the HTML. Look in the **body** tag and you'll see:

```
<img id="logo" src="img/noble-desktop-logo.svg" alt="Noble Desktop">
```

9. Note it has an ID of **logo**. We'll be using that ID to identify this element as a target for our tweens.

Loading the GSAP JavaScripts

GSAP has a single core JavaScript file which handles animating just about any property of any object. It is relatively lightweight, yet full-featured.

GSAP's architecture keeps the overall file size small by letting you load just the features you need. There are extra plugins (some are only available to paid Club GreenSock members), but all we need now is the GSAP core. You can see what's included in the GSAP core file (and learn all about GSAP) at greensock.com/docs

1. Before we start animating, we need to link to the GSAP core JavaScript file. Add the following bold code before the closing **`</body>`** tag:

```
   </main>
   <script src="js/gsap.min.js"></script>
</body>
```

NOTE: While you can can use a CDN link, we'll use a local copy of the script that we've included in the class files. This does not rely on someone else's servers, and allows you to work offline if necessary.

2. Add an empty script tag below that, which is where we'll write our own code:

```
   <script src="js/gsap.min.js"></script>
   <script></script>
</body>
```

Using the gsap.from() Method

1. We'll start by calling the gsap.from() method with the following bold code:

```
<script>
   gsap.from();
</script>
```

2. In the gsap.from() method, add the following bold parameters:

```
gsap.from('#logo', {duration:1, scale:0});
```

The gsap.from() Method

This method requires 2 parameters. The first is the **target** (the object being animated), which in our example is an element with an ID of **logo**. We use the same syntax used to target elements with CSS.

The second parameter is a **vars object { }** defining properties of the animation, such as:

- The **duration** of the animation (**1** second in this example).
- The **start** values for each property that is being tweened. In this example the **scale** property is being tweened **from** a value of **0** to the current value (which is how the element is styled before GSAP animates it).

The **vars object { }** can contain multiple tweenable properties as well as special properties like eases, delays, and more (which we'll cover later). Refer to greensock.com/docs/GSAP/gsap.from() for more details.

gsap.from() specifies **start** values, while **gsap.to()** specifies **end** values.

3. Save the file.
4. Preview **intro.html** in a web browser.
5. Notice the logo takes one second to scale up to the normal size the logo would be if we didn't animate it (the size dictated by HTML/CSS).

   To see the animation again, hit **Cmd–R** (Mac) or **Ctrl–R** (Windows) to reload the web browser.
6. Leave the page open in the browser, so you can reload it later.

Tweening Multiple Properties

You can put multiple properties in the curly braces to animate simultaneously. Let's keep the scale and add a rotation.

1. Return to your code editor and add the following bold code. Be sure to add a comma to separate the properties!

```
gsap.from('#logo', {duration:1, scale:0, rotation:45});
```

NOTE: The order of the properties does not matter.

2. Save and reload the page in the browser.

   Over the course of one second, the logo tweens from a scale of zero as it rotates. It ends at normal rotation, because we're animating **from** these values. Spiffy!

   NOTE: To rotate in the opposite direction, we could use a negative value of –45.

3. Return to your code editor.

4. Delete the **rotation** property so that the gsap.from() method looks like this:

```
gsap.from('#logo', {duration:1, scale:0});
```

5. Let's make the logo move up as it scales. For horizontal movement we animate **x** and for vertical we animate **y**. Type the bold code as shown below:

```
gsap.from('#logo', {duration:1, scale:0, y:100});
```

6. Save and reload the page in the browser.

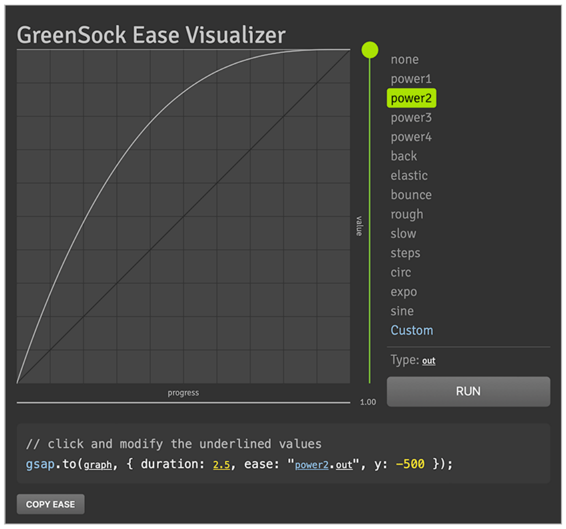
   The logo is moving up and scaling at the same time. Awesome!

Easing

An ease alters the rate of change (speed) during a tween, giving the movement a different feel. There are many different eases we can add to the tween.

1. To see the eases that are available in GSAP, open GreenSock's Ease Visualizer tool at greensock.com/ease-visualizer

2. You will see the following interface. There is a list of eases on the right which allows you to preview its animation.



```
// click and modify the underlined values
gsap.to(graph, { duration: 2.5, ease: "power2.out", y: -500 });
```

NOTE: There's also a video on the Ease Visualizer page with more explanation.

3. To the right of the visualizer:
   - Click **none** and watch the green ball move up at a constant speed (which feels a bit robotic).
   - Click **none** again if you need to rewatch it.
   - Click **power4** and watch how the ball slows down when it approaches the top, as though it's being slowed by gravity. This is more realistic movement.
4. Below the ease graph, click the **Copy Ease** button.
5. Keep this page open so you can come back to it, but return to your code editor.
6. At the end of the vars list, **type a comma** then **ease:** and paste so you end up with:

   ```
   gsap.from('#logo', {duration:1, scale:0, y:100, ease:"power4.out"});
   ```
7. Save and reload the page in the browser.

   The effect is subtle, but the speed of the animation is now faster at the beginning and it slows down as the approaches the final size and position, for a more natural feeling animation.
8. Switch back to the easing visualizer in your browser.
9. To the right of the visualizer, click on **back**.

   Eases can be applied to the beginning (in) or the end (out). Currently the **type** of ease is set to **out**.

   This back ease will go too far at the end, overshooting the target value and will have to come **back** to the final target value… which is why it's called a back ease!
10. Below the ease graph:
    - In the code area we can change the options of the ease. Click the **1.7** and from the menu that appears choose **2**.
    - Notice the graph changes, showing the ease is more exaggerated, going even farther past the final target value.
    - Click the **Copy Ease** button.
11. Return to your code editor.
12. At the end of the vars list replace the existing ease with the new one:

    ```
    gsap.from('#logo', {duration:1, scale:0, y:100, ease:"back.out(2)"});
    ```
13. Save and reload the page in the browser.

    As you can now see, the **Back** ease overshoots the end values before settling back into its final position. This gives a bouncy or bubbly feeling. Cool!

14. Try out 2 more eases:

- Use the easing visualizer to copy the **elastic** ease code (then paste it into your file and preview). This will give you a springy bounce at the end.
- Then try the **bounce** ease. This will feel like the logo is coming forward and hitting an invisible wall, which it bounces off of.

Additional Notes on Easing

Eases are traditionally named after the mathematical equations that are used to generate their smooth curves. Unfortunately names like Quad, Cubic, and Quart do very little to help animators understand which ease is stronger than the next. GSAP's power ease names, on the other hand, make it very easy for animators to experiment with and adjust the strength of the ease they are using. Start with power1 and work your way up to power4.

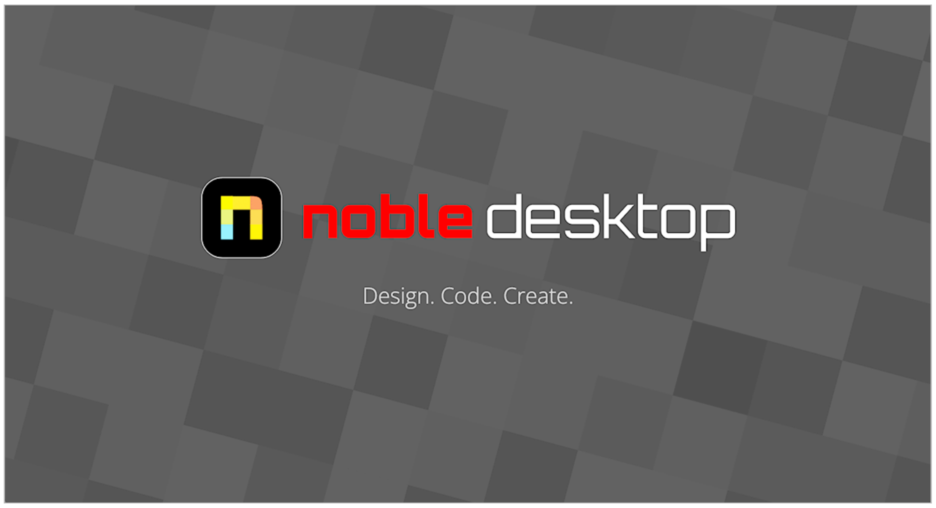
none:	No easing (like power0), also known as linear
power1:	Same as quad
power2:	Same as cubic
power3:	Same as quart
power4:	Same as quint

There are additional eases not included in the gsap core .js file, such as RoughEase, and SlowMo, CustomEase, etc. They can be loaded separately. To see what's included in GSAP's core and the extra eases, visit **greensock.com/docs/installation**

For more info about eases, refer to **greensock.com/docs/v3/Eases**

GSAP Timelines: Animating Multiple Elements in Sequence

Exercise Preview



Exercise Overview

In this exercise, you'll learn how to animate multiple elements in a sequence, using GSAP's timeline feature.

Getting Started

1. For this exercise we'll be working with the **GSAP Timeline** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).

2. In your code editor, open **timeline.html** from the **GSAP Timeline** folder.

3. Preview **timeline.html** in Chrome.

   It's a static logo like we worked with in the previous exercise, but now we've added a tagline below.

4. Back in your code editor, let's get acquainted with the HTML. Look in the **body** tag and you'll see:

```
<img id="logo" src="img/noble-desktop-logo.svg" alt="Noble Desktop">
<p id="tagline">Design. Code. Create.</p>
```

5. Note the IDs on both elements: **logo** and **tagline**. We'll be using those IDs to identify the elements.

Loading the GSAP JavaScripts

1. Before we start animating, we need to link to the GSAP JavaScript file. Add the following bold code before the closing **</body>** tag:

```
   </main>
   <script src="js/gsap.min.js"></script>
</body>
```

2. Add an empty script tag below that, which is where we'll write our own code:

```
   <script src="js/gsap.min.js"></script>
   <script></script>
</body>
```

Animating the First Element

1. As we did in the previous exercise, start by calling the gsap.from() method with the following bold code:

```
<script>
    gsap.from();
</script>
```

2. In the gsap.from() method, add the following bold parameters:

```
gsap.from('#logo', {duration:1, scale:0, y:100});
```

3. Save and reload the page in Chrome.

   The logo should move up slightly as it scales in.

Animating Additional Elements

After the logo is done animating, we want the tagline to appear with the same animation.

1. Back in your code editor, copy and paste the line of code so you have the following:

```
<script>
   gsap.from('#logo', {duration:1, scale:0, y:100});
   gsap.from('#logo', {duration:1, scale:0, y:100});
</script>
```

2. In the second line, change the element we're targeting to the tagline's ID (we gave it that ID in the provided HTML):

```
gsap.from('#logo', {duration:1, scale:0, y:100});
gsap.from('#tagline', {duration:1, scale:0, y:100});
```

3. Save and reload the page in Chrome.

   Both elements animate the same way, at the same time.

   Interesting, but not what we wanted. We want the tagline to **appear** after the logo is done animating. There are a couple ways to accomplish this. First let's use a delay.

4. Switch back to your code editor.

5. Add the delay option shown below in bold (don't miss the comma before it!)

   ```
   gsap.from('#logo', {duration:1, scale:0, y:100});
   gsap.from('#tagline', {duration:1, scale:0, y:100, delay:1});
   ```

6. Save and reload the page in Chrome.

   That's better. The logo takes one second to animation, and the tagline has a delay of one second, so it starts as soon as the logo is done.

7. Switch back to your code editor.

8. Let's tweak this animation slightly. Let's make the tagline move down instead of up. Add a minus sign in front of the y value:

   ```
   gsap.from('#tagline', {duration:1, scale:0, y:-100, delay:1});
   ```

9. Save and reload the page in Chrome.

   That's more interesting.

Creating & Using a Timeline

Sequencing the two animations this way worked, however there are downsides to this approach. Imagine animating 10 different elements. Changing the duration or delay of any tween will mess up the sequence. Another downside is that these types of sequences can't be paused, reversed, or controlled in any manner. All the tweens are running independently of each other. Luckily GSAP has a solution: a timeline.

1. Return to your code editor.

2. Let's get started creating our first timeline. Add the following bold code to instantiate a new timeline:

   ```
   let tl = gsap.timeline();
   gsap.from('#logo', {duration:1, scale:0, y:100});
   gsap.from('#tagline', {duration:1, scale:0, y:-100, delay:1});
   ```

3. Replace **gsap** with **tl** (the name we gave to our timeline):

   ```
   let tl = gsap.timeline();
   tl.from('#logo', {duration:1, scale:0, y:100});
   tl.from('#tagline', {duration:1, scale:0, y:-100, delay:1});
   ```

4. Save and reload the page in Chrome.

   The timing is a bit different now. The logo still takes one second to animate in, but there's now a delay of one second before the tagline appears. That's because each element you add to a timeline happens sequentially, one after another!

   We put the logo into the timeline first, so it happens first. Then we put the tagline in second so it happens after the previous animation is complete. We no longer need a delay, unless we wanted time between the elements.

5. Delete the tagline's delay (and comma before it) so you end up with:

```
let tl = gsap.timeline();
tl.from('#logo', {duration:1, scale:0, y:100});
tl.from('#tagline', {duration:1, scale:0, y:-100});
```

6. Save and reload the page in Chrome.

   Now the tagline will animate as soon as the logo is done!

Cleaning Up the Syntax with Chaining

We can make this code even cleaner by chaining the animations. In short (without including vars in the example) you'll use the syntax **`tl.from().from().from();`**

In the chained animation syntax:

- We only use the name of the timeline once at the beginning.
- We only have one semicolon at the very end (which is optional).
- We can put each from() on a different line to help readability.

2. Back in your code editor, delete the **tl** from the beginning of the from() lines, so you end up with:

```
let tl = gsap.timeline();
.from('#logo', {duration:1, scale:0, y:100});
.from('#tagline', {duration:1, scale:0, y:-100});
```

3. Delete the **semicolons ;** from the end of the from() lines, so you end up with:

```
let tl = gsap.timeline();
.from('#logo', {duration:1, scale:0, y:100})
.from('#tagline', {duration:1, scale:0, y:-100})
```

4. Lastly, put the timeline name back in at the beginning, which we can do on it's own line:

```
let tl = gsap.timeline();
tl
.from('#logo', {duration:1, scale:0, y:100})
.from('#tagline', {duration:1, scale:0, y:-100})
```

5. Save and reload the page in Chrome.

   The animation should still function as it did before. You just have cleaner code.

Adding More Elements to the Timeline

Now that we have a timeline, each animation will display in the order that we list them in the code. So adding a new animation is easier because we don't have to worry about delays.

In the previous exercise, we animated the logo so it scaled and rotated at the same time. What if we want it to scale and then rotate? We can add that as the second animation in our timeline.

1. Back in your code editor, below the logo animation add the following bold line of code for the new animation. Notice this time we're using a **to()** animation. This will start with the current appearance, and animate **to** the desired appearance:

```
.from('#logo', {duration:1, scale:0, y:100})
.to('#logo', {duration:1, rotate:360})
.from('#tagline', {duration:1, scale:0, y:-100})
```

2. Save and reload the page in Chrome.

   Notice the sequence of the animation matches the order of our code:

   - first the logo scales up
   - then the logo rotates
   - then the tagline scales up

Adjusting Timing with the Position Parameter

It's a little monotonous having each tween start the moment the last one ends. What if we want to offset the start times of each tween? For greater flexibility, GSAP's position parameter allows us to adjust the start time of tweens.

How do we access the position parameter? The timeline's **from()** and **to()** methods accepts a position value as the 3rd parameter. The format is:

```
tl.from(target, {vars}, position)
```

To see how this works, let's focus on the tween that rotates the logo. Lets start the rotation 1 second after the logo animation ends.

1. Back in your code editor, add the following bold code to the logo's rotate tween and don't miss the comma!

   ```
   .from('#logo', {duration:1, scale:0, y:100})
   .to('#logo', {duration:1, rotate:360}, '+=1')
   .from('#tagline', {duration:1, scale:0, y:-100})
   ```

   NOTE: This string value is relative to the end of the previous tween, meaning this tween will start 1 second **after** the previous tween is done.

2. Save and reload the page in Chrome.

   Notice the logo scales up, there's a 1 second pause where nothing happens, and then the logo rotates.

3. What if we want the logo to start rotating as it's scaling up? We can use a negative value instead of a positive value. Back in your code editor, make the following change shown in bold:

   ```
   .to('#logo', {duration:1, rotate:360}, '-=.5')
   ```

4. Save and reload the page in Chrome.

   Notice the rotation starts a little after the scaling starts, so for some of the time both the scaling and rotation are happening at the same time.

5. What if we wanted the tagline to start exactly 3 seconds after the start of the timeline, rather than being in the sequence of elements? Instead of using += and -= which are relative to end the previous animation, we can use an absolute number (which is not a string).

   In the tagline tween, add the following bold code and don't miss the comma!

   ```
   .from('#tagline', {duration:1, scale:0, y:-100}, 3)
   ```

6. Save and reload the page in Chrome.

   Notice the tagline starts animating 3 seconds after the animation starts.

7. Let's say we want the tagline to animate at exactly the same time as the previous tween (which is the logo rotating).

   In the tagline tween, change the position parameter as shown below:

   ```
   .from('#tagline', {duration:1, scale:0, y:-100}, '<')
   ```

8. Save and reload the page in Chrome.

   The tagline should start animating at exactly the same time as the logo rotates.

   NOTE: We could also use `'<2'` to start 2 seconds after the previous animation starts. To learn more refer to **greensock.com/position-parameter**

Repeating the Animation

1. One of the benefits of using a timeline, is we can repeat the entire animation. Inside the gsap.timeline() method, add the following bold code to make it repeat twice:

   ```
   let tl = gsap.timeline( {repeat:2} );
   ```

2. Save and reload the page in Chrome.

   Notice it plays once, and then repeats twice. So it plays a total of 3 times.

3. The animation repeats immediately, but it would be nice to see the final appearance for a little while before it repeats. Back in your code editor add the following bold code, and don't miss the comma!

   ```
   let tl = gsap.timeline( {repeat:2, repeatDelay:2} );
   ```

4. Save and reload the page in Chrome.

   Now after the animation plays, it should pause for 2 seconds before repeating.

5. Back in your code editor, let's make it repeat forever by changing the repeat to **negative 1**:

   ```
   let tl = gsap.timeline( {repeat:-1, repeatDelay:2} );
   ```

6. Save and reload the page in Chrome.

   Now it will play, pause for 2 seconds, repeat, and keep doing that forever!

Safari Bug (on macOS and iOS)

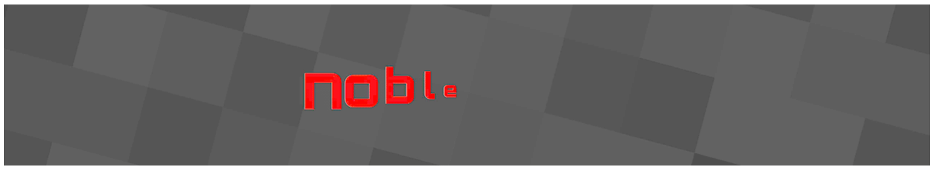
In Safari when the timeline repeats, an imprint of the logo is left behind as it starts animating a new logo, giving us 2 logos. Yuck! This glitch doesn't affect all GSAP animations, and we found a fix with CSS **will-change**.

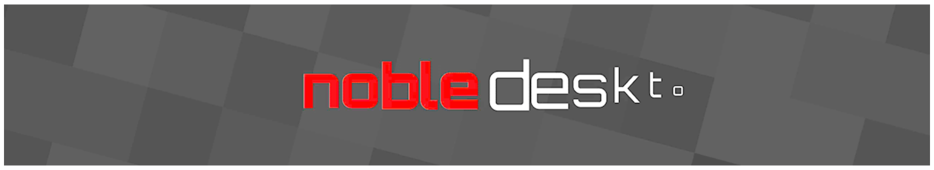
MDN web docs: "The **will-change** CSS property hints to browsers how an element is expected to change. Browsers may set up optimizations before an element is actually changed. Warning: will-change is intended to be used as a last resort, in order to try to deal with existing performance problems."

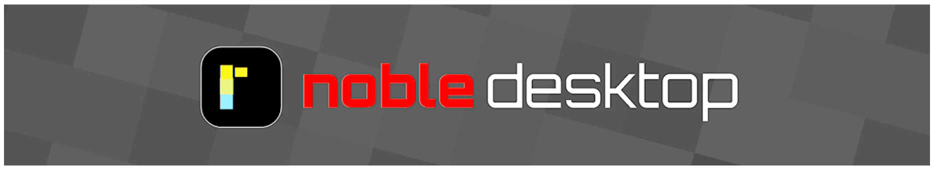
Behind the scenes GSAP animations use a CSS **transform**. We can tell browsers that will change as shown below:

```
#logo {
   will-change: transform;
   CODE OMITTED TO SAVE SPACE
}
```

Exercise Preview







Exercise Overview

In this exercise, you'll learn how to animate SVG (Scalable Vector Graphics) and how to use GSAP's amazing stagger property to create awesome animations with very little code.

Getting Started

1. For this exercise we'll be working with the **GSAP-SVG-Stagger** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).

2. In your code editor, open **svg-stagger.html** from the **GSAP-SVG-Stagger** folder.

3. Preview **svg-stagger.html** in a browser.

   This is the same Noble Desktop logo you've see in the previous exercises, but the HTML is set up differently so we can animate the individual elements within the logo, rather than only the logo as a whole.

4. Back in your code editor, let's get acquainted with the HTML. Notice the following:

   - Instead of an **img** tag like we used in the previous exercises, we copied and pasted all the SVG code from the .svg file into our HTML. This will allow us to refer to specific parts of it!

     FYI, we did not write this SVG code. It was generated by our design app (Adobe Illustrator, Figma, Adobe XD, Sketch, etc.)

   - In the SVG you'll see **<g>** tags, which stands for **group**. They were created and named in our design app which turned them into IDs when exporting the SVG.

   - Notice the various IDs and nested structure of the groups inside the SVG logo:

     — **noble-desktop**
        noble (the letters of the word "noble")
        desktop (the letters of the word "desktop")

     — **n-background** (the rounded black rectangle behind the colored n)
     — **n** (the colored n)

Creating a Timeline & Fixing a Display Issue

1. We've already linked this file to the GSAP .js file, so in the script tag at the bottom of the file, add the following bold code to instantiate a new timeline:

```
<script>
   let tl = gsap.timeline();
</script>
```

2. Before we animate, we must see a problem where the SVG elements will be cropped if they go outside the SVG element. To see this, let's move something with GSAP. To instantly set a property (without animating it), we can use **.set** (instead of **.from** and **.to** that we used before). This does not have a duration as it happens instantly.

   Add the following bold code to move the entire word group a bit to the right:

```
<script>
   let tl = gsap.timeline();
   tl.set('#noble-desktop', {x:50})
</script>
```

3. Save and reload the page in the browser.

   Notice the right side of **desktop** is being cut off because it's protruding beyond the boundaries of the SVG container. Luckily we can fix this easily with CSS.

4. Back in your code editor, from the **css** folder open **main.css**

5. In the **#logo** rule, add the following bold code:

```
#logo {
   overflow: visible;
   width: 100%;
   max-width: 700px;
   height: auto;
}
```

6. Save and reload the page in the browser.

   Now you should be able to see the entire logo, even the part that sticks out of the SVG container.

7. Switch to **svg-stagger.html** in your code editor and replace the **.set** line of code with the tween shown below in bold:

```
<script>
   let tl = gsap.timeline();
   tl
   .from('#noble-desktop', {duration:.4, scale:0, x:-5})
</script>
```

8. Save and reload the page in the browser.

   The words are being scaled up (and moved over slightly from left to right).

   We don't like how the scaling is being done from the top (of the left side). It would look better if they were scaled from the vertical center (of the left side).

Transform Origin

1. In your code editor, add the following bold property and don't miss the comma!

```
.from('#noble-desktop', {duration:.4, scale:0, transformOrigin:'left center',
x:-5})
```

   NOTE: transformOrigin also accepts percent and pixel values, so these both work:
   `transformOrigin:'0% 50%'`
   `transformOrigin:'left center'`

2. Save and reload the page in the browser.

   That looks better with it scaling up from the vertical center (of the left side).

Stagger: Animating Multiple Elements from a Single Tween

The **g** tag in SVG stands for **group**. Inside the **noble-desktop** group we have 2 groups: **noble** and **desktop** (each group contains together the letters for that word). Let's animate those 2 groups instead of the entire Noble Desktop.

We'll use the CSS selector for direct descendant > to target only the groups directly inside noble-desktop (so it doesn't target groups within those groups).

1. Back in your code editor, add **> g** as shown below in bold:

   ```
   .from('#noble-desktop > g', {duration:.4, scale:0, transformOrigin:'left
   center', x:-5})
   ```

2. Save and reload the page in the browser.

   Notice how both of the words animate at the same time. While that's interesting, we want the words to animated one after the other.

   Instead of making two different animations in our timeline as we did in the previous exercise, because these will always be the same animations applied to multiple elements sequentially, we can use a stagger.

3. In your code editor, add the following bold code for the **stagger** property (again don't miss the comma):

   ```
   .from('#noble-desktop > g', {duration:.4, scale:0, stagger:1,
   transformOrigin:'left center', x:-5})
   ```

4. Save and reload the page in the browser.

   - The first word animates, there's a 1 second delay (from the stagger), and then the second work animates.
   - That's cool, but it will be even more impressive when we animation each letter one at a time!

5. In your code editor, reduce the stagger amount to a very short 0.03 seconds:

   ```
   .from('#noble-desktop > g', {duration:.4, scale:0, stagger:0.03,
   transformOrigin:'left center', x:-5})
   ```

6. Change **> g** to ***** so GSAP targets any element inside the **noble-desktop** group (***** is a wildcard that means anything):

   ```
   .from('#noble-desktop *', {duration:.4, scale:0, stagger:0.03,
   transformOrigin:'left center', x:-5})
   ```

7. Save and reload the page in the browser.

   Each letter animates the same way, but with a slight delay between the animations creating a wave of animated letters. Now that's sweeet!

8. To finish this tween, in your code editor add a back ease (don't miss the comma!):

```
.from('#noble-desktop *', {duration:.4, scale:0, stagger:0.03,
transformOrigin:'left center', x:-5, ease:'back.out'})
```

NOTE: The stagger property can be used on any element, not just SVG like we're working with in this example.

9. Save and reload the page in the browser.

Easing is a simple way to change the feeling of an animation. This really adds some bubbliness to it.

Repeating the Animation

Let's make the animation repeat forever, with a slight delay between repeats so you can appreciate the logo for a little while after the animation finishes.

1. Inside the gsap.timeline() method, add the following bold code:

```
let tl = gsap.timeline( {repeat:-1, repeatDelay:1} );
```

2. Save and reload the page in the browser.

The animation should now repeat indefinitely.

Animating the Rest of the Logo

The **n** icon of the logo needs to be animated.

1. Let's start by scaling up the black rectangle background. Below the first tween, add a second shown below in bold:

```
.from('#noble-desktop *', {duration:.4, scale:0, stagger:0.03,
transformOrigin:'left center', x:-5, ease:'back.out'})
.from('#n-background', {duration:.3, scale:0, transformOrigin:'right center',
ease:'back.out'})
```

2. Save and reload the page in the browser.

- Ignore the colored part of the n for now, we'll animate that next.
- Notice the rounded black rectangle (behind the n) now animates after noble desktop.

3. All that's left is the colored parts of the n. We'll use a stagger so they appear one after the other. Below the last tween, add another as shown below in bold:

```
.from('#noble-desktop *', {duration:.4, scale:0, stagger:0.03,
transformOrigin:'left center', x:-5, ease:'back.out'})
.from('#n-background', {duration:.3, scale:0, transformOrigin:'right center',
ease:'back.out'})
.from('#n *', {duration:.2, scale:0, transformOrigin: 'center center',
stagger:0.03, ease:'back.out'})
```

4. Save and reload the page in the browser.

   The logo animation is now done.

   The colored parts of the n should appear sequentially in a quick animation. Isn't it amazing you can do that part with just one line of GSAP code? And the entire logo animation is only a few lines of code. GSAP is indeed very powerful.

Optional Bonus: Finessing Timing with a Position Parameter

Even though the black rectangle background animates immediately after the noble desktop letters are done, it still feels slow. Let's start the black rectangle animation slightly earlier.

1. Add a position parameter that makes the black rectangle (n-background) start a little earlier using a relative value as shown below in bold. Again do not miss that comma!

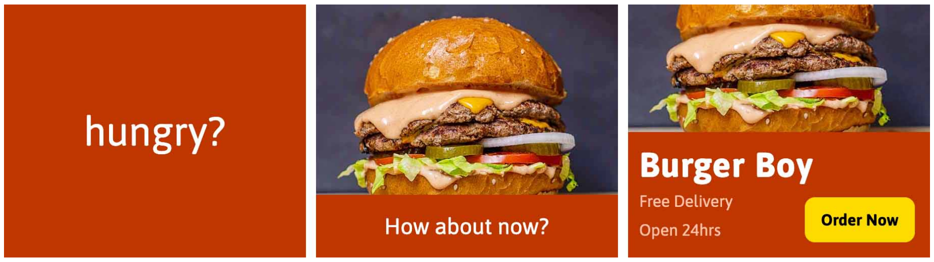
```
.from('#n-background', {duration:.3, scale:0, transformOrigin:'right center',
ease:'back.out'}, '-=0.3')
```

2. Save and reload the page in the browser.

   It's subtle, but nicer with the faster timing. Having the animations slightly overlap can be a nice way to speed up your animations and make them more fluid.

GreenSock: Animating an HTML5 Ad (Google Banner Ads) 5B

Exercise Preview



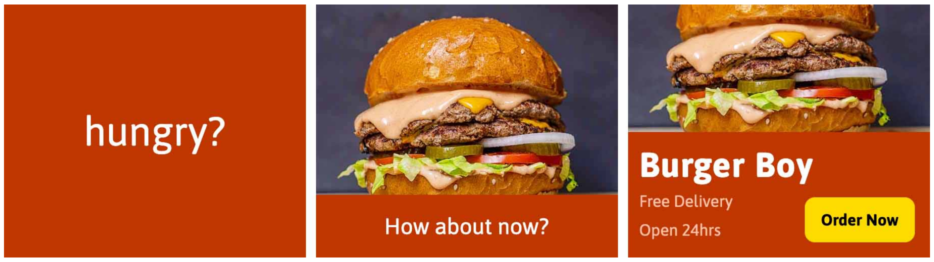
Exercise Overview

In this exercise, you'll create an animated HTML5 banner ad for Google Ads. You'll be doing a fair bit of coding in this exercise, so we'll emphasize best practices and a proper workflow to make the process as easy as possible.

Previewing the Finished Animation

1. On your Desktop (the **Finder** on Mac or **Windows Explorer** on Windows), go into the **Class Files** folder, then **JavaScript Class**.
2. Go into the **Done** folder, then into **GSAP-Ad-Banner**.
3. **Ctrl–click** (Mac) or **Right–click** (Windows) on **index.html**, go to **Open with** and select your favorite browser.

   Watch the animation a few times to get a sense of the pacing and the interaction of the various elements. The animation essentially consists of these 3 main scenes, which we'll call panels:



   Static designers who give you a design to animate will often design these types of static screens (scenes, panels, whatever you want to call them).

Linking to the Google Hosted Version of GreenSock

1. Switch to your code editor.

2. For this exercise we'll be working with the **GSAP-Ad-Banner** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).

3. In your code editor, open **index.html** from the **GSAP-Ad-Banner** folder.

4. Before we look at the page, let's discuss linking to the GSAP library.

   Google Ads limit the total file size of all assets in your ad (150k at the time of this writing). That includes HTML, CSS, images, and custom JS. If you included your own copy of GSAP, it would count again that file size but luckily Google hosts their own copy of GSAP which does not count again the file size!

5. Scroll to the bottom and notice we've already included a script tag linked to the Google hosted version of GSAP.

   We got that link on Google's website here: **tinyurl.com/gads-gs**

Examining the DOM Structure

1. Preview **index.html** in a browser.

   We've laid out all 3 panels in a vertical stack, like a storyboard. When creating a longer, more complex animation, it's a good idea to section out the key moments in the sequence and code the HTML/CSS for these moments first—even before you create a single tween. This let's you work on one panel at a time.

2. Let's look at the HTML. Switch to your code editor and scan over the content, paying attention to the IDs and classes of things (particularly the main sections, which we call a panel):

```
<div id="panel1" class="panel">
   <h1 id="panel1-text">hungry?</h1>
</div>
<div id="panel2" class="panel">
   <h2 id="panel2-text">How about now?</h2>
</div>
<div id="panel3" class="panel">
   <div id="info">
```

   There are 3 **panel** divs, each with a unique ID that indicates their order of appearance in the sequence we'll create. Each panels is an identically-sized (300px by 250px) div that shares the class of **panel**.

3. Now that we know what's going on with the HTML, we should formulate a plan of action. The easiest way to create this animation will be to reposition **#panel2** and **#panel3** so they sit directly atop **#panel1**.

   We'll use a timeline to tell **#panel2** and **#panel3** to appear at the appropriate times. Because each panel is the same size, the animation will look seamless.

Creating the Timeline & Animating the First Panel

1. In the script tag at the bottom of the file, add the following bold code to instantiate a new timeline:

```
<script>
   let tl = gsap.timeline();
</script>
```

2. Now we can add tweens to the timeline. Let's start working on the first panel by adding the following code:

```
<script>
   let tl = gsap.timeline();
   tl
   .from('#panel1-text', {duration:0.5, scale:0.5, opacity:0, ease:'back.out'})
</script>
```

3. Save and reload the page in the browser.

   - The word **hungry?** should scale up.

     If the animation does not work and your code is correct, disable any ad blockers in your browser. They can block the GSAP library from Google!

   - Looks excellent, the first panel is done. Let's move on to the second panel.

Animating the Second Panel

Looking at the second panel down on the page, it has two elements: the burger picture (which is the panel's background image) and the **How about now?** text which has a red background (and an id of **panel2-text**).

1. First, let's position **#panel2** so it sits directly on top of **#panel1** by setting its **top** position to **0** (we've used absolute positioning to position them within the page). Add the following bold code:

```
.from('#panel1-text', {duration:0.5, scale:0.5, opacity:0, ease:'back.out'})
.set('#panel2', {top:0})
```

2. Save and reload the page in the browser.

   As soon as the **hungry?** text fades in, **#panel2** jumps into position to cover it. It's a little abrupt, but we're on the right track. Let's give people a little time to read the first panel.

3. Add the following bold code to the **set()** tween you just added:

   ```
   .set('#panel2', {top:0}, '+=2')
   ```

   By adding this position parameter, we ensure viewers will have 2 seconds to read the **hungry?** text before **#panel2** appears.

4. Save and reload the page in the browser.

   That's much better, but **#panel2** ought to appear with some sort of transition.

5. At the bottom of the timeline, add the following bold code to scale down **#panel2** as it fades in:

   ```
   .set('#panel2', {top:0}, '+=2')
   .from('#panel2', {duration:0.2, opacity:0, scale:1.5})
   ```

6. Save and reload the page in the browser.

   The quick scaling transition is really dynamic—perfect for an eye-catching banner ad.

7. Add the following bold code to make the **How about now?** heading slide in from below the panel after viewers have a moment to savor the burger picture:

   ```
   .from('#panel2', {duration:0.2, opacity:0, scale:1.5})
   .from('#panel2-text', {duration:0.2, yPercent:100}, '+=0.5')
   ```

X & Y versus xPercent & yPercent

Previously we used **x** and **y** for moving a fixed pixel amount. **xPercent** and **yPercent** are GSAP properties for doing percentage-based translations.

We want the #panel2-text to animate up **from** a position where its top edge is aligned with the bottom of the banner. We could try to determine the height of the element and then offset its y position by that amount, but **yPercent** does this for us. It uses a percentage of the element's height to change its position (in this case 100 percent of its height).

Learn more at greensock.com/mistakes#percents

8. Save and reload the page in the browser.

   Looking good. You'll see that **#panel2-text** nicely slides in from the bottom. We want the #panel2-text to disappear again after 2 seconds, so let's add that next.

9. Add the following bold code:

   ```
   .from('#panel2-text', {duration:0.2, yPercent:100}, '+=0.5')
   .to('#panel2-text', {duration:0.2, yPercent:100}, '+=2')
   ```

   This code means that, after **2** seconds, the **#panel2-text** will tween back **to** sitting perfectly below the panel.

10. Save and reload the page in the browser. Cool.

    The second panel is done. (Don't worry if you notice the image scaling in spills outside the panel, we'll hide that later.)

Animating the Third Panel

Looking at the third panel down on the page, it's a red **#info** panel which contains list items (2 lines of text) and an **#order-now** button.

We want the red **#info** panel to slide up, then the list items (2 lines of text) to stagger in. Finally, the **#order-now** button should appear.

1. Move **#panel3** over the other 2 panels by adding the following:

```
.to('#panel2-text', {duration:0.2, yPercent:100}, '+=2')
.set('#panel3', {top:0})
```

2. Save and reload the page in the browser.

   Good, **#panel3** jumps into place at the appropriate time, albeit abruptly.

   By now you're probably sick of seeing the first 2 panels animate. Let's use a label to skip straight to the **set()** for **#panel3** while we are developing and testing the rest of the animation.

3. Add the following bold code to the line you just wrote:

```
.set('#panel3', {top:0}, 'final')
```

4. Create a couple lines of space after that line (so the **seek()** method that jumps to the label doesn't get confused with the timeline's chained tweens) and add:

```
   .set('#panel3', {top:0}, 'final')

   tl.seek('final');
</script>
```

5. Save and reload the page in the browser.

   Good, nothing will animate because we skipped ahead to the last position on the timeline.

6. Add the following code above the **seek()** method to slide up the **#info** panel from the bottom:

```
.set('#panel3', {top:0}, 'final')
.from('#info', {duration:0.5, yPercent:100})

tl.seek('final');
```

7. Save and reload the page in the browser.
   - The panel should slide up from the bottom.
   - The **#info** panel now covers up the meatiest part of the burger, so let's fix that.

Creating Labels

When you provide a label as the position parameter, GSAP will search to see if that label already exists.

If the label DOES NOT exist, GSAP will create the label at the current end of the timeline (that is, the beginning of the tween).

If the label DOES already exist, then the tween will be added at that position.

8. Add the following bold code to move the burger up a bit:

```
.from('#info', {duration:0.5, yPercent:100})
.to('#panel2', {duration:0.5, y:-65})

tl.seek('final');
```

9. Save and reload the page in the browser.

   Better, but it would be nice to have the burger move simultaneously with the #info panel. To do so, let's use the `'<'` position parameter as we learned previously.

10. Edit the **to()** method for **#panel2** by adding the following bold code:

```
.to('#panel2', {duration:0.5, y:-65}, '<')
```

11. Save and reload the page in the browser.

    Excellent!

12. Let's work on staggering the list items. Add the following bold code:

```
.to('#panel2', {duration:0.5, y:-65}, '<')
.from('#info li', {duration:0.3, opacity:0, x:50, stagger:0.1}, '+=0.2')

tl.seek('final');
```

13. Let's review the parameters for this stagger tween.
    - The **from()** tween will take a duration of **0.3** seconds.
    - The **#info li** items will fade in from an opacity of **0** and move in from an initial position of **x:50** (thus appearing to move in from the right).
    - The **#info li** items will stagger in **0.1** second intervals (the second **li** will start animating 0.1 second after the first).
    - All of this will occur **0.2** seconds after the previous tween has ended, as indicated in the position parameter.
14. Save and reload the page in the browser. Looking cool!
15. Add a tween for the **#order-now** button:

```
.from('#info li', {duration:0.3, opacity:0, x:50, stagger:0.1}, '+=0.2')
.from('#order-now', {duration:0.5, scale:0, opacity:0, ease:'back.out'})
```

16. Save and reload the page in the browser. Fantastic!
17. Let's see the animation all the way from the beginning again. **Delete** this line:

```
tl.seek('final');
```

18. Save and reload the page in the browser.

    Watch it from the beginning. The banner animation is complete!

Hiding Overflow

In the second panel, the burger image scales in a larger size, so it extends (or "bleeds") beyond the edges of the panel. Let's clean it up.

1. In your code editor, from the **css** folder open **main.css**.
2. Find the **#banner** rule and change **overflow** to **hidden** as shown below in bold:

```
#banner {
   overflow: hidden;

   CODE OMITTED TO SAVE SPACE

}
```

3. Save and reload the page in the browser.

   Good, now the "bleed" is invisible.

Optional Bonus: Repeating the Animation

Let's make the banner ad repeat, but we want to pause on the last screen for a little while before repeating.

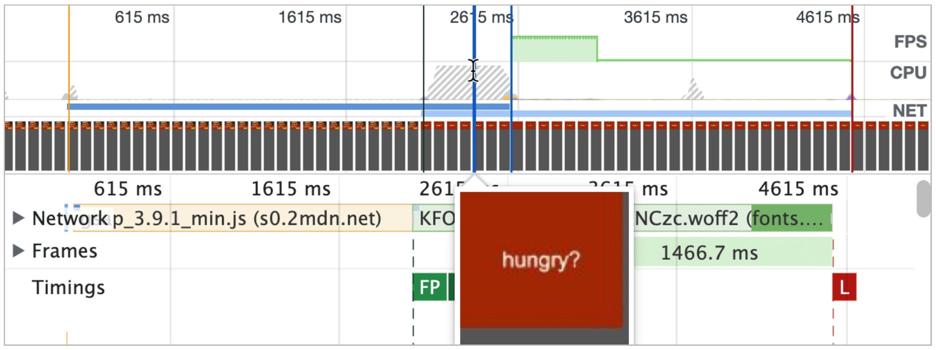
1. In **index.html**, add the following bold code:

```
let tl = gsap.timeline( {repeat:3, repeatDelay:2} );
```

2. Save the file and preview in the browser.

   2 seconds after the animation ends and should repeat automatically. Good stuff.

Exercise Preview



Exercise Overview

In this exercise, you'll prevent a flash of unstyled content which can appear prior to the GSAP .js file being downloaded. You'll also learn more about setting up and validating Google Ads.

Getting Started

1. For this exercise we'll be working with the **GSAP-FOUC** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
2. In your code editor, open **index.html** from the **GSAP-FOUC** folder.
3. Preview **index.html** in Chrome. (We'll be using its DevTools.)

   This is the animated banner ad you created in the previous exercise, but let's look at a couple issues we didn't address yet.

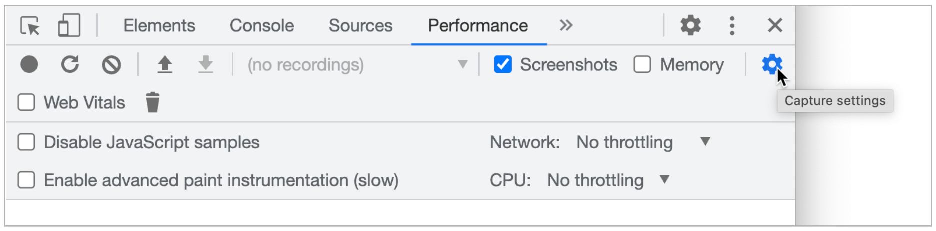
   If the animation does not work, disable any ad blockers in your browser. They can block the GSAP library from Google!

Using Chrome's DevTools to Simulate a Slow Network

How will this load if people are on a slow internet connection? Even if you have fast internet, you can simulate a slow connection using Chrome's DevTools.

1. **Ctrl–click** (Mac) or **Right–click** (Windows) anywhere on the page and choose **Inspect** from the menu.
2. In the DevTools, click on the **Performance** tab. If you don't see it, to the right of the **Elements** tab click the **>>** and choose **Performance** from the menu.

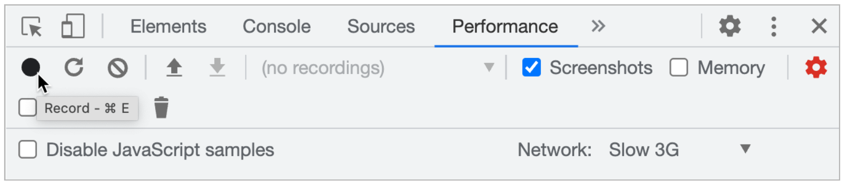
3. Click the gear on the right to show settings (we're interested in **Network** throttling).



Disable JavaScript samples
Enable advanced paint instrumentation (slow)
Network: No throttling
CPU: No throttling

4. For **Network**, set the throttling to **Slow 3G**.

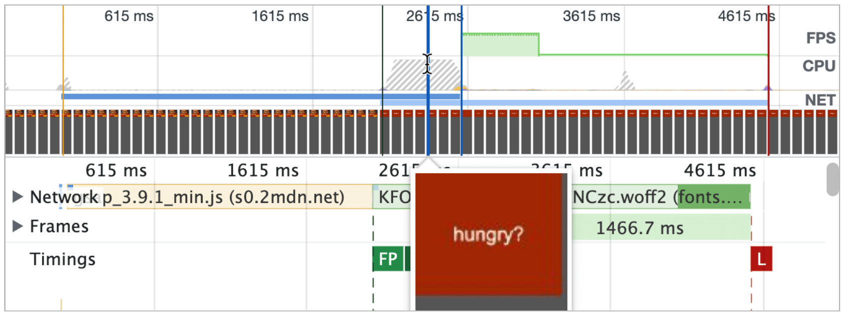
5. At the top left of the DevTools, click the **Record** button.



Disable JavaScript samples
Network: Slow 3G

- Hit **Cmd–R** (Mac) or **Ctrl–R** (Windows) to reload the page.
- As soon the banner starts to animate click **Stop** (be patient, it can take a while on the throttled connection).

6. Hover over the timeline and you'll see a preview (screenshot) of what was recorded.



7. While hovering over the timeline, move right/left to find the point where you reloaded the page and notice the text will be visible for quite some time, and then it will disappear and scale back up.

   So what's happening? Before we animated the page with GSAP, we could see the text. Once GSAP loads, GSAP will notice parse all the animation code and notice we want to scale up the text, so it will shrink it down (or make it transparent, etc., whatever it needs to do to prepare it for animations that are to come) and then GSAP will animate it to the final appearance.

   So if GSAP takes a while to load (like it can on a slower connection), people could see the unprepared assets before they flash away and then animate. Obviously this is not a good situation, but don't worry there's a solution.

Preventing a Flash of Unstyled Content (FOUC) on Page Load

1. In your code editor, from the **css** folder open **main.css**.
2. In the **#banner** rule add **visibility: hidden** as shown below in bold:

```
#banner {
   visibility: hidden;
   overflow: hidden;

   CODE OMITTED TO SAVE SPACE

}
```

3. Save and reload the page in Chrome.

   Everything should be now hidden, so all you'll see is the gray background. Now we need to unhide things once the page (including GSAP) has completely loaded. We'll do this with an event listener.

4. Event listeners will execute a function, so back in **index.html** wrap the entire animation in a function:

```
<script>
function animate() {
   let tl = gsap.timeline( {repeat:3, repeatDelay:2} );

   CODE OMITTED TO SAVE SPACE

   .from('#order-now', {duration:0.5, scale:0, opacity:0, ease:'back.out'})
}
</script>
```

5. Below the function, add an eventlistener that will run the function when the page has finished loading.

```
}
window.addEventListener('load', animate);
</script>
```

6. Lastly we need GSAP to reset the banner's visibility before it animates anything:

```
tl
.set('#banner', {visibility:'visible'})
.from('#panel1-text', {duration:0.5, scale:0.5, opacity:0, ease:'back.out'})
```

7. Save and reload the page in Chrome.
    - Be patient as the throttled connection slowly loads everything. During this time you should see only the gray background.
    - Once it's done loading, the red background should appear and the animation will immediately start.

    Problem solved!
8. In the **Performance** panel, for **Network**, set the throttling back to **No throttling**.
9. Close the DevTools.

For Your Reference: Checking Your Ad Using the Google Ads HTML5 Validator

If this will be submitted to an ad network like Google, you should look over their guidelines to make sure it won't get rejected. For example, Google HTML5 ads:

- Must be 150 KB or less (GSAP & Google Fonts are not included in that amount).
- Custom fonts can only be from Google Fonts.
- The maximum number of files included is 40.
- You must have a meta tag (shown below) to tell Google the ad size (they have a list of specific ad sizes you must use).

  ```
  <meta name="ad.size" content="width=300,height=250">
  ```

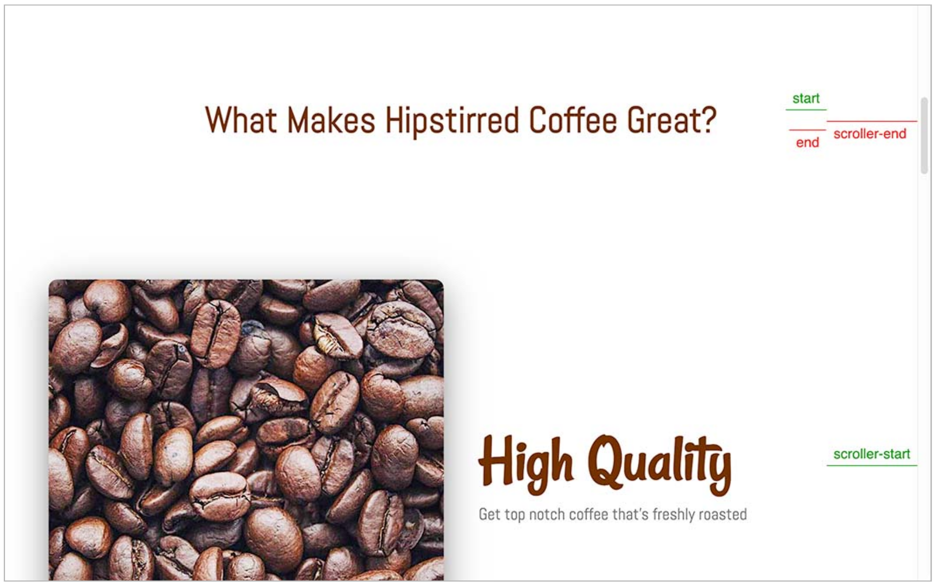
  NOTE: This was added to the file in this project if you want to see an example.
- Do not link to websites. One link will be added through the Google Ad campaign. To use their link, wrap the entire ad in a link (it has a javascript reference to Google's clickTag where it grabs the link). We did that in the file for this project if you want to see an example.
- Learn more about Google's HTML5 guidelines for Ad Manager at **support.google.com/admanager/answer/7046799**

All the files must be in a folder that you zip compress. The .zip file is submitted to Google. Before you submit the zip file to Google, run it through **h5validator.appspot.com/adwords/asset** to check for problems.

For example, we got a validation error and discovered that Google fonts includes the following 2 lines of code, but you must remove them for it to validate properly.

```
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
```

Exercise Preview


What Makes Hipstirred Coffee Great?
start
end
scroller-end
High Quality
Get top notch coffee that's freshly roasted
scroller-start

Exercise Overview

In this exercise you'll learn how to use GSAP's ScrollTrigger plugin to start making animations that are based on scroll position within the page.

Getting Started

1. For this exercise we'll be working with the **GSAP-ScrollTrigger-Intro** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
2. In your code editor, open **index.html** from the **GSAP-ScrollTrigger-Intro** folder.
3. Preview **index.html** in Chrome. (We'll be using its DevTools.)
   - Scroll down and below the photo notice there's a heading **What Makes Hipstirred Coffee Great?**
   - We want to animate the heading when it first scrolls into view.

Linking to the GSAP Scripts

1. First we need to link to the GSAP JavaScript file. Add the following bold code before the closing **</body>** tag:

```
   <script src="js/gsap.min.js"></script>
</body>
```

2. ScrollTrigger is not part of the GSAP core, so we have to load it separately:

```
   <script src="js/gsap.min.js"></script>
   <script src="js/ScrollTrigger.min.js"></script>
</body>
```

3. Add an empty script tag below that for our code:

```
   <script src="js/gsap.min.js"></script>
   <script src="js/ScrollTrigger.min.js"></script>
   <script></script>
</body>
```

4. Add an animation for the heading (which is the only **h2** tag):

```
<script>
   gsap.from('h2', {duration:5, opacity:0, scale:0.5});
</script>
```

5. Save and switch back to Chrome.
   - Scroll up to the top and reload the page.
   - Wait a moment and then scroll down to see the **What Makes Hipstirred Coffee Great?** just below the top photo.
   - Notice the headline is either finishing or has finished its animation. That's not good! Let's make the animation happen when it scrolls into view.

Setting Up a ScrollTrigger

1. Add the **scrollTrigger** property.

```
gsap.from('h2', {scrollTrigger:'h2', duration:5, opacity:0, scale:0.5});
```

2. Save and switch back to your browser.
   - Scroll up to the top and reload the page.
   - Pause a moment and then scroll down.

   When the trigger element (the heading) comes into view (enters the viewport), it triggers the animation to start. That's very different from a timeline where the animations start from when the page loads.

3. Scroll up and down so the heading scrolls scrolls off the top of the screen and scrolls off the bottom.

   Notice the animation only happened once and is not triggered again.

Toggle Actions

1. To set additional scrollTrigger options, we need to change it into an object by changing the **scrollTrigger** from **`'h2'`** to **`{}`** as shown below:

```
gsap.from('h2', {scrollTrigger:{}, duration:5, opacity:0, scale:0.5});
```

2. To make things easier to read, add some lines breaks:

```
gsap.from('h2', {
   scrollTrigger:{}, duration:5, opacity:0, scale:0.5
});
```

3. Now add a line break inside the **scrollTrigger** object (and one before **duration**):

```
gsap.from('h2', {
   scrollTrigger:{

   },
   duration:5, opacity:0, scale:0.5
});
```

4. Put the **`'h2'`** trigger back in, and add a new **toggleAction** property:

```
scrollTrigger:{
   trigger:'h2',
   toggleActions:'restart none none none'
},
```

What are toggleActions?

From the GSAP Docs greensock.com/docs/v3/Plugins/ScrollTrigger

toggleActions: Determines how the linked animation is controlled at the 4 distinct toggle places—**onEnter**, **onLeave**, **onEnterBack**, and **onLeaveBack**, in that order. The default is `'play none none none'`.

So toggleActions: `'play pause resume reset'` will play the animation when entering, pause it when leaving, resume it when entering again backwards, and reset (rewind back to the beginning) when scrolling all the way back past the beginning.

5. Save and reload the page in Chrome.

   toggleActions control what happen when the animation leaves and enters the viewport from the top and bottom of the window. We just set **onEnter** (when the element comes into view from the bottom) to **restart**.

   - Scroll down and when the animation enters the window it should start to animate.
   - Keep scrolling so the heading scrolls off the top of the window and then bring it back into view to see it does not animate again.
   - Scroll up so the heading goes down off the bottom of the window.
   - Scroll back down to see the animation **restarts** (what our code says) every time the headings enters the window from the bottom.

6. Set the second toggleAction to **pause**:

```
scrollTrigger:{
   trigger:'h2',
   toggleActions:'restart pause none none'
}
```

7. Save and reload the page in Chrome.

   We just set **onLeave** (when the element scrolls off the top of the window) to **pause**.

   - Scroll down so the heading starts animating, but keep scrolling so the heading goes off the top of the screen before it's done animating.
   - When the heading leaves the top of the screen it will pause.
   - Bring the heading back down into view and notice the animation is still **paused**.

8. Set the third toggleAction to **resume**:

```
scrollTrigger:{
   trigger:'h2',
   toggleActions:'restart pause resume none'
}
```

9. Save and reload the page in Chrome.

   We just set **onEnterBack** (when the element scrolls into view coming down from the top of the window) to **resume**.

   - Scroll down so the heading starts animating, but keep scrolling so the heading goes off the top of the screen before it's done animating.
   - When the heading leaves the top of the screen it will pause.
   - Bring the heading back down into the view and notice the animation will now **resume** where it left off.

10. **Ctrl–click** (Mac) or **Right–click** (Windows) on the heading we're animating and choose **Inspect**.
11. As you bring the heading into view from the bottom of the window notice in the DevTools some code will have a colored highlight and changes as it animates.

    Once the animate is done, the colored highlight goes away. This is how we know when GSAP is animating something.

    Trigger the animation again but quickly move the heading back down off the bottom of the screen and notice in the DevTools that the animation continues, even though it's offscreen. It's not ideal performance to be animating something that you can't see. Let's fix that next.
12. Now set the last toggleAction to **pause**:

```
scrollTrigger:{
   trigger:'h2',
   toggleActions:'restart pause resume pause'
}
```

13. Save and reload the page in Chrome.

    We just set **onLeaveBack** (when the element scrolls off the bottom of the window) to **pause**.

    - Scroll down so the heading starts animating, but quickly scroll back up so the heading moves down off screen while it's still animating.
    - Notice in the DevTools that the colored highlight stops, indicating GSAP has **paused** the animation. For performance, it's better to stop the animation when it's not onscreen.

Turning on Markers

Let's tell GSAP to show some markers so we can better understand what's going on.

1. Add markers (don't miss the comma on the line above):

```
scrollTrigger:{
   trigger:'h2',
   toggleActions:'restart pause resume pause',
   markers:true
},
```

2. Save and reload the page in Chrome and notice:
    - At the bottom right there's **scroller-start** and at the top right there's **scroller-end**.
    - As you scroll, when **start** meets the **scroller-start** the animation starts.
    - We'll see how **scroller-end** works in a moment.

Scrubbing: Linking an Animation to the Scrollbar

1. Turn on scrubbing (don't miss the comma):

```
scrollTrigger:{
   trigger:'h2',
   toggleActions:'restart pause resume pause',
   markers:true,
   scrub:true
},
```

2. Delete the **toggleActions** and **duration** as they are no longer need:

```
gsap.from('h2', {
   scrollTrigger:{
      trigger:'h2',
      markers:true,
      scrub:true
   },
   opacity:0, scale:0.5
});
```

3. Save and reload the page in Chrome.

   - Notice as you scroll through, the animation is tied to the scrollbar. So when you drag down or back up, it's playing forward or reverse through the animation. Nice!
   - The animation starts at the bottom of the window and continues all the way to the top of the window, so we barely see the completed animation.

Starting & Ending Scroll Positions

1. Add **start** and **end** positions (don't miss the comma):

```
scrollTrigger:{
   trigger:'h2',
   markers:true,
   scrub:true,
   start:'top 80%',
   end:'bottom 20%'
},
```

   - The first value is the position relative to the element being animated.
     The second value is the position relative to the top of the scroller/viewport.
   - In this example the animation will **start** when the **top** of the heading hits **80%** down from the top of the window, and it will **end** when the **bottom** of the heading hits **20%** down from the top of the window.

   You can use: top, center, bottom, pixels, or percentages (always relative to the top).

2. Save and reload the page in Chrome and notice the following:

   - The **scroller-start** is 80% down from the top of the window.
     The **scroller-end** is 20% down from the top.
   - As you scroll, when **start** meets the **scroller-start** the animation will start.
     When **end** meets the **scroller-end** the animation have ended.
   - Scrubbing can cause the animation to be a bit harsh or jerky at times, so let's smooth things out by adding a time for scrub.

3. Change the **scrub** value to **2** (seconds):

```
scrollTrigger:{
   trigger:'h2',
   markers:true,
   scrub:2,
   start:'top 80%',
   end:'bottom 20%'
},
```

4. Save and reload the page in Chrome.

   Notice when you drag it will take 2 seconds to catch up to the scrollbar, smoothing out the animation. Smooth!

5. Now that we're done, delete the markers:

```
scrollTrigger:{
   trigger:'h2',
   scrub:2,
   start:'top 80%',
   end:'bottom 20%'
},
```

6. Save and reload the page in Chrome.

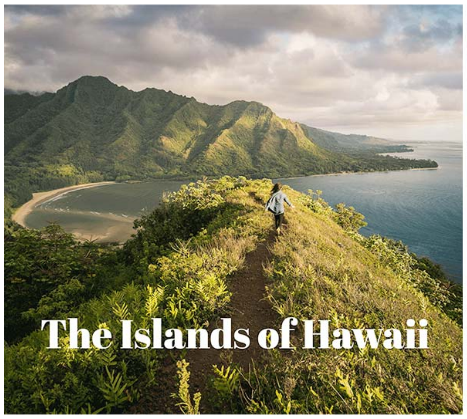
   Enjoy the final scrolling animation!

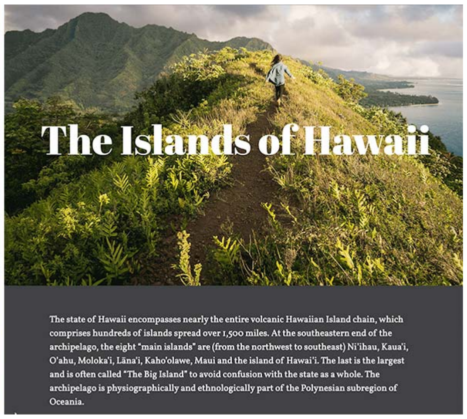
   TIP: This example added scrollTrigger to a single tween, but for more complicated animations it can also be added to a gsap.timeline() where we've put the **repeat** property in previous exercises.

   Learn more about ScrollTrigger in the GSAP Docs at **greensock.com/docs/Plugins/ScrollTrigger**

GreenSock: Parallax Animation

Exercise Preview





The state of Hawaii encompasses nearly the entire volcanic Hawaiian Island chain, which comprises hundreds of islands spread over 1,500 miles. At the southeastern end of the archipelago, the eight "main islands" are (from the northwest to southeast) Ni'ihau, Kaua'i, O'ahu, Moloka'i, Lāna'i, Kaho'olawe, Maui and the island of Hawai'i. The last is the largest and is often called "The Big Island" to avoid confusion with the state as a whole. The archipelago is physiographically and ethnologically part of the Polynesian subregion of Oceania.

Exercise Overview

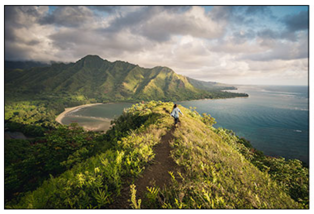
In this exercise you'll learn how to create a parallax scrolling animation.

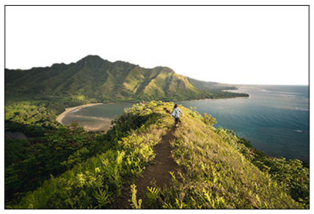
What is Parallax & How Does it Work?

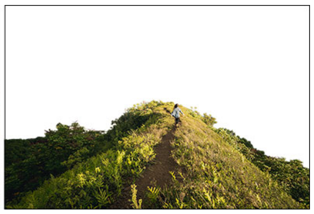
Parallax is a visual effect where multiple layers move at different rates of speed to create a sense of three dimensional depth.

The key to making parallax convincing, is having 3 or more layers on top of each other, where far away objects move slower and as objects get closer to you, they move faster. It's the different rates of speed which create the sense of depth.

One of the hardest parts of creating a parallax animation, is the Photoshop work. We've already done this for you, but you'll need each image with the unwanted elements removed (transparent). To get them into your webpage with transparent backgrounds, you must save them as PNG. Below is how our 3 images (background.jpg, middleground.png, and foreground.png) look:







6B GreenSock: Parallax Animation

Getting Started

1. For this exercise we'll be working with the **GSAP-Parallax** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).

2. In your code editor, open **index.html** from the **GSAP-Parallax** folder.

3. Preview **index.html** in a browser.

   - Notice we've already added the full background image (which is not transparent), and a heading below that.

   - We'll need to add the other transparent images over this image, and move the heading up on top of those layers.

4. Leave the page open in the browser, so you can reload it later.

Setting Up the HTML

1. Back in your code editor, find the **hero** div and we'll see the background image and heading you just saw in the browser:

```
<div id="hero">
   <img src="img/background.jpg">
   <h1>The Islands of Hawaii</h1>
</div>
```

2. Copy and paste the image code twice, and change their **src** so you have all 3 parallax images we'll be layering on top of each other:

```
<div id="hero">
   <img src="img/background.jpg">
   <img src="img/middleground.png">
   <img src="img/foreground.png">
   <h1>The Islands of Hawaii</h1>
</div>
```

3. Save and reload the page in the browser.

   - From the top of the page, scroll down and notice you can see all the images one below the other.

   - We need to put them on top of each other in the same place (in a front to back layering) so they'll initially look like a single image (users won't know there are multiple layers).

Styling the Parallax Layers

1. In your code editor, from the **css** folder open **main.css**
2. Above the **#content** rule, add this new rule to make the **#hero** container fill the screen's height:

```
#hero {
   height: 100vh;
}
```

3. Below the **#hero** rule, add the following new rule to size and position any element inside the hero on top of each other (remember we have images and a heading):

```
#hero * {
   height: 100vh;
   width: 100%;
   position: fixed;
   object-fit: cover;
}
```

   NOTE: * is a wildcard which targets any element. **`object-fit: cover;`** works like **`background-size: cover;`** but for regular images instead of background images.

4. Save and reload the page in the browser.
   - Notice that you can't scroll down to see what's below the hero, but the images and heading remain fixed to the window with all layers stacking on top of each other. So at least their layering is now working.
   - Don't worry that scrolling doesn't work, the GSAP code we'll add soon will animate them on scroll.
   - Notice the heading is at the top of the window. Let's move it to the bottom.

Positioning the Heading at the Bottom

1. Back in your code editor, below the **`#hero *`** rule, add the following:

```
#hero h1 {
   display: flex;
   align-items: flex-end;
   justify-content: center;
   padding-bottom: 15vh;
}
```

2. Save and reload the page in the browser.
   - Notice the heading is now at the bottom of the window.
   - With the page styled properly, now we can add the scrolling animation.

Adding the Depth Info

The GSAP script we'll be using requires us to tell it a depth for each layer, so elements that are farther back will move slower, and elements that are closer will move faster. For this we'll use a data attribute in HTML. They are a way to add info (data) which you make up, that JavaScript can use.

1. In **index.html** add the **data-depth** attributes shown below in bold:

```
<div id="hero">
   <img data-depth="0.25" src="img/background.jpg">
   <img data-depth="0.5" src="img/middleground.png">
   <img data-depth="1" src="img/foreground.png">
   <h1 data-depth="1.5">The Islands of Hawaii</h1>
</div>
```

NOTE: **data-depth** is a made-up attribute we're using to pass info to JS. For this script, 0 would be the farthest back and would not animate. The higher the number the faster it will move, therefore appearing closer to us. If you're not happy with the speed of an element, you could tweak these numbers to your liking. You can also have as many layers as you want!

Using GSAP to Make the Parallax Work

1. We need to link to the GSAP JavaScript file and the ScrollTrigger plugin. Add the following bold code before the closing **</body>** tag:

```
   <script src="js/gsap.min.js"></script>
   <script src="js/ScrollTrigger.min.js"></script>
</body>
```

2. Next we'll link to a JS file with some GSAP code (which you'll look at in a moment):

```
   <script src="js/gsap.min.js"></script>
   <script src="js/ScrollTrigger.min.js"></script>
   <script src="js/parallax.js"></script>
</body>
```

3. Save the file.

4. From the **js** folder, open **parallax.js**

   This code was posted to the GSAP Forums **tinyurl.com/gsap-parx** by Jack Doyle, the creator of GSAP.

5. All we have to do is point it at the correct IDs or classes we used our HTML. Change **`.header`** to **`#hero`** which points to our container.

```
const tl = gsap.timeline({
   scrollTrigger: {
   trigger: "#hero",
```

6. Change **`.header__background-layer`** to **`#hero *`** which points to the items inside the container (just like we did in the CSS).

```
gsap.utils.toArray("#hero *").forEach(layer => {
```

7. Save the file and close it.
8. Reload the page in the browser.
   - Starting at the top of the page, watch the photo and heading as you scroll down.
   - The parallax effect should be working!
   - We have a problem though, we see the bottoms of the background images which aren't moving as fast as the layers in front.
   - The parallax images are also covering the content lower in the page, so we have some fixing to do.

Moving the Parallax Content Behind the Page Content

First let's move the content backwards so it's behind the other page content. By default **z-index** is **0**, so we have to make it less than **0** (in this case **-1** will work) to move it backwards in the front to back ordering.

1. In **main.css** add the following bold code to the **`#hero *`** rule:

```
#hero * {
   height: 100vh;
   width: 100%;
   position: fixed;
   z-index: -1;
   object-fit: cover;
}
```

2. Save and reload the page in the browser.
   - Now the page content (the paragraph below the photo) is in front the parallax images, but we can see through the content to the photos behind it.
   - Setting overflow to hidden on the hero won't work, because the elements inside are position fixed which makes them relative to the viewport (not their parent element).

     We'll have to set a background color on the content to hide what's behind.

3. Back in your code editor, add the following bold code to the existing **#content** rule.

```
#content {
   background: var(--bg-color);
   CODE OMITTED TO SAVE SPACE
}
```

NOTE: We're using a color variable we've already defined for this page, but you could use a standard hex value here as well.

4. Save and reload the page in the browser.

Starting at the top of the page, scroll down and enjoy the finished parallax animation!

Reducing the File Size of PNGs

This technique relies on PNG files for their transparency. At such a large pixel width, their filesizes can be HUGE!

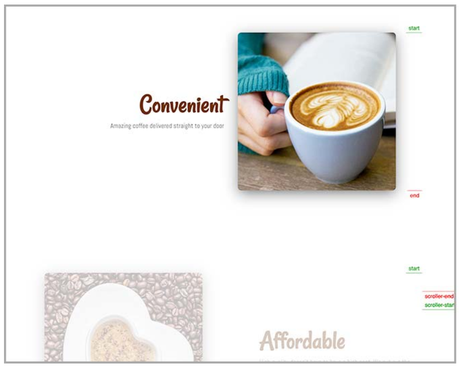
We highly recommend running PNG files through a compressor like tinypng.com which can reduce their filesize compared to what you'll get out of most apps such as Photoshop.

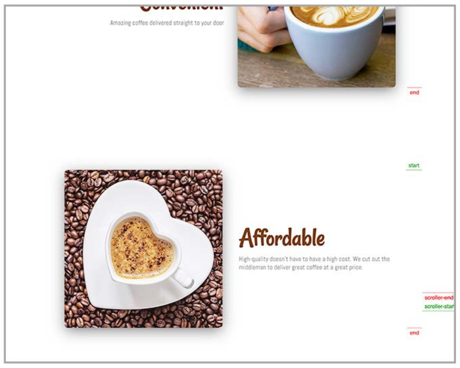
Making Sense of the Provided GSAP Code

If you're trying to make sense of the GSAP code provided, here are some things to note:

- **forEach** is a way to loop over all array items. It's like doing a **for** loop but with less code.
- **offsetHeight** is JavaScript (not GSAP) which is a read-only property that returns the height of an element (including vertical padding and borders if there are any).
- **`y: () =>`** is a GSAP function-based value that does the calculations inside so whenever the ScrollTrigger refreshes (when the window is resized) and invalidates that timeline, it calls that function again and processes the new values accordingly (as stated by Jack Doyle who wrote this code).

Exercise Preview




Convenient
Amazing coffee delivered straight to your door
end
start
Affordable
High-quality doesn't have to have a high cost. We cut out the middleman to deliver great coffee at a great price.
scroller-end
scroller-start
end

Exercise Overview

In this exercise, you'll create apply ScrollTrigger animations to multiple elements.

Getting Started

1. For this exercise we'll be working with the **GSAP-ScrollTrigger-Multiple** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
2. In your code editor, open **index.html** from the **GSAP-ScrollTrigger-Multiple** folder.
3. Preview **index.html** in a browser.
   - Scroll down and notice the alternating sections that each have a photo with text.
   - We'd like these sections to animate in as we scroll down to them.
4. Leave the page open in the browser, so you can reload it later.

Setting Up a ScrollTrigger Animation

1. We're already linked this page to the GSAP core and ScrollTrigger plugins, so we're ready to start coding our GSAP animation. At the bottom of **index.html**, in the **script** tag add a tween that fades in the image as it moves in a bit from the left:

```
<script>
   gsap.from('section img', {scrollTrigger:'section img', x:-100, opacity:0});
</script>
```

2. Save and switch to the browser.

   - Scroll up to the top and reload the page.
   - Scroll down and see the first image slides in the from the left as it fades in.
   - Scroll down to the next image with the heading **Convenient**. If possible, make it so you can see some of this photo and the one above (or below it).
   - Reload the page to see that all the images are actually animating at the same time. That's not what we want. This is one of the most common ScrollTrigger mistakes, which you can read about at **greensock.com/st-mistakes**

     The solution is to apply a separate ScrollTrigger to each image, which we can do by looping over them. But first let's perfect this animation.

3. To set additional scrollTrigger options, we need to change it into an object by changing the **scrollTrigger** from **'section img'** to **{}** as shown below:

```
gsap.from('section img', {scrollTrigger:{}, x:-100, opacity:0});
```

4. To make things easier to read, add some lines breaks:

```
gsap.from('section img', {
   scrollTrigger:{},
   x:-100, opacity:0
});
```

5. Put the **'section img'** trigger back in, and add the markers so we can see what's going on:

```
gsap.from('section img', {
   scrollTrigger:{trigger:'section img', markers:true},
   x:-100, opacity:0
});
```

6. Change when the animation starts and stops, and link it to the scrollbar so we can scrub through the animation:

```
gsap.from('section img', {
   scrollTrigger:{trigger:'section img', start:'top 85%', end:'bottom 80%',
scrub:1, markers:true},
   x:-100, opacity:0
});
```

7. Save and switch to the browser.

   - Scroll up to the top and reload the page.
   - Scroll down and notice the first image scrolls on (its animation scrubs with the scrollbar), and finishes just a little after it's fully visible on screen.

Making ScrollTrigger Work with Each Image Individually

Our scroll animation works, but it's changing all images at the same time. Let's use a loop to add a ScrollTrigger to each image.

1. Create a variable for the array (list) of all the images.

```
<script>
   let sectionImg = document.querySelectorAll('section img');
```

2. Wrap a **for** loop around the scrollTrigger animation:

```
let sectionImg = document.querySelectorAll('section img');

for(let i=0; i < sectionImg.length; i++) {
   gsap.from('section img', {

   CODE OMITTED TO SAVE SPACE

   });
};
```

3. Change the element we're targeting to the image in our array, based on the loop's counter (i). Be sure to not to leave quotes around the variable/array name:

```
for(let i=0; i < sectionImg.length; i++) {
   gsap.from(sectionImg[i], {
      scrollTrigger:{trigger:sectionImg[i], start:'top 85%', end:'bottom 80%', 
scrub:1, markers:true},
      x:-100, opacity:0
   });
};
```

4. Save and reload the page in the browser.

- Starting at the top of the page, scroll down and notice that each image animates on when it comes onto the screen.
- It's great that our animation is now working with each image, but we don't like how they always animate left to right. Because the images alternate, we'd prefer them always move in toward the center (later we'll do the same thing with the text so the text and photo will come together).

Alternating Directions For Each Row

1. To alternate the direction, we'll have to alternate our x value between positive and negative amounts. Store the current x amount (100) as a variable:

```
for(let i=0; i < sectionImg.length; i++) {
   let movement = 100; // odd rows
   gsap.from(sectionImg[i], {

   CODE OMITTED TO SAVE SPACE

   });
};
```

NOTE: You may notice our x value in the ScrollTrigger animation is currently negative 100, but just wait a moment to see how this is going to work.

2. Swap the hard-coded value (**-100**) for our new **movement** variable:

```
for(let i=0; i < sectionImg.length; i++) {
   let movement = 100; // odd rows
   gsap.from(sectionImg[i], {
      scrollTrigger:{trigger:sectionImg[i], start:'top 85%', end:'bottom 80%', 
scrub:1, markers:true},
      x:movement, opacity:0
   });
};
```

3. Add a test to detect whether the loop's counter (i) is odd or even:

```
for(let i=0; i < sectionImg.length; i++) {
   let movement = 100; // odd rows
   if( i % 2 == 0 ) {
      movement = -movement; // even rows
   };
   gsap.from(sectionImg[i], {
```

NOTE: % is a "remainder operator" which returns the remainder (what's left over) when one number is divided by another number. Even numbers do not have a remainder (so the remainder would be 0, like we're testing for above), while odd numbers do have a remainder (so the remainder would not be 0).

Setting a variable to a negative of itself with either change a positive value into a negative, or change a negative value into a positive.

Remember that arrays start counting with 0, which is an even number (1 is the next number and it's odd). So **`sectionImg[0]`** will be **`-100`** (just like we originally started with) and then **`sectionImg[1]`** will be **`100`**.

4. Save and reload the page in the browser.
   - The row animations should now alternate, with the images always moving inward toward the text.
   - Now let's apply the same animation to the text, keeping in mind it will need to animate in the opposite direction, so the text and photo will both be moving inward toward each other.

Animating the Text

1. Create a variable for the array of all the text:

```
<script>
   let sectionImg = document.querySelectorAll('section img');
   let sectionText = document.querySelectorAll('section .text');
```

2. Copy and paste the **gsap.from()** animation so you end up with two of them:

```
for(let i=0; i < sectionImg.length; i++) {

   CODE OMITTED TO SAVE SPACE

   gsap.from(sectionImg[i], {
      scrollTrigger:{trigger:sectionImg[i], start:'top 85%', end:'bottom 80%',
scrub:1, markers:true},
      x:movement, opacity:0
   });
   gsap.from(sectionImg[i], {
      scrollTrigger:{trigger:sectionImg[i], start:'top 85%', end:'bottom 80%',
scrub:1, markers:true},
      x:movement, opacity:0
   });
};
```

3. In the second **gsap.from()** change **sectionImg** to **sectionText**

```
   gsap.from(sectionImg[i], {
      scrollTrigger:{trigger:sectionImg[i], start:'top 85%', end:'bottom 80%',
scrub:1, markers:true},
      x:movement, opacity:0
   });
   gsap.from(sectionText[i], {
      scrollTrigger:{trigger:sectionText[i], start:'top 85%', end:'bottom
80%', scrub:1, markers:true},
      x:movement, opacity:0
   });
};
```

4. In the second **gsap.from()** add a minus (-) in front of **movement** to make it the opposite direction as the image (if one is positive the other will be negative, or vice versa):

```
   gsap.from(sectionImg[i], {

   CODE OMITTED TO SAVE SPACE

   });
   gsap.from(sectionText[i], {
      scrollTrigger:{trigger:sectionText[i], start:'top 85%', end:'bottom 
80%', scrub:1, markers:true},
      x:-movement, opacity:0
   });
};
```

5. Save and reload the page in the browser.

   Now the text and images should both animate inward towards each other, which looks great!

Hiding the Markers

1. We no longer need to see the **markers** so in both places change **true** to **false**

   NOTE: We often like to keep the marker property there in case we want to show them again in the future)

2. Save and reload the page in the browser.
   - The page is looking good now without the markers.
   - Reduce the width of the browser window until the photos start scaling down.
   - You should now be able to scroll horizontally a little bit, even though we didn't intentionally make anything wider than the page.

   Our page didn't scroll horizontally before we added the scrolling animation, so why is this happening? It's because the elements are starting from farther outside and animating into the page, which is causing the page to get wider. We'll have to hide the "overflow" that gets created to prevent this unwanted scrolling.

Fixing Unwanted Horizontal Scrolling

1. In your code editor, from the **css** folder open **main.css**

2. Near the top, above the **body** rule, add the following new rule:

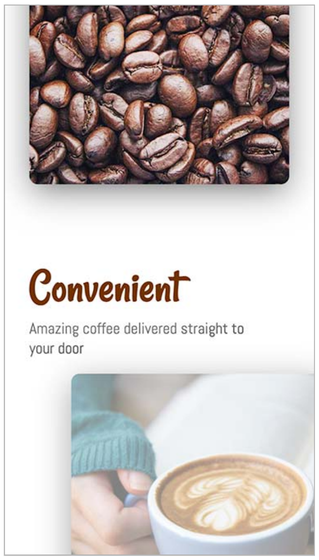
```
html, body {
   overflow-x: hidden;
}
```

3. Save and reload the page in the browser.

   - Notice the horizontal scrolling problem is fixed.
   - Congrats, you can now enjoying the finished animation scrolling animation!

GreenSock: ScrollTrigger Media Queries with matchMedia() B1

Exercise Preview



Exercise Overview

In this exercise, you'll learn how to change ScrollTrigger based on different screen sizes (like we do with media queries in CSS).

Getting Started

1. For this exercise we'll be working with the **GSAP-ScrollTrigger-Multiple** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).
2. In your code editor, open **index.html** from the **GSAP-ScrollTrigger-Multiple** folder.
3. Preview **index.html** in a browser.
   - This is the scrolling animation we created in the previous exercise.
   - It looks great when the window is wide, but resize the window to be narrow so the text stacks on top of the photo (and not very tall to simulate a phone size).
   - The scrolling animations still work. They look OK, but could be improved:
     – The text animates very close to the bottom, we'd prefer it to animate a bit higher up in the page.
     – The image takes a long time to finish animating. We'd prefer it to finish earlier to we see the finished image for more time.
4. Leave the page open in the browser at this small size, so you can reload it later.

Using ScrollTrigger.matchMedia() to Define Different Animations for Different Screen Sizes

We want to use essentially the same animations on all screen sizes, but to tweak the settings a bit for smaller screens.

1. In the script tag, add **ScrollTrigger.matchMedia()** as shown below in bold (with an object **`{}`** inside, into which we'll be adding options).

```
<script>
   let sectionImg = document.querySelectorAll('section img');
   let sectionText = document.querySelectorAll('section .text');

   ScrollTrigger.matchMedia({});

   for(let i=0; i < sectionImg.length; i++) {
```

2. Inside matchMedia(), add the following 2 parameters, which essentially act like media queries in CSS, but these are for our JS!

```
ScrollTrigger.matchMedia({
   '(min-width: 600px)': function() {

   },
   '(max-width:599px)':function() {

   }
});
```

3. Cut the entire **for** loop and paste it into **both** functions, as shown below.

```
<script>
   let sectionImg = document.querySelectorAll('section img');
   let sectionText = document.querySelectorAll('section .text');

   ScrollTrigger.matchMedia({
   '(min-width: 600px)': function() {
      for(let i=0; i < sectionImg.length; i++) {

      CODE OMITTED TO SAVE SPACE

      };
   },
   '(max-width:599px)':function() {
       for(let i=0; i < sectionImg.length; i++) {

      CODE OMITTED TO SAVE SPACE

      };
   }
});
</script>
```

4. In the bottom **'(max-width:599px)'** function, change the **start** and **end** values as shown below in bold:

```
'(max-width:599px)':function() {
   for(let i=0; i < sectionImg.length; i++) {

      CODE OMITTED TO SAVE SPACE

      gsap.from(sectionImg[i], {
         scrollTrigger:{trigger:sectionImg[i], start:'top 75%', end:'150px
70%', scrub:1, markers:false},
         x:movement, opacity:0
      });
      gsap.from(sectionText[i], {
         scrollTrigger:{trigger:sectionText[i], start:'top 75%', end:'bottom
70%', scrub:1, markers:false},
         x:-movement, opacity:0
      });
```

5. Save and switch to the browser.
   - Make sure the window still narrow and short as we previously left it.
   - Reload the page.
   - Scroll through the animations and notice:
     – We think it looks better with the text animation starting slightly higher up.
     – The image animation finishes quicker so you see the finished image for longer (because its end point is now 150px from the top of the photo, instead of the bottom of the photo as it on wider screens).
6. Resize the window wider and notice the images and text disappear! What's wrong?

   GSAP does all its animation as inline CSS. When resizing the window the inline CSS from one media query is remaining and contaminating the other media. Luckily GSAP has a quick fix.

Fixing Inline Style Contamination Across Media Queries

ScrollTrigger.saveStyles() records the initial inline CSS for the specified elements so that ScrollTrigger can revert them even if animations add inline styles later. All it needs to know is which items you're animating, and it will save those styles to prevent contamination across media queries.

1. Add the following bold line of code, which will save the styles for the two types of elements we're animating:

```
<script>
   let sectionImg = document.querySelectorAll('section img');
   let sectionText = document.querySelectorAll('section .text');

   ScrollTrigger.saveStyles('section img, section .text');
   ScrollTrigger.matchMedia({});
```

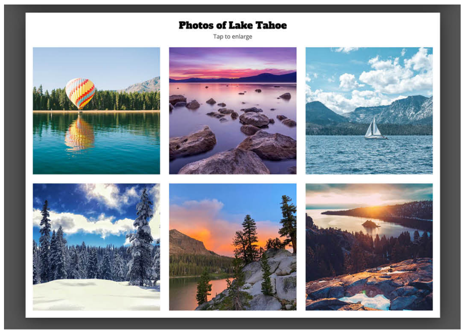
2. Save and reload the page in the browser.
   - Scroll through the animation when the window is wide and narrow.
   - Resizing the window no longer creates a problem, and the animations look good at every size now.

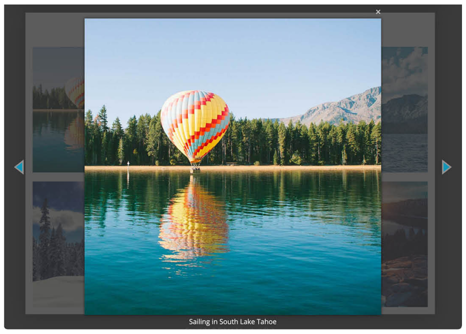
   Congrats you're all done!

   Visit the GSAP Docs **tinyurl.com/gsap-matchmedia** to learn more about ScrollTrigger.matchMedia()

Using a jQuery Plugin: Lightbox (A Pop-up Image Viewer) B2

Exercise Preview





Exercise Overview

In this exercise, you'll learn how to use a free jQuery lightbox plugin called **Magnific Popup**. When a user clicks a thumbnail image to see a larger version, JS can create a "lightbox" effect. The images open within the page, and you can easily switch between photos with buttons/keystrokes, and then close it.

About jQuery

jQuery is a free JavaScript framework (pre-written JavaScript) that has been widely for many years. There are many jQuery plugins (JavaScripts written using jQuery) to create various kinds of functionality.

With improvements in JavaScript we no longer need jQuery as much, so it's use is starting to decline. Most developers are not creating new things with jQuery. But there are a still useful jQuery plugins which work great, so it's useful to know how to use them. If you'll be working with premade WordPress themes, many of them will likely use jQuery plugins, so again it's useful to be familiar with them.

Getting Started

1. For this exercise we'll be working with the **Lightbox-Gallery** folder located in **Desktop > Class Files > JavaScript Class**. Open that folder in your code editor if it allows you to (like Visual Studio Code does).

2. In your code editor, open **photos.html** from the **Lightbox-Gallery** folder.

3. Preview **photos.html** in Chrome. (We'll be using its DevTools.)

   Click some of the photos in the gallery to see they are just links to a large picture. They are not pages, so each browser can display them differently. Let's improve the look and functionality.

4. Leave the page open in Chrome so we can come back to it later.

Linking to the Plugin Files

We've included the **Magnific Popup** file in our class files, but you can also visit dimsemenov.com/plugins/magnific-popup to download it, read the documentation, and see examples.

At a minimum, Magnific Popup requires 3 files to run—**jQuery**, the **Magnific Popup JS file**, and the **Magnific Popup CSS file**. Let's link to the CSS file first.

1. Return to **photos.html** in your code editor.
2. It's important to link to the Magnific Popup CSS **before** our custom CSS. This way our CSS will override any styling in the plugin. Above the link to **main.css**, link to the Magnific Popup CSS file as shown in bold:

```
   <link rel="stylesheet" href="css/normalize.css">
   <link rel="stylesheet" href="js/vendor/magnific-1.1.0/magnific-popup.css">
   <link rel="stylesheet" href="css/main.css">
</head>
```

3. Next we'll link to the jQuery JS file. At the bottom near the end of the body tag, above the link to our own main.js file, add the following bold code:

```
   <script src="js/vendor/jquery-3.6.0.min.js"></script>
   <script src="js/main.js"></script>
</body>
```

4. Next we'll link to the Magnific Popup JS file. Below the jQuery link, add the following bold code:

```
<script src="js/vendor/jquery-3.6.0.min.js"></script>
<script src="js/vendor/magnific-1.1.0/jquery.magnific-popup.min.js"></script>
<script src="js/main.js"></script>
```

5. Save the file.

Initializing the Pop-up

1. We've linked to all the files, so we're ready to initialize Magnific Popup. We'll start with a prepared code snippet we got from the documentation on the Magnific Popup website. From the **snippets** folder, open **magnific-initialize.js**.
2. This code initializes the **magnificPopup()** function. When you click a link with an **image-link** class (which is just an example), it will load the linked image into the pop-up "lightbox".

   Select and copy all the code.

3. From the **js** folder, open **main.js**.
4. Paste the code.
5. All our thumbnail image links reside in a div which has class of **photo-gallery**, so change the code as shown below in bold:

```
$('.photo-gallery a').magnificPopup({type:'image'});
```

6. Save and reload the page in Chrome.
   - Click any of the image thumbnails and a large image should pop up. Sweet!
   - Below the photo notice there's a caption. Later we'll explain how that got there.
   - To close the image, click the close X at the top right of the image, or click in an empty area of the page outside the image.

Grouping the Photos into a Gallery

It would be nice if the photos were grouped together so that the user could go between them with next and previous buttons (or keystrokes). We can do that Magnific Popup.

1. Switch back to **main.js** in your code editor.
2. Inside the options object {}, add returns to put the setting on it's own line, so you end up with the following:

```
$('.photo-gallery a').magnificPopup({
    type:'image'
});
```

3. Add the following bold code:

```
$('.photo-gallery a').magnificPopup({
   gallery: {
      enabled: true
   },
   type: 'image'
});
```

4. Save and reload the page in Chrome.
   - Click on any photo thumbnail to see the pop-up.
   - There should now be arrows on the left and right. Click them to change images.

5. Press the **Left** or **Right Arrow** key on the keyboard. Nice! Users can navigate through the images with the Arrow keys or by clicking the buttons.
   - On the bottom right there's now an image counter.
   - Notice the image, caption, and counter all change as you change images.
6. There's one more way to change photos. Click on the enlarged photo to advance to the next image.
7. Press the **Esc** (**Escape**) key to close the enlarged image.

Captions

Magnific Popup can add captions to the enlarged images. To do this it looks at the **title** attribute of the link that is clicked. If the link has a **title**, it will add it as a caption. If there is no title, then there's no caption.

1. Switch back to **photos.html** in your code editor.
2. On the first link (around line 36), notice there's a **title** attribute as shown below:

```
<a href="img/gallery/enlargements/hot-air-baloon.jpg" title="Sailing in South
Lake Tahoe">
```

This **title** attribute is used for two purposes:

- When you hover over something in a title attribute and pause for a moment, desktop browsers will display a tooltip with that text.
- Magnific Popup uses the title attribute's text as the caption for the enlarged photo.

Removing the Counter

We like the counter, but what if you want to remove it? Magnific Popup has an option to customize the text of the counter. Instead of changing the text to something else, we'll simply set it to nothing at all!

1. Switch back to **main.js** in your code editor.
2. To remove the counter, we need to customize its text. By setting the text to nothing, we'll remove it. Add the following bold code, but don't miss the comma at the end of the line above it!

```
gallery: {
   enabled: true,
   tCounter: '' // those are 2 single quotes!
},
type: 'image'
```

3. Save and reload the page in Chrome.
   - Click on the any thumbnail image.
   - The counter should not be there anymore.

   Keep the lightbox open, we're not done with it yet.

Customizing the Look of the Captions

We can change the styling of the captions with CSS. First we need to find out how Magnific Popup is styling them.

1. The caption should be visible on the bottom left of the enlarged image. **Ctrl–click** (Mac) or **Right–click** (Windows) on the caption text and choose **Inspect**.
2. In the DevTools you should be able to see that the caption is wrapped by **`<div class="mfp-title">`** which we can target for styling.
3. While we're in the DevTools, make sure the **.mfp-title** div is selected, and in the **Styles** panel notice the CSS. You should see it has **padding-right: 36px;** (to make room for the counter).

   We want to center the text, so now that we've removed the counter, we must equalize the left and right padding or the text won't be properly centered.
4. Switch back to your code editor.
5. From the **css** folder, open **main.css**.
6. Above the **min-width: 600px** media query (around line 92), add the following:

```
.mfp-title {
   text-align: center;
   font-size: 1.2rem;
   padding: 5px;
}

@media (min-width: 600px)
```

7. Save and reload the page in Chrome.
   - Click any thumbnail image.
   - Notice the caption should now be centered, larger than before, with a little more space above (and equal padding on the left/right so it's truly centered).

   Keep the lightbox image open, we're not done with it yet.

Customizing the Overlay's Background Color

Magnific Popup adds an overlay behind the enlarged photo to help separate it from the page's content. We can change the color of that, and we'd like to lighten it.

1. With the lightbox still open, **Ctrl–click** (Mac) or **Right–click** (Windows) outside the enlarged image (on the background) and choose **Inspect**.

2. In the DevTools, find the first tag inside the **body** tag. It should be
   `<div class="mfp-bg mfp-ready"></div>`

3. Select that tag and in the **Styles** panel, find the **.mfp-bg** rule.

   Notice it has a **background** of **#0b0b0b**

4. Switch back to **main.css** in your code editor.

5. Below the **.mfp-title** style, add the following:

```
.mfp-bg {
   background: #333;
}
```

6. Save and reload the page in Chrome.

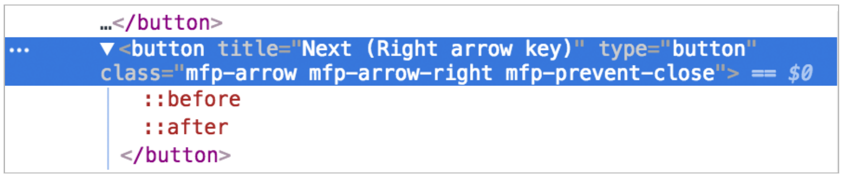
   Click on any thumbnail image and notice the background is now a lighter gray.

Optional Bonus: Customizing the Look of the Arrows

With the lightbox still open, notice the color of the left/right arrows. Let's see how we could change that color.

1. **Ctrl–click** (Mac) or **Right–click** (Windows) on the lightbox's **right arrow (next)** and choose **Inspect**.

2. In the DevTools, expand the **button** element to see the **::before** and **::after** elements inside.



```
      …</button>
...   ▼<button title="Next (Right arrow key)" type="button"
      class="mfp-arrow mfp-arrow-right mfp-prevent-close"> == $0
         ::before
         ::after
       </button>
```

   NOTE: The **::before** element is the outline and the **::after** element is the fill.

3. Select the **::before** element.

   - In the **Styles** panel, find the **.mfp-arrow-right:before** rule and notice that it has **border-left: 27px solid #3f3f3f;**

4. Select the **::after** element.

   - In the **Styles** panel, find the **.mfp-arrow-right:after** rule and notice that it has **border-left: 17px solid white;**

   To adjust the color of the fill and outline we can change these two elements. If we made you manually style these, you'd copy and paste several class names, so we've saved you some time and made a snippet.

5. Switch back to your code editor.

6. From the **snippets** folder, open **magnific-arrow-colors.css**.

7. Select and copy all the code.

8. Switch back to **main.css**.

9. Paste the new rules below the **.mfp-bg** rule.

   NOTE: You can use this snippet in your own projects, but you'll want to customize the colors to fit your design.

10. Save and reload the page in Chrome.

    - Click any thumbnail image and notice the arrows are now filled with blue instead of gray.

    - Hover over them and notice the opacity changes so they appear brighter.

Fancybox: Another Good Lightbox

There are tons of lightboxes, some which use jQuery and some which don't. They all work pretty similarly, but we're not thrilled by all of them. Some aren't attractive, or have slow animations that are annoying as a user. One lightbox we particularly like is Fancybox. It does not require jQuery and you can learn more at fancyapps.com. You can try it out for free, but will have to pay a small price if you want to use it.

Noble Desktop Training

Learn live online or in person in NYC

Front-End Web Development
Full-Stack Web Development
JavaScript
Python
Software Engineering
Data Science & Data Analytics
SQL
WordPress

Motion Graphics & Video Editing
Adobe Premiere Pro
Adobe After Effects
InDesign, Illustrator, & Photoshop
Web, UX, & UI Design
Figma, Adobe XD, & Sketch
Digital & Social Media Marketing
and much more...

noblesdesktop.com

Common JavaScript Terms Defined

Here's a concise list of common JavaScript terms/concepts and their definitions.

- **Arguments vs. Parameters**
  When defining a function, you can request information to be passed to it (so you can use it in the function). When defining a function, it's written as:
  `function myFunctionName(parameter1,parameter2)`

  Later when calling the function, you pass **arguments** to those **parameters** as:
  `myFunctionName(argument1,argument2)`

- **Array**
  Like an object, it's a collection of things such as strings, numbers, booleans, functions, objects, or other arrays. Unlike an object, you don't get to name the keys. Keys are numbered automatically in ascending order starting from 0.

- **Concatenation**
  Putting strings together is called concatenation. The strings are put together, one after the other, for instance: **`string1 + string2 = string1string2`**

- **Function**
  Defines a group of code. This code is not executed until it is called.

- **Key-Value Pair**
  How objects are typically organized. Key-value pairs have a **key** (attribute) followed by a **colon** and the **value** associated with the key. You can name a key anything as long as it doesn't start with a number. Multiple key-value pairs are comma-delimited except for the last one. As shown in the bold key-value pairs below, you can assign any type of value to a key, such as strings, numbers, and booleans:

  ```
  let myObject = {
     name: 'Bob',
     age: 23,
     alive: true
  }
  ```

- **Loop**
  A chunk of code that is used for quickly repeating a task many times in a row. The `for` loop is the most common. It takes three pieces of information in this order: a **counter** variable (usually named `i`), a **condition** to test, and an **incrementer**. If the condition is true, the instructions in curly braces will execute every time the loop runs. The syntax is shown below:

  ```
  for (counter; condition; incrementer) {
     // Code to execute if the condition is true
  }
  ```

- **Method**
  These are like the verbs of JavaScript. They are built-in actions that do various tasks. They are written as: **`methodName()`**.

- **Object**
  A bundle of information that is packaged in an organized way and has key-value pairs that you write. Nearly everything in JavaScript is an object. You can name an object anything as long as it doesn't start with a number.
- **Property**
  Objects have properties. These are like the adjectives of JavaScript. They tell you things about an object. You can get these properties and sometimes you can change the properties. They are written as **`Object.property`**.
- **Single Quotes vs. Double Quotes**
  JavaScript doesn't care which you use, except when nesting. For instance, if you are dealing with a string of HTML code that contains quotes, be sure to balance them so the outer quotes are not used anywhere in the string. For instance:
  `myCode = '<div id="myDiv">content here</div>'`
- **String**
  Strings are stored pieces of text. They are looked at as a group of characters.
- **Unobtrusive JavaScript**
  A set of best practices for using JavaScript on a website. Under this paradigm, you should keep your JavaScript functionality and HTML markup completely separate and make sure all the content is available even if users have disabled JavaScript.
- **Variable**
  Variables hold information. They can hold numbers, text (strings), etc. There are various ways to declare variables. In this book we used **`let`** but **const** and **var** (which is older) are others ways.
- **Library or Framework**
  JavaScript libraries or frameworks are pre-written JavaScript (that are often free) which let you use complex JavaScript functionality more easily by writing less code yourself (using the code that's already written the library). Some examples are jQuery, GreenSock (GSAP), React, but there are many more.