# Introduction to jQuery

> **Goals**
>
> By the end of this lesson, you will understand:
> - What jQuery is, and why it's useful
> - How to install jQuery and the basics of its syntax
> ### Introduction
> This week, we'll start exploring how to work with external JavaScript libraries, the different problems that they solve, and the different functionalities they help us achieve in our code. In this lesson, we'll begin by learning the basics of jQuery, which is one of the most widely used JavaScript libraries out there. We will see how we can easily include jQuery in a project, followed by a brief overview of jQuery's syntax.

# What is jQuery?

> So, what is jQuery? According to their website, jQuery is a “fast, small, and feature-rich JavaScript library” which “makes things like HTML document traversal and manipulation, event handling, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers.”
>
> jQuery was created by John Resig in 2005 when he was still a student at the Rochester Institute of Technology, but was released to the public as an open source library in early 2006.
>
> It’s worth noting that, despite all of the new-fangled JavaScript frameworks and libraries out there today, jQuery is still in use in millions of websites!
>
> ### Why learn jQuery?
>
> jQuery provides a simple and intuitive way for manipulating HTML and CSS. This ease of use is, in fact, one of the main reasons jQuery became one of the most widely used JavaScript libraries on the web in the first place.
>
> The library also provides a range of features and plugins that can help developers add advanced functionality to their web applications, including animation and form validation. With jQuery, developers can add these and other features with relatively few lines of code, saving time and effort.
>
> In addition, learning jQuery can help developers build (or solidify) a strong foundation in JavaScript fundamentals, since jQuery is built upon JavaScript, and thus can help pave the way for learning other frontend frameworks.

# Getting Started

> In order to get started with jQuery, to keep things simple, we're going to use a CDN (which stands for "Content Delivery Network" - to learn more about what these are, click here).

To use jQuery from the CDN, go ahead and copy the following line into the HTML file just *above* the other script tag linking to our main JavaScript file:

```
<script
        src="https://cdn.jsdelivr.net/npm/jquery@3.7.0/dist/jquery.min.j
        s"></script>
```

> NOTE: The order here is important! If you remember, the browser will load our HTML file line by line from top to bottom, so if the jQuery script is placed *after* our main JavaScript file it won't work!

And that's all we have to do. We can now use jQuery in our JavaScript!
>

# Syntax Overview

> In jQuery's syntax, you're going to see a lot of `$`, which represents jQuery's default exported function. This is what we will now use to reference it throughout our code. Most things in jQuery will begin with `$` followed by parentheses **()**:

```
$(/* Selector or other code */)
```

> To try this out, add the following code to **script.js**:

```
$(function() {
  $("h1").text("Hello from jQuery!");
})
```

What's happening here, to begin with, is we're using the jQuery to check that the document has finished loading (everything inside **$(function() { ... })** will execute when it has). Then we're getting our **h1** element that we added to the HTML earlier and setting its inner text to "Hello from jQuery!" If this seems confusing now, don't worry – we'll go over all of this in more detail throughout the following lessons!
>
A quick note, however: it used to be (and you may see this in many places online when looking up answers to coding questions) that the syntax for this initial check to see if the document had loaded looked like this:

```
$(document).ready(function () {
  /* code here */
})
```

However, this syntax has been deprecated since version 3.0. The code we entered a moment ago in **script.js** is the new syntax and it works in exactly the same way.
>
When you refresh the preview window, you should now see the heading text "Hello from jQuery!" on the page!

# Conclusion & Takeaways

> jQuery is a powerful JavaScript library that simplifies web development by streamlining HTML document traversing, event handling, and animation. It's beneficial to learn jQuery because it helps reduce the amount of JavaScript code you need to write. In this lesson, we covered how to include jQuery in our project using a CDN and also introduced the basics of it's syntax.

# jQuery DOM Manipulation: Introduction

> ### Goals
>
> By the end of this lesson, you will be able to:
> - Select DOM elements using jQuery selectors
> - Traverse the DOM using jQuery's traversal methods
> - Modify styles and attributes of DOM elements using jQuery
> - Add and remove DOM elements using jQuery
>
> ### Introduction
>
> You're already familiar with the Document Object Model (DOM) and how to manipulate it using vanilla JavaScript. However, using jQuery can simplify and speed up this process significantly. In this lesson, we'll be exploring DOM manipulation with jQuery: selecting DOM elements, DOM traversal, modifying styles and attributes, and adding and removing DOM elements. By the end of this lesson, you'll have a solid understanding of how to use jQuery to manipulate the DOM and enhance your web development skills. So, let's dive in and see how jQuery can make your life as a frontend developer easier and more efficient!

# Selecting Elements

> As touched upon briefly earlier, jQuery provides a simple syntax for selecting DOM elements based on CSS selectors. The syntax for selecting elements in jQuery is done using the library's default function (most often represented by a '**$**'):
>
> `$('SELECTOR')`
>
> Here, **SELECTOR** stands for a string containing a normal CSS selector. This can be a tag name, a class, an id – basically anything that is a valid selector in CSS. Let's look at a few examples.

> Let's select all the **p** elements on our page (in this case we only have one), using the following code:

```
// Select all paragraph elements (in this case, we only have
        one)
$('p')

// Select element with class 'myDiv'
$('.myDiv')

// Select element with the id 'header'
$('#header')
```

> We can also use attribute selectors to select elements based on their attributes. For example, to select all the input elements with the attribute type set to **text**, we can use the following code:

```
$('input[type=text]')
```

# DOM Traversal

> Once we have selected an element, we can use jQuery's traversal methods to move around the DOM and select other elements. The most commonly used traversal methods are **parent()**, **children()**, **siblings()**, **next()**, and **prev()**. Let's look at each of these methods in more detail.
>
> The **parent()** method selects the parent element of the selected element. For example, we can use the following code to select the element that is the parent of the **p** element:

```
$('p').parent()
```

The **children()** method selects all the child elements of the selected element. Let's add the following code to select all the **li** elements that are children of the **ul** element:
>

```
$('ul').children('li')
```

The **siblings()** method selects all the sibling elements of the selected element, whereas **next()** selects the *next* sibling element and **prev()** selects the *previous* sibling element:
>

```
// Select all siblings
$('.myDiv').siblings()

//Select next sibling
$('.myDiv').next()

// Select previous sibling
$('.myDiv').prev()
```

# Modifying Styles and Attributes

> jQuery provides several methods for adding and modifying CSS styles. These methods can be used to change the appearance of an element by adding or modifying its style properties.
>
> To set the value of a style property, we can use the **css()** method. For example:

```
$('.myDiv').css('background-color', 'red')
```

> We can also set multiple style properties at once by passing an object to the **css()** method. For example:

```
$('.myDiv').css({
   'font-size': '24px',
   'font-weight': 'bold'
 });
```

To get the value of a style property, we can use the **css()** method with only that property as an argument. For example:
>

```
const bgColor = $('.myOtherDiv').css('background-color')
```

> jQuery also provides methods for adding and modifying HTML attributes. These methods can be used to change the behavior of an element by adding or modifying its attributes.
>
> To set the value of an attribute, we can use the **attr()** method. For example:

```
$('.cool-img1').attr('src', 'https://picsum.photos/200/300')
```

> We can also set multiple attributes at once by passing an object to the **attr()** method. For example:

```
$('.cool-img2').attr({
  'src': 'https://picsum.photos/200/300?random=2',
  'alt': 'A random image'
});
```

To get the value of an attribute, we can use the **attr()** method without any arguments. For example, to get the **src** attribute of an img element, we can use the following code:
>

```
$('.cool-img2').attr('src')
```

> One other thing we can do here with jQuery is something called method chaining. As an example of this, let's say we want to change the text of the **h1** element and also change its **background-color**. We can achieve that like so:

```
$("h1").text("Hello, jQuery!").css("background-color", "yellow")
```

# Adding and Removing Elements

> jQuery also provides methods for adding and removing elements from the DOM. These methods can be used to dynamically add or remove content from a web page.
>
> To add an element to the DOM, we can use the **append()** or **prepend()** methods. The **append**() method adds an element as the last child of the selected element, while the **prepend()** method adds an element as the first child of the selected element. For example, to add a **p** element with the text “Hello World!” to a **div** element, we could do the following:

```
$('.myOtherDiv2').append('<p>Hello World!</p>')
```

> To remove an element from the DOM, we can use the **remove()** method. For example:

```
$('.myOtherDiv3').remove()
```

> We can also remove all the child elements of a selected element using the **empty()** method. For example:

```
$('.myOtherDiv4').empty()
```

# Conclusion & Takeaways

> In this lesson, we've covered how to manipulate the DOM using jQuery. We looked at selecting DOM elements, DOM traversal, modifying styles and attributes, and adding and removing DOM elements. As you know by now, these skills are essential for creating dynamic and engaging experiences on the web, and you should now have a solid understanding of how to use jQuery to achieve this. Next, we'll continue exploring the capabilities of jQuery by looking at handling events and working with animations.

# Practice Time

> Work alone or with a partner to answer the following practice questions. Try to come up with solutions first, but If you get stuck you can always refer back to the material in this lesson or by asking an instructor or TA for help. Happy coding!

1. Write jQuery code necessary to select the following elements.You don't necessarily have to create an HTML document for this exercise (just showing the jQuery code is sufficient) but you certainly can create one if you want to.
   **A.** All the h1 elements on the page
   **B.** The element with the ID navbar
   **C.** All the input elements that are descendants of the element with the class form-group
   **D.** All the li elements that are children of the element with the ID menu
   **E.** All the elements with the class btn that are siblings of the element with the ID form

> 2. Given the following HTML, write jQuery code to select the specified elements:

```
<div>
     <h1>My Website</h1>
     <ul>
       <li>Home</li>
       <li>About</li>
       <li>Contact</li>
     </ul>
     <p>Welcome to my website!</p>
   </div>
```

**A.** The **h1** element
**B.** The **ul** element
**C.** The **p** element
**D.** The **li** element with the text About

> 3. Write jQuery code to modify the following styles and attributes:
>    **A.** Set the **background color** of all div elements to blue
>    **B.** Set the **href** attribute of all **a** elements to #
>    **C.** Set the **src** attribute of all **img** elements to a random image from Picsum
>    **D.** Set the **font size** of all **p** elements to 16px
>    **E.** Get the value of the **alt** attribute of the first **img** element

4. Given the following HTML code, write jQuery code to add and remove elements:

```
<div>
   <h1>My Website</h1>
   <p>Welcome to my website!</p>
 </div>
```

**A.** Add a nav element with the following links: Home, About, Contact
**B.** Add a span element to the end of the h1 element with the text “(New)”
**C.** Remove the p element

<!–
POSSIBLE SOLUTIONS:
### 1.
**A.** `$('h1')`
**B.** `$('#navbar')`
**C.** `$('.form-group').find('input')`
**D.** `$('#menu').children('li')`
**E.** `$('#form').siblings('.btn')`
### 2.
**A.** `$('ul').prev('h1')`
**B.** `$('h1').next('ul')`
**C.** `$('ul').next('p')`
**D.** `$('li:contains("About")')`
### 3.
**A.**`$('div').css('background-color', 'blue')`
**B.** `$('a').attr('href', '#')`
**C.** `$('img').attr('src', 'image.jpg')`
**D.** `$('p').css('font-size', '16px')`
**E.** `$('img:first').attr('alt')`
### 4.
**A.** `$('div').append('<nav><a href="#">Home</a><a href="#">About</a><a href="#">Contact</a></nav>');`
**B.** `$('h1').append('<span>(New)</span>');`
**C.** `$('p').remove();`

–>

# jQuery Events and Animations: Introduction

> **Goals**
>
> By the end of this lesson you will:
> - Understand how to handle events in jQuery
> - Have explored some of the various effects and animations available in jQuery, such as fading, sliding, and toggling
> ### Introduction
> In addition to DOM manipulation, jQuery also makes handling events and animations easier! In this lesson, we will explore how to work with events, event listeners, and animations in jQuery, including many common events and animations found in a multitude of websites, and some of which we have already looked at previously in this course. And, of course, we'll have a few exercises to practice the concepts we'll learn.

# Events and Listeners part 1

> You've already seen how to handle events in native JavaScript, which is typically accomplished in two steps: first, you select a given element from the DOM and store it in a variable; then, you use that variable to attach an event listener to the DOM element that variable is referencing. While theoretically you could do both of those things in a single step in native JavaScript, that particular line of code would get a bit verbose and would make your code harder to read.
>
> In jQuery, you can combine the two steps of selecting an element and attaching an event listener to it, and you can still have clean, easy-to-read code! Let's have a quick look at the syntax for this:

```
$('SELECTOR').on('EVENT TYPE', function() {
  /* DO SOMETHING */
})
```

> In jQuery, once you select an element using its already concise syntax, attaching an event listener is as easy as chaining the **on()** method. Like in vanilla JavaScript's native **addEventListener** method, the **on()** method in jQuery takes two arguments – a string defining the event type, and a callback function to describe what should happen when that event is fired.
>
> Let's see some examples of the common events we can work with in JavaScript, and see how to handle them with jQuery.
>
> ### click

```
$('.myButton').on('click', function() {
  alert('Button clicked!')
})
```

In this code, **$('.myButton')** selects all `<button>` elements on the page with a class of "myButton", and **.on('click', ...)** attaches a click event listener to each button. When a button is clicked, the function inside **.on()** is called, which in this case will show an alert dialog.
>
You can also use jQuery's **.off()** method to remove event listeners. For example, the following code removes the click event listener from our button:
>

```
$('.myButton').off('click')
```

### mouseover

```
$('.myDiv').on('mouseover', function() {
  // 'this' is the element that triggered the mouseover event
  $(this).css('background-color', 'yellow')
})
```

In this code, **$('.myDiv')** selects the div element with the ID of "myDiv", and **.on('mouseover', ...)** attaches a mouseover event listener to that div. When the mouse pointer moves over the div, the function inside **.on()** will run, changing the div's background color to yellow.
>

### keydown

```
$(document).on('keydown', function(event) {
  if (event.key === 'Backspace') {
    console.log('Backspace key was pressed')
  } else {
    console.log(event.key)
  }
})
```

> NOTE: For this example, click the button in the preview panel to open the preview in a new browser tab. Then, open the console to see the output!

In above code, **$(document)** selects the entire document, and **.on('keydown', ...)** attaches a keydown event listener to the document itself. When a key is pressed down while the document has focus, the function inside **.on()** is called, which checks if the key pressed is the backspace key using the **event.key** property. If the key pressed *is* the backspace key, the code inside the if statement is executed. If the key pressed is any other key, the code inside the else statement is executed.
>

# Events and Listeners part 2

### focus and blur

> NOTE: For the following few examples, click the button in the preview panel to open the preview in a new browser tab. Then, open the console to see the output!

```
$('input[type="text"]').on('focus', function() {
  console.log('Input was focused')
})


$('input[type="text"]').on('blur', function() {
  console.log('Input was blurred')
})
```

In this code, we're selecting any input with the type attribute set to "text" and attaching both a **focus** event listener to run some code when the input receives focus (i.e. - the user clicks inside the input) and a **blur** event listener to run other code when the input is blurred (i.e. - the user clicks outside the input). In this simple example, we're just logging to the console in both cases.
>
### input

```
$('input[type="text"]').on('input', function(e) {
  console.log('Input event: ', e.target.value)
})
```

In this code, **$('input[type="text"]')** selects all our same text input from before, and now attaches an **input** event listener to it with **.on('input', ...)**. Whenever we type anything in the input now, its value will get logged to the console.
>
### change

```
$('input[type="text"]').off('input')

$('input[type="text"]').on('change', function(e) {
  console.log('Change event: ', e.target.value)
})
```

In this code, we first remove the previous input listener, and now attach a change listener to our same input. If you remember, the change event differs from the input event on a text input in that the change event will only fire when the input loses focus.
>

### submit

```
$('form').on('submit', function(event) {
  event.preventDefault();
  // Do something when the form is submitted
  console.log('Submit event target: ', event.target.value)
})
```

In this code, **$('form')** selects the form element on the page, and **.on('submit', ...)** attaches a submit event listener to the form. Inside the callback function, we execute the **preventDefault()** method to stop the page from reloading so we can actually do something with the form data. In this case, we're just logging to the console, but in the wild calling **preventDefault()** on the submit event is usually best practice.
>
There are, of course, many more events that we can handle with jQuery – basically, any event that can be handled in native JavaScript can also be handled in jQuery. The above events were included here as a sampling to get you used to jQuery's syntax.

# Working with Animations part 1

> jQuery also comes with a variety of functions for working with effects and animations:
> - **.fadeIn()** - Transition the selected element to fully opaque (opacity: 1)
> - **.fadeOut()** - Transition the selected element to fully transparent (opacity: 0)
> - **.fadeToggle()** - Fade the selected element in or out depending on its current opacity
> - **.show()** - Reveals the selected element by changing its display property
> - **.hide()** - Hides the selected element by changing its display property
> - **.slideUp()** - Hides the selected element with a slide up motion by animating the height
> - **.slideDown()** - Shows the selected element with a slide down motion by animating the height
> - **.slideToggle()** - Shows or hides the selected element with a slide up or slide down motion depending on its current state
> - **.animate()** - Define custom animations on numeric CSS properties
>
> Let's take a look at some examples of these different animations.
> ### .fadeIn()

```
$('#fade-in-button').on('click', function() {
  $('#fade-in-element').fadeIn(1000)
})
```

> **NOTE:** With this animation, the element being animated needs to have its **display** property initially set to "none"!

> In this code, when the button is clicked **$('#fade-in-element')** selects the element with the ID "fade-in-element", and **.fadeIn(1000)** gradually increases its opacity over a period of 1000 milliseconds (1 second).

### .fadeOut()

```
$('#fade-out-button').on('click', function() {
  $('#fade-out-element').fadeOut(1000)
})
```

> In this code, when the button is clicked **$('#fade-out-element')** selects the element with the ID "fade-out-element", and **.fadeOut(1000)** gradually decreases its opacity over a period of 1000 milliseconds (1 second).

### .fadeToggle()

```
$('#fade-toggle-button').on('click', function() {
  $('#fade-toggle-element').fadeToggle(1000)
})
```

In this code, when the button is clicked the **.fadeToggle()** method is called on an element with the ID "fade-toggle-element", causing it to fade in or out over a period of 1000 milliseconds (1 second).
>

### .show()

```
$('#show-button').on('click', function() {
  $('#show-hide-element').show(1000)
})
```

> In this code, when the **Show** button is clicked **$('#show-element')** selects the element with the ID "show-element", and **.show(1000)** gradually increases its opacity and size over a period of 1000 milliseconds (1 second).

### .hide()

```
$('#hide-button').on('click', function() {
  $('#show-hide-element').hide(1000)
})
```

> In this code, when the **Hide** button is clicked **$('#hide-element')** selects the element with the ID "hide-element", and **.hide(1000)** gradually decreases its opacity and size over a period of 1000 milliseconds (1 second).

# Working with Animations part 2

### .slideUp()

```
$('#slide-up-button').on('click', function() {
  $('#slide-up-element').slideUp(1000)
})
```

> In this code, when the button is clicked **$('#slide-up-element')** selects the element with the ID "slide-up-element", and **.slideUp(1000)** slides it up and fades it out over a period of 1000 milliseconds (1 second).

### .slideDown()

```
$('#slide-down-button').on('click', function() {
  $('#slide-down-element').slideDown(1000)
})
```

> **NOTE:** With this animation, the element being animated needs to have its **display** property initially set to "none"!

> In this code, when the button is clicked **$('#slide-down-element')** selects the element with the ID "slide-down-element", and **.slideDown(1000)** slides it down and fades it in over a period of 1000 milliseconds (1 second).

### .slideToggle()

```
$('#slide-toggle-button').on('click', function() {
  $('#slide-toggle-element').slideToggle(1000)
})
```

> In this code, when the button is clicked the **.slideToggle()** method is called on an element with the ID "slide-toggle-element", causing it to slide up and fade out or down and fade in over a period of 1000 milliseconds (1 second).

> ### .animate()

```
$('#animate-button').on('click', function() {
  $('#animate-element').animate({
    width: '200px',
    height: '200px',
    opacity: 0.5,
  }, 1000)
})
```

In this code, when the button is clicked **$('#animate-element')** selects the element with the ID "animate-element", and **.animate({...}, 1000)** changes its width, height, and opacity properties over a period of 1000 milliseconds (1 second).

# Conclusion & Takeaways

> In this lesson, we learned how to use jQuery to handle a variety of different events. We covered the click, mouseover, keydown, focus, blur, input, change, and submit events, as well as many of jQuery's built-in animations. Having already learned how to work with these concepts in native JavaScript, we've now seen how jQuery can make our lives a bit easier when writing code to achieve the same functionality. And, in essence, that is the purpose of using external libraries – leveraging what others have built so that we don't have to reinvent the wheel. Next, we'll cover another helpful and very popular JavaScript library: Lodash!

## Resources

### Beginner

W3Schools
- W3Schools provides a beginner-friendly jQuery tutorial with clear explanations, examples, and an interactive editor to try out code snippets.

SoloLearn
- SoloLearn offers a jQuery tutorial as part of their web development curriculum, including quizzes and coding exercises for beginners.

### Intermediate

jQuery Official Website
- The official jQuery website offers a comprehensive learning center that includes tutorials, examples, and API documentation for intermediate learners.

FreeCodeCamp
- FreeCodeCamp offers a jQuery section as part of their Front End Libraries certification, which includes challenges and projects for intermediate learners.

# Practice Time - Coding Exercises

> Work alone or with a partner to answer the following practice questions. Try to come up with solutions first, but If you get stuck you can always refer back to the material in this lesson or by asking an instructor or TA for help. Happy coding!
>
> 1. Create a button that changes the background color of the page when clicked.
> 2. Create a form that displays a message when it is submitted.
> 3. Create a div that slides up when clicked.
> 4. Create a button that fades in an image when clicked. For the image you can use Unsplash, Picsum, or something along those lines.
> 5. Create a button that animates a div when clicked.

# Introduction to Lodash

## Goals

By the end of this lesson you will:
- Have learned what Lodash is and why we might want to use it
- Understand how to begin working with Lodash

## Introduction

Lodash is a JavaScript library that provides utility functions for common programming tasks. It is designed to be modular, so you can use only the parts of the library that you need. Lodash makes it easy to manipulate arrays, objects, strings, and other data types in a functional style. It is compatible with all modern web browsers and can also be used in server side JavaScript using Node.js. In this lesson, we will cover a brief overview of its syntax and practice using it.

# Getting Started

> Using Lodash is fairly easy. We will use the CDN option like we did for jQuery:

```
<script
        src="https://cdn.jsdelivr.net/npm/lodash@4.17.21/lodash.min.j
        s"></script>
```

Copy the above script and paste it into the HTML file just above our main script:

```
<!-- include Lodash here -->
<script src='script.js'></script>
```

We can now begin to use Lodash in our JavaScript!
>

# Syntax Overview

> Lodash provides a plethora of utility functions that we can use to manipulate arrays, objects, strings, and other data types. The syntax for using Lodash functions is similar to the syntax for using built-in JavaScript functions. The _ symbol represents the Lodash library.
>
> Here's an example of using Lodash to manipulate an array:

```
const numbers = [1, 2, 3, 4, 5];

const sum = _.sum(numbers); // Output: 15

console.log(sum)
```

In this example, we're using the sum function from Lodash to calculate the sum of the numbers in an array. We pass the array as an argument to the sum function, and it returns the sum of the array elements. We then log the sum to the console using console.log. In the IDE, you can see that example already set up.
>
Here's another example, this time using Lodash to manipulate an object:

```
const person = {
  name: 'John',
  age: 30,
  address: {
    street: '123 Main St',
    city: 'Anytown',
    state: 'CA'
  }
};

const addressString = _.join(_.values(person.address), ', '); //
        Output: "123 Main St, Anytown, CA"

console.log(addressString)
```

In this example, we're using the **join** and **values** functions from Lodash to concatenate the values of an object's properties into a string. We pass the object's address property to the **values** function, which returns an array of the property values. We then pass this array to the **join** function, which concatenates the values into a string using a separator of `, `.
>

Try putting each example into the JavaScript file with a console log statement to see the output!

>

Lodash provides many other useful functions for manipulating data. Here are some examples of common tasks you can perform with Lodash:

>

- Find the minimum or maximum value in an array: `_.min` and `_.max`
- Sort an array: `_.sortBy`
- Filter an array based on a condition: `_.filter`
- Map an array to a new array with transformed values: `_.map`
- Reduce an array to a single value: `_.reduce`
- Merge two or more objects into a single object: `_.merge`
- Clone an object or array: `_.cloneDeep`

>

You can find a complete list of Lodash functions in the Lodash documentation.

# Conclusion & Takeaways

> Lodash is a powerful utility library for JavaScript that can help you write cleaner and more efficient code. It provides a set of functions for manipulating arrays, objects, strings, and other data types in a functional style. The purpose of libraries is to save you time and help you compose functional code. In this lesson we looked at how to get up and use Lodash, then went over some basics of the Lodash syntax. In the upcoming lessons, we'll be covering several useful Lodash methods in detail.

# Lodash: Deep Dive - Part 1

> ### Goals
>
> By the end of this lesson you will:
> - Be familiar with many of Lodash's array and object utility methods
>
> ### Introduction
>
> Lodash boasts an extensive set of array and object methods, which can greatly simplify working with these two data structures in JavaScript. In this lesson, we'll cover some of the many and very useful array and object methods which Lodash provides, along with code examples.

# Array Methods part 1

Let's start by going over several helpful array methods in Lodash.
>
### _.compact()
The **_.compact()** method allows us to remove any falsey values from an array, such as null, undefined, 0, and false. This can be useful when we need to clean up data before processing it. Here's an example:

```
const array1 = [0, 1, false, 2, '', 3]
const compactedArray = _.compact(array1)

console.log(compactedArray)

// Output: [1, 2, 3]
```

In this example, we've removed the falsey values from the array using the *.compact() method. The resulting compactedArray contains only the truthy values.*
>
### .flattenDeep()
The **_.flattenDeep()** method allows us to recursively flatten all nested arrays into a single array. This can be useful when we need to combine data from multiple nested arrays. Here's an example:

```
const array2 = [1, [2, 3], [4, [5, 6]]]
const flattenedArray = _.flattenDeep(array2)

console.log(flattenedArray)

// Output [1, 2, 3, 4, 5, 6]
```

In this example, we've flattened the nested array into a single array using the **_.flattenDeep()** method. The resulting **flattenedArray** contains all the elements from the original array, now as a new one-dimensional array.
>

### _.difference()

The **_.difference()** method allows us to create an array of values that are present in the first array but not in the second array. This can be useful when we need to compare two arrays and find the differences between them. Here's an example:

```
const array3 = [1, 2, 3, 4, 5]
const array4 = [2, 4, 6]
const differenceArray = _.difference(array3, array4)

console.log(differenceArray)

// Output: [1, 3, 5]
```

In this example, we've created a new array that contains the elements in **array3** that are not present in **array4**. The resulting **differenceArray** contains the values 1, 3, and 5.
>

## _.intersection()

The **_.intersection()** method allows us to create an array of values that are present in all of the input arrays. This can be useful when we need to compare multiple arrays and find the common elements between them. Here's an example:

```
const array5 = [1, 2, 3, 4, 5]
const array6 = [2, 4, 6]
const array7 = [3, 4, 5, 7]

const intersectionArray = _.intersection(array5, array6, array7)

console.log(intersectionArray)

// Output: [4]
```

In this example, we've created a new array containing only the elements that are present in all three input arrays. The resulting **intersectionArray** contains only the value 4.
>

## _.reverse()

The **_.reverse()** method allows us to reverse the order of elements in an array. This can be useful when we need to reverse the order of data for display or processing. Here's an example:

```
const array8 = [1, 2, 3, 4, 5]
const reversedArray = _.reverse(array8)

console.log(reversedArray)

// Output: [5, 4, 3, 2, 1]
```

In this example, we've reversed the order of elements in array using the **_.reverse()** method. The resulting **reversedArray** contains the same elements as the original array, but in reverse order.

# Array Methods part 2

> ### _.sortBy()
>
> The **_.sortBy()** method allows us to sort an array by a specified property or function. This can be useful when we need to order data in a specific way. Here's an example:

```
const array9 = [
  { name: 'Alice', age: 30 },
  { name: 'Bob', age: 25 },
  { name: 'Charlie', age: 35 },
]

const sortedArray = _.sortBy(array9, ['age'])

console.log(sortedArray)

/* Output:
  [
   { name: 'Bob', age: 25 },
   { name: 'Alice', age: 30 },
   { name: 'Charlie', age: 35 }
  ]
*/
```

In this example, we've used ***.sortBy()** to sort the array by the age property. The resulting sortedArray contains the same objects as the original array, but in ascending order by age.*
>
**### .sortedIndex()**
**The** _.sortedIndex()** method allows us to find the index at which a value should be inserted into a sorted array in order to maintain the sorted order. This can be useful when we need to insert new values into an already sorted array. Here's an example:

```
const array10 = [10, 20, 30, 40]
const index = _.sortedIndex(array10, 25)

array10.splice(index, 0, 25)

console.log(`Sorted index: ${index}`) // Output: Sorted index: 2
console.log(`Array with new value at sorted index: ${array10}`)
```

In this example, we've used ***.sortedIndex()** to find the index at which the value 25 should be inserted into the sorted array. The resulting index is 2, indicating that 25 should be inserted between the values 20 and 30 in the array.*

>

### **.uniq()**

**The** _.uniq()** method allows us to create a new array with all duplicate values removed. This can be useful when we need to work with a unique set of data. Here's an example:

```
const array11 = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
const uniqueArray = _.uniq(array11)

console.log(uniqueArray)

// Output: [1, 2, 3, 4]
```

In this example, we've used ***.uniq() to create a new array that contains only the unique values from the original array. The resulting uniqueArray** contains the values 1, 2, 3, and 4.*

>

### **.uniqBy()**

**The** _.uniqBy()** method is similar to **_.uniq()**, but it allows us to specify a function that determines how to compare elements for uniqueness. This can be useful when we need to compare elements based on a specific property or attribute. Here's an example:

```
const array12 = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
  { id: 3, name: 'Alice' },
  { id: 4, name: 'Charlie' },
]

const uniqueByArray = _.uniqBy(array12, 'name')

console.log(uniqueByArray)

/* Output:
  [
    { id: 1, name: 'Alice' },
    { id: 2, name: 'Bob' },
    { id: 4, name: 'Charlie' }
  ]
*/
```

In this example, we've used **_.uniqBy()** to create a new array that contains only the unique objects from the original array based on the **name** property. The resulting **uniqueArray** contains only one object for each unique name value: { id: 1, name: 'Alice' }, { id: 2, name: 'Bob' }, and { id: 4, name: 'Charlie' }.

# Object Methods

> Lodash also provides several useful methods for working with objects. Let's have a look at some of them.
>
> ### _.pickBy()
>
> The **_.pickBy()** method is used to construct a new object from an existing one, including only the properties that satisfy a provided predicate function. This can be useful when you need to filter an object based on certain criteria, such as creating a new user profile object that only includes properties with non-null values.
>
> Here's an example of how to use the **_.pickBy()** method:

```
let userProfile = {
  name: 'John',
  age: 30,
  email: 'john@example.com',
  address: null,
  phone: null
}

let filteredProfile = _.pickBy(userProfile, (value) => value !==
        null)

console.log(filteredProfile)

/*
  Output:
  {
    name: 'John',
    age: 30,
    email: 'john@example.com'
  }
*/
```

### _.omitBy()

The **_.omitBy()** method is used to create a new object from an existing one, excluding the properties that satisfy a provided predicate function. This can be useful when you need to remove certain properties from an object based on specific conditions, such as creating a simplified version of a complex object by omitting properties that are not needed for a

particular operation or function.
>
Here's an example of how to use the **_.omitBy()** method:
>

```
let complexProfile = {
  name: 'John',
  age: 30,
  email: 'john@example.com',
  address: '123 Main St',
  phone: '123-456-7890',
  internalCode: 'XYZ123',
  debugInfo: 'Some debug info'
}

let simplifiedProfile = _.omitBy(complexProfile, (value, key) =>
        key === 'internalCode' || key === 'debugInfo')

console.log(simplifiedProfile)

/*
  Output:
  {
    name: 'John',
    age: 30,
    email: 'john@example.com',
    address: '123 Main St',
    phone: '123-456-7890'
  }
*/
```

## _.merge()

The **_.merge()** method is used to merge two or more objects together, with the subsequent objects' properties overwriting the preceding ones if they have the same key. This can be useful when you need to combine multiple objects into a single one while allowing for flexible modification as needed.
>
Here's a simple example of how to use **_.merge** to combine two objects:

```
const person = {
  name: 'John',
  age: 30
}

const address = {
  street: '123 Main St',
  city: 'New York',
  state: 'NY'
}

const mergedObject = _.merge(person, address)

console.log(mergedObject)

/* Output:
{
  name: 'John',
  age: 30,
  street: '123 Main St',
  city: 'New York',
  state: 'NY'
}
*/
```

# Conclusion & Takeaways

> Lodash provides a wide range of collection methods for working with arrays and objects, along with many other useful utility functions – more of which we'll see in the upcoming lesson. By mastering these methods, you can work with these data structures more efficiently and write code that is often easier to read and maintain.

# Practice Time - Coding Exercises

> Work alone or with a partner to answer the following practice questions. Try to come up with solutions first, but If you get stuck you can always refer back to the material in this lesson or by asking an instructor or TA for help. Happy coding!
>
> 1. Write a function that takes this array `[1, 1, 2, 3, 9, 6, 11, 11, 11, 5]` as an argument and returns a new array with all the duplicate values removed.
> 2. Use a Lodash utility method to transform the values of a map and log the resulting map to the console. Use the same array as in 1.
> 3. Use _.difference() to find the unique elements in three or more sets and log the resulting set to the console. Use these sets:

```
[1, 2, 7, 3, 5]
[19, 6, 8, 3]
[7, 7, 8, 9, 45, 21]
```

> 4. Use _.merge to merge two or more objects and log the resulting object to the console.

# Lodash: Deep Dive - Part 2

> ### Goals
>
> By the end of this lesson you will:
> - Be familiar with more utility methods from Lodash, including **string methods** and an assortment of other helpful methods
> - Have even more tools in your JavaScript toolbox to help speed up development and easily solve many common problems that may arise
>
> ### Introduction
>
> In this lesson, we will cover another batch of utility methods that Lodash makes available to us, including methods to perform various operations on strings, and miscellaneous other helpful methods such as debounce, partition, and random. By the end of this lesson, you will have a solid understanding of how to use these powerful helper functions to solve many common programming problems and, as always, improve the speed and efficiency of the development process.

# String Methods part 1

> In addition to methods for arrays and objects, Lodash also includes several methods for working with strings. We'll go over a few of them here.
>
> ### _.capitalize()
>
> The **_.capitalize()** method capitalizes the first character of a string. This is useful when you want to format user input or display text in a certain way.
>
> Here's an example of how you might use the **_.capitalize()** method:

```
const string1 = 'hello world'
const capitalizedString = _.capitalize(string1) // Output:
        "Hello world"

console.log(`_.capitalize() result: "${capitalizedString}"`)
```

### _.truncate()

The **_.truncate()** method truncates a string to a specified length and adds an ellipsis (...) to the end. This is useful when you want to limit the amount of text displayed to the user.
>
Here's an example of how you might use the **_.truncate()** method:

```
const string2 = 'The quick brown fox jumps over the lazy dog.'
const truncatedString = _.truncate(string2, { length: 22 }) //
        Output: "The quick brown fox..."

console.log(`_.truncate() result: "${truncatedString}"`)
```

> **NOTE**: The length specified in the second argument of the **_.truncate()** method will include spaces and the ellipsis!

### _.camelCase()

The **_.camelCase()** method converts a string to camel case, which is lowercase with the first letter of each word capitalized and no spaces or punctuation. This is useful when you want to convert user input or

database field names to a standard format.
>
Here's an example of how you might use the **_.camelCase()** method:

```
const string5 = 'The quick brown fox jumps over the lazy dog.'
const camelCaseString = _.camelCase(string5) // Output:
        "theQuickBrownFoxJumpsOverTheLazyDog"


console.log(`_.camelCase() result: "${camelCaseString}"`)
```

## _.escape()

The **_.escape()** method escapes special characters in a string, such as HTML tags and entities, to help prevent cross-site scripting (XSS) attacks. This is useful when you want to display user-generated content on a web page.
>
Here's an example of how you might use the **_.escape()** method:

```
const string7 = '<script>alert("Hello, world!");</script>'
const escapedString = _.escape(string7)
// Output: "&lt;script&gt;alert(&quot;Hello,
        world!&quot;);&lt;/script&gt;"


console.log(`_.escape() result: "${escapedString}"`)
```

# String Methods part 2

## _.pad()

The **_.pad()** method adds padding to the beginning and/or end of a string to make it a specified length. This is useful when you want to format text in a certain way, such as aligning columns in a table.
>
Here's an example of how to use the **_.pad()** method:

```
const string8 = "123"
const paddedString = _.pad(string8, 5, '0') // Output: "01230"

console.log(`_.pad() result: "${paddedString}"`)
```

## _.words()

The **_.words()** method splits a string into an array of words. This is useful when you want to count the number of words in a string or perform operations on individual words.
>
Here's an example of how to use the **_.words()** method:

```
const string10 = 'The quick brown fox jumps over the lazy dog.'
const wordsArray = _.words(string10)

console.log(`_.words() result: ${JSON.stringify(wordsArray)}`)

// Output ["The", "quick", "brown", "fox", "jumps", "over",
        "the", "lazy", "dog."]
```

## _.template()

The **_.template()** method compiles a string template into a function that can be used to generate dynamic HTML or other types of text. This is useful when you want to generate dynamic content or reuse the same HTML structure in multiple places.
>
The syntax for including placeholders where dynamic values will be injected is similar to that of the EJS templating engine.

Here's an example of how to use the **_.template()** method:

```
const template = _.template('<div><%= name %> is <%= age %>
        years old.</div>')
const compiledTemplate = template({ name: 'John', age: 30 })

console.log(`_.template() result: "${compiledTemplate}"`)

// Output: "<div>John is 30 years old.</div>"
```

### _.replace()

Like the string replace method in native JavaScript, the Lodash replace method replaces occurrences of a string with another string. The difference is the Lodash version replaces all occurrences.
>
Here's an example of how to use the **_.replace()** method:

```
const string11 = 'The quick brown fox jumps over the lazy dog.'
const replacedString = _.replace(string11, 'quick', 'slow')

console.log(`_.replace() result: "${replacedString}"`)

// Output: "The slow brown fox jumps over the lazy dog."
```

# Other Useful Methods part 1

> ### _.partition()
>
> The **_.partition()** method creates an array of two arrays, one for the elements that satisfy a given condition and one for the elements that do not satisfy the condition.
>
> Here's an example of how to use the **_.partition()** method:

```
const users1 = [
  { name: 'John', age: 21 },
  { name: 'Jane', age: 17 },
  { name: 'Robert', age: 19 }
]

const [adults, minors] = _.partition(users1, (user) => user.age
        >= 18)

console.log(`_.partition() result (adults):
        ${JSON.stringify(adults)}`)
console.log(`_.partition() result (minors):
        ${JSON.stringify(minors)}`)

// Output:
//  [{ name: 'John', age: 21 }, { name: 'Robert', age: 19 }]
//  [{ name: 'Jane', age: 17 }]
```

### _.debounce()

The **_.debounce()** method creates a new function that will delay its execution until after a specified amount of time has passed since the last time it was called.
>
Here's an example of how you might use the **_.debounce()** method:

```
function onResize() {
  console.log('resizing...')
}

const debounced = _.debounce(onResize, 1000)
window.addEventListener('resize', debounced)
```

> **NOTE**: For this example, open the preview in a new browser tab and then resize the window to see the output of the debounced function!

Due to being rate-limited by the **_.debounce()** method, the message logged to the console by the **onResize** function will only be executed at most every 1000 milliseconds (1 second), no matter how much the window is resized.
>

## _.orderBy()

The **_.orderBy()** method sorts an array of objects based on one or more properties, in either ascending or descending order.
>
Here's an example of how you might use the **_.orderBy()** method:

```
const users = [
  { name: 'John', age: 25 },
  { name: 'Jane', age: 22 },
  { name: 'Bob', age: 30 }
]

const sorted = _.orderBy(users, ['name', 'age'], ['asc',
        'desc'])


console.log(`_.orderBy() result: ${JSON.stringify(sorted)}`)

// Output: [{ name: 'Bob', age: 30 }, { name: 'Jane', age: 22 },
        { name: 'John', age: 25 }]
```

# Other Useful Methods part 2

> ### _.cloneDeep()
>
> The **_.cloneDeep()** method creates a deep clone of a value, meaning it creates a new object or array that has the same properties and/or methods as the original. Also, any nested objects or arrays are also cloned recursively, so that the resulting object or array is completely separate from the original.
>
> Here's an example of how you might use the **_.cloneDeep()** method:

```
const obj = { a: 1, b: { c: 2 } }
const clone = _.cloneDeep(obj) // Output: { a: 1, b: { c: 2 } }

// Checking equality between obj and clone would return false

console.log(`_.cloneDeep() result: ${JSON.stringify(clone)}`)
console.log(`_.cloneDeep() objects the same?: ${obj === clone}`)
```

### _.isEqual()

The isEqual method compares two values for equality. It recursively compares object properties and array elements.
>
Here's an example of how you might use the _.isEqual() method:

```
const obj1 = { a: 1, b: { c: 2 } }
const obj2 = { a: 1, b: { c: 2 } }


const isEqual = _.isEqual(obj1, obj2)

console.log(`_.isEqual() result: ${isEqual}`)

// Output: true
```

### _.attempt()

The attempt method invokes a function with the provided arguments and returns either the result or the caught error.
>
Here's an example of how you might use the _.attempt() method:

```
//  Write your code here
function divide(a, b) {
  if (b === 0) {
    throw new Error('division by zero')
  }
  return a / b
}

const result1 = _.attempt(divide, 4, 2) // Output: 2
const result2 = _.attempt(divide, 4, 0) // Output: ERROR:
        division by zero

console.log(`_.attempt() result1: ${result1}`)
console.log(`_.attempt() result2: ${result2}`)
```

# Conclusion & Takeaways

> In this lesson, we explored some additional useful Lodash string methods, including methods for escaping special characters, padding strings, trimming whitespace, splitting strings, and many more helpful methods. By incorporating these Lodash methods into your JavaScript code, you can write more powerful and flexible applications that can handle a wide range of user input and data.

## Resources

- The Lodash Library - Getting Started
- The Beginner's Guide to Lodash - This guide explains how to use Lodash and provides examples of how to use some of its functions.

# Practice Time - Coding Exercises

> Work alone or with a partner to answer the following practice questions. Try to come up with solutions first, but If you get stuck you can always refer back to the material in this lesson or by asking an instructor or TA for help. Happy coding!

1. Create an array of objects that are books, with properties such as title, author, and year. Use the groupBy method to create a new object where the keys are the decades based on the year property and the values are arrays of the books that were published in that decade.
   >
2. Create an array of animal objects, with properties such as species, color, and size. Then, use the reject method to create a new array from the animals that are not larger than a certain size, based on a size property.
   >
3. Create a function, add, that adds two numbers and returns the result. Use the attempt method to invoke the add function with two numbers and catch any errors that occur.
   >

# Introduction to D3

## Goals

By the end of this lesson you will:
- Understand what D3 is and why it's useful
- Know how to select DOM elements with D3 and append other elements to them
- Have learned the basic building blocks for creating a simple bar chart

## Introduction

D3.js is a popular JavaScript library for creating data-driven visualizations, including charts, maps, and more. In this lesson, we will cover the basics of D3, including how to get set up with the library, how to select and append DOM elements, apply styles and attributes to them, and also how to create your first bar chart visualization using an SVG element and data binding.

# Overview of D3.js

> D3.js (short for Data-Driven Documents) is a JavaScript library that is used for manipulating and visualizing data on the web. It was developed by Mike Bostock and his team at the New York Times in 2011, and has since grown into one of the most widely used and powerful data visualization tools available.
>
> D3.js is used by a wide range of organizations and developers for everything from simple data visualizations to complex interactive dashboards, and offers several benefits for developers looking to create dynamic and interactive data visualizations on the web. Some of the key benefits include:
> - **Powerful data binding**: D3.js allows you to bind data to HTML or SVG elements, making it easy to create dynamic and interactive visualizations that update in real-time.
>
> - **Customizability**: With D3.js, you have full control over every aspect of your visualization, from the layout to the colors to the animation effects.
>
> - **Flexibility**: D3.js can be used with a wide range of data sources, including CSV files, JSON objects, and even live data from APIs.
>
> - **Community support**: D3.js has a large and active community of developers, which means there are plenty of resources available to help you get started and troubleshoot any issues you encounter.### Getting Set Up
>
> ### Setting up D3
>
> To access the functionality of D3, we can do something similar to what we did for jQuery and Lodash. Add this line to your HTML file above the script tag linking your main JavaScript file:

```
<script
        src="https://cdn.jsdelivr.net/npm/d3@7.8.3/dist/d3.min.j
        s"></script>
```

And voila! You're ready to go.
>
### Create our First Element in D3
To get started with D3.js, let's create a simple HTML element and then use D3.js to manipulate it. Here's the code you'll need:

```
d3.select('body')
  .append('div')
  .attr('id', 'myDiv')


d3.select('#myDiv')
  .append('h1')
  .text('Hello, D3!')
```

Paste that code into the blank JavaScript file.
>
In this example, we're creating a new HTML element (a div with the ID "myDiv") and then using D3.js to select that element and append a new h1 element to it with the text "Hello, D3!".
>
Let's break down how this code works:
- The div element with the ID "myDiv" is appended to the body of our web page.
>
- The d3.select function is used to select the div element with the ID "myDiv".
>
- The append function is used to append a new h1 element to the selected div element.
>
- The text function is used to set the text of the new h1 element to "Hello, D3!".
>
When you load this web page in your browser, you should see a new div element with the text "Hello, D3.js!" inside it.

# A More Complex Example

> Let's create a simple bar chart using D3.js. Here are the necessary steps to achieve this:
> 1. Create an SVG element with a width of 500, a height of 300, and a solid blue border 2 pixels wide.

2. Create an array of data (e.g., [10, 20, 30, 40, 50]).
3. Use D3.js to create a rectangle for each data point, with a width of 50 and a height equal to the data value.
4. Position each rectangle so that they are evenly spaced along the x-axis.
5. Set the fill color of each rectangle to blue.
   >
   Let's see what the code for this looks like:

```
// STEP 1: Create an SVG element with a width of 500, a height
        of 300, and a solid blue border 2 pixels wide
const svg = d3.select('body')
  .append('svg')
  .style('border', '2px solid blue')
  .attr('width', 500)
  .attr('height', 300);


// STEP 2: Create an array of data
const data = [10, 20, 30, 40, 50];


// STEP 3: Create a rectangle for each data point
svg.selectAll('rect') // Select all existing rect elements
  // Bind the data array to the selection
  .data(data)
  // Create a placeholder for each data point that doesn't have
        a corresponding rect element yet
  .enter()
  // Append a rect element for each placeholder
  .append('rect')
  // Set the width of each rect element to 50
  .attr('width', 50)
  // Set the height of each rect element to the corresponding
        data value
  .attr('height', d => d)
  // STEP 4: Position each rectangle along the x-axis based on
        its index in the data array
  .attr('x', (d, i) => i * 60)
  // Invert the y-axis and set the y-position of each rect
        element based on its data value
  .attr('y', d => 300 - d)
  // STEP 5: Set the fill color of each rectangle to blue using
        the style method
  .style('fill', 'blue');
```

In this example, we're creating an SVG element with a width of 500 and a height of 300, as well as a 2 pixel solid blue border. We then create an array of data (10, 20, 30, 40, 50) and use D3.js to bind each data point to a rectangle element in the SVG.

>

The x position of each rectangle is determined by its index in the data array, and the y position is calculated so that the rectangles are anchored to the bottom of the SVG. The width of each rectangle is fixed at 50, and the height is set to the value of the corresponding data point. Finally, we set the fill color of each rectangle to blue.

>

When you load this web page in your browser, you should see a simple bar chart with five blue rectangles of varying heights.

# Conclusion & Takeaways

> D3.js is a powerful JavaScript library for creating data-driven visualizations on the web. It provides a wide range of tools for working with data and creating dynamic, interactive graphics. Getting started with D3.js is relatively easy, as it only requires a basic knowledge of HTML, CSS, and JavaScript. Once we have set up a new project using Vite and installed the D3 library using NPM, we created our first visualizations by selecting and appending elements, then creating an SVG element and binding data to it. D3 may seem a bit complex at first, but we'll look at plenty more examples throughout the coming two lessons and with some practice you will be well on your way to creating sophisticated and visually stunning visualizations that are driven by data.

# D3: Deep Dive - Part 1

> ### Goals
>
> By the end of this lesson you will:
> - Understand how to create and manipulate SVG elements using D3
> - Understand how to load and bind data to visual elements
> - Know how to create basic bar charts, line charts, and scatter plots
>
> ### Introduction
>
> In this lesson, we will continue diving into the basics of D3.js, including more on how to create and manipulate SVG elements, the different ways to load and bind data, and how to create a variety of visualizations (including bar charts, line charts, and scatter plots). By the end of this lesson, you will have the foundational knowledge you need to start creating your own data-driven visualizations with D3.js.

# SVG Elements

> The first step in creating a visualization with D3.js is to create an SVG element. SVG stands for Scalable Vector Graphics and is a format for describing two-dimensional graphics. To create an SVG element with D3.js, you can use the following code:

```
const svg = d3
    .select("#svg-example")
    .append("svg")
    .attr("width", 400)
    .attr("height", 200)
    .style("background-color","midnightblue")
```

This code creates an SVG element with a width of 400 pixels, a height of 200 pixels, and a background color of “midnightblue” and appends it to the element with an id of **svg-example**. You can, of course, customize the size of the SVG element to fit your specific needs; this is just an example.
>

# Bar Charts

> One of the most common types of visualizations is a bar chart. A bar chart is a chart that represents data with rectangular bars. Here's how we might create a bar chart with D3:

```
const barChartData = [17, 23, 39, 47, 55]

const barChartSVG = d3
    .select("#bar-chart")
    .append("svg")
    .attr("width", 400)
    .attr("height", 200)


barChartSVG.selectAll("rect")
   .data(barChartData)
   .enter()
   .append("rect")
   .attr("x", function(d, i) {
        return i * 50
   })
   .attr("y", function(d) {
        return 200 - d
   })
   .attr("width", 40)
   .attr("height", function(d) {
        return d
   })
   .attr("fill", "red")
```

This code creates a bar chart with five bars, each with a height corresponding to the value in the data array. The x-coordinate of each bar is determined by the index of the data in the array, and the y-coordinate is determined by subtracting the value of the data from the height of the SVG element. The width and color of the bars can be customized to fit your specific needs.
>
See for yourself! Replace the code in the `script.js` file with the bar chart example code above. You also have to create a place for it in your html document: you can do this by replacing the text in the `id` in your empty

 with “bar-chart” try this with all of the examples in this lesson to see D3 in action!
>

# Line Charts

> Another common type of visualization is a line chart. A line chart is a chart that represents data with a series of points connected by a line. To create a line chart with D3, we can do as follows:

```
const lineChartData = [
  {x: 0, y: 10},
  {x: 1, y: 20},
  {x: 2, y: 30},
  {x: 3, y: 40},
  {x: 4, y: 50}
]


const lineChartSVG = d3
    .select("#line-chart")
    .append("svg")
    .attr("width", 200)
    .attr("height", 200)


const line = d3.line()
           .x(function(d) { return d.x * 50 })
           .y(function(d) { return 200 - d.y });


lineChartSVG.append("path")
 .datum(lineChartData)
 .attr("fill", "none")
 .attr("stroke", "steelblue")
 .attr("stroke-width", 2)
 .attr("d", line)
```

This code creates a line chart with five points, each with an x-coordinate and a y-coordinate. The line connecting the points is created using the d3.line() function, which takes in functions to determine the x and y coordinates of each point. The x-coordinate of each point is determined by multiplying the x value by 50, and the y-coordinate is determined by subtracting the y value from the height of the SVG element. The line is then appended to the SVG element, and its fill color, stroke color, stroke width, and path data can be customized to fit your specific needs.
>

# Scatter Plots

> A scatter plot is a chart that represents data with a series of points. Each point represents a value for two different variables, and their position on the chart corresponds to their values for those variables. To create a scatter plot with D3.js, you can use the following code:

```
const scatterPlotData = [
  {x: 10, y: 20},
  {x: 20, y: 30},
  {x: 30, y: 40},
  {x: 40, y: 50},
  {x: 50, y: 60}
]


const scatterPlotSVG = d3
    .select("#scatter-plot")
    .append("svg")
    .attr("width", 300)
    .attr("height", 350)


scatterPlotSVG.selectAll("circle")
 .data(scatterPlotData)
 .enter()
 .append("circle")
 .attr("cx", function(d) {
      return d.x * 5
 })
 .attr("cy", function(d) {
      return 350 - d.y * 5
 })
 .attr("r", 5)
 .attr("fill", "orange")
```

This code creates a scatter plot with five points, each with an x-coordinate and a y-coordinate. The x-coordinate of each point is determined by multiplying the x value by 5, and the y-coordinate is determined by subtracting the y value from the height of the SVG element and multiplying it by 5. The radius and fill color of the points can be customized to fit your specific needs.
>

# Data Loading

> In D3, there are a few different ways to load data. You can manually create an array of data (like we did in one of the introductory examples in the previous lesson), or you can load data from other sources like CSV files, TSV files, and JSONs. You can even load data by making requests to 3rd party APIs (more on APIs and requesting data from them later in the course). For this lesson and following, we'll use CSV, TSV or JSON.
>
> ### Loading data
>
> D3.js allows you to load data from a variety of sources, including CSV files, JSON files, and TSV files. Be aware, however, that the following D3 methods we will utilize are all asynchronous– which, in JavaScript, means that they perform operations that will take an unknown amount of time to complete and thus will allow the rest of the code to continue executing.
>
> You will learn a lot more about asynchronous JavaScript a little later in the course. For now, suffice it to say that when an asynchronous method is called, it immediately returns a Promise object that represents the eventual completion (or failure) of the operation. Since this is the case, they have to be handled in JavaScript in a particular way, otherwise your code will error out.
>
> What we will do to handle each of these data loading methods is chain on two methods: **.then()** to handle the async code completing successfully, and **.catch()** to handle any errors (you'll learn more about these two methods in a later module).
>
> ### CSV Files
>
> To load data from a CSV file:

```
d3.csv('csvExample.csv')
    .then(data => console.log('CSV Data: ', data))
    .catch((error) => {
      console.log(error)
    })
```

This code loads the data from a CSV file called "data.csv" and logs it to the console. The data is loaded asynchronously, so the code inside the callback function will not execute until the data has finished loading.

>
### JSON Files
To load data from a JSON file:

```
d3.json('jsonExample.json')
    .then(data => console.log('JSON Data: ', data))
    .catch((error) => {
      console.log(error)
    })
```

This code loads the data from a JSON file called “data.json” and logs it to the console. Like loading data from a CSV file, loading data from a JSON file is an asynchronous operation.
>
### TSV Files
To load data from a TSV file:

```
d3.tsv('tsvExample.tsv')
    .then(data => console.log('TSV Data: ', data))
    .catch((error) => {
      console.log(error)
    })
```

This code loads the data from a TSV file called “data.tsv” and logs it to the console. Like loading data from a CSV file and a JSON file, loading data from a TSV file is an asynchronous operation.
>

# Data Binding

## Binding data to visual elements

Once you have loaded your data, you can bind it to visual elements to create your visualization. You already saw this in the different chart examples, but these were using hard-coded arrays for the data. This time, we'll see how we can join data after loading from an external source (whether a file or API).
>
#### Data joins
>
Data joins are used to associate each piece of data with a visual element. To perform a data join after loading from an external source, we once again should wrap everything in a **try**/**catch** statement. Then, let's just copy the example above where we loaded data from the CSV file:

```
d3.csv('csvExample.csv')
    .then((data) => {
      const dataJoinsSVG = d3
        .select("#data-joins")
        .append("svg")
        .attr("width", 200)
        .attr("height", 200)


      dataJoinsSVG
        .selectAll("rect")
        .data(data)
        .enter()
        .append("rect")
        .attr("x", function(d, i) {
            return i * 50
        })
        .attr("y", function(d) {
            return 200 - d.age
        })
        .attr("width", 40)
        .attr("height", function(d) {
            return d.age
        })
        .attr("fill", "purple")
    })
    .catch((error) => {
      console.log(error)
    })
```

This code performs a data join by selecting all of the “rect” elements in the SVG element and binding them to the data array. The enter() method is then used to create new elements for any data that is not already associated with a visual element. In this case, new “rect” elements are created for each element in the data array. The exit() method is used to remove any visual elements that are not associated with data. In this case, any “rect” elements that do not have data associated with them are removed.

# Conclusion & Takeaways

> In the preceding lesson, we delved deeper into working with SVG elements and creating a variety of basic visualizations with them. We covered bar charts, line charts, scatter plots and some of the different ways we can load and bind data to our SVG elements in order to create these graphics. While we've gone over a lot so far, there's much more to learn about D3. We won't be able to learn everything in this course, but in the following lesson we'll continue familiarizing ourselves with some of the most commonly utilized features and functionalities.

# D3: Deep Dive - Part 2

## Goals

By the end of this lesson you will:
- Understand how to use scales to map data to visual elements
- Know how to add axes and labels to visualizations
- Be familiar with the basics of animating visualizations, manipulating data, and using layouts
>
### Introduction
In this lesson, we will continue our deep dive into the D3 library and cover how to use scales, axes, and labels to make our visualizations informative and engaging. We will also explore how to add interactivity and animations to visualizations, and how to manipulate data using D3.js. Finally, we will introduce layouts and how they can be used to create more complex visualizations.

# Scales

> Scales in D3.js are functions that map data to visual elements, such as the position or size of the elements. D3.js supports several types of scales, including linear scales, logarithmic scales, and time scales. In this lesson, we'll just have a look at linear scales.
>
> ### Linear Scales
>
> Linear scales are used to map a continuous input domain to a continuous output range. This is useful for mapping data to visual elements such as the position of an element on a chart. Here is an example of creating a linear scale in D3:

```
// Select container element and append an SVG with a height and
        width
const linScalesElement = d3
  .select('#lin-scales')
  .append('svg')
  .attr('width', 500)
  .attr('height', 200)


// Define the input domain
const linScaleData = [10, 20, 30, 40, 50]

const xScaleLin = d3
  .scaleLinear()
  .domain([d3.min(linScaleData), d3.max(linScaleData)])
  .range([0, 400])


// Use the scale to set the position of a circle
linScalesElement
  .append("circle")
  .attr("cx", xScaleLin(50))
  .attr("cy", 50)
  .attr("r", 20)
```

In this example, we define an input domain of [10, 50] and an output range of [0, 400]. This means that any value in the input domain will be mapped to a value between 0 and 400.
>

# Axes

> Axes are used to display scales in a more human-readable format. D3 provides several built-in functions for creating axes, including `d3.axisBottom()`, `d3.axisTop()`, `d3.axisLeft()`, and `d3.axisRight()`.
>
> ### Creating Axes
>
> Here is an example of creating a bottom axis:

```
// Select container element and append an SVG with a height and
        width
const createAxesElement = d3
  .select('#create-axes')
  .append('svg')
  .attr('width', 500)
  .attr('height', 200)


// Create the scale
const axesDataCreate = [10, 20, 30, 40, 50]

const xScaleAxesCreate = d3.scaleLinear()
  .domain([d3.min(axesDataCreate), d3.max(axesDataCreate)])
  .range([0, 400])


// Create the axis
const xAxisCreate = d3.axisBottom(xScaleAxesCreate)


// Add the axis to the chart
createAxesElement
  .append("g")
  .attr("transform", "translate(0, 100)")
  .call(xAxisCreate)
```

In this example, we create a linear scale and an axis using the d3.axisBottom() function. We then add the axis to the chart by appending a g element and using the call() method to apply the axis to the g element.

# Customizing Axes

> Axes in D3 can be customized by modifying the tick values, labels, and other properties. Here is an example of customizing an axis:

```
// Select container element and append an SVG with a height and
        width
const customizeAxesElement = d3
  .select('#customize-axes')
  .append('svg')
  .attr('width', 500)
  .attr('height', 200)


// Create the scale
const customData = [10, 20, 30, 40, 50]

const xScaleCustom = d3.scaleLinear()
  .domain([d3.min(customData), d3.max(customData)])
  .range([0, 400])


// Create the axis
const xAxisCustom = d3.axisBottom(xScaleCustom)
  .tickValues([20, 40])
  .tickFormat(function(d) { return "$" + d })
  .tickSize(10)


// Add the axis to the chart
customizeAxesElement
  .append("g")
  .attr("transform", "translate(0, 50)")
  .call(xAxisCustom)
```

> In this example, we customize the tick values to [20, 40], add a prefix of $ to the tick labels, and set the tick size to 10.

# Labels

> Labels in D3.js are used to add text to visual elements, such as the axes or the chart title.

### Creating Labels

Here is an example of adding a label to the x-axis:

```
// Select container element and append an SVG with a height and
        width
const labelsElement = d3
  .select('#labels')
  .append('svg')
  .attr('width', 700)
  .attr('height', 200)


// Create the scale and axis
const labelsData = [10, 20, 30, 40, 50]


const xScaleLabels = d3
    .scaleLinear()
    .domain([d3.min(labelsData), d3.max(labelsData)])
    .range([0, 700])


const xAxisLabels = d3.axisBottom(xScaleLabels)


// Add the axis to the chart
labelsElement
  .append("g")
  .attr("transform", "translate(0, 50)")
  .call(xAxisLabels)
  .selectAll('text')
  .style('text-anchor', 'end')
  .attr('dx', '-.8em')
  .attr('dy', '.15em')


// Add the label to the x-axis
labelsElement
  .append("text")
  .attr("x", 300)
  .attr("y", 100)
  .text("X Axis Label")
```

> In this example, we add a label to the x-axis by appending a `text` element and setting the `x`, `y`, and `text` attributes.

### Customizing Labels

Labels in D3 can be customized by modifying the font size, color, and other properties. Add the following just below the label example we added a moment ago, to see an example of some simple label customizations:

```
// Customize Labels
  .style("font-size", "20px")
  .style("fill", "blue")
```

# Interactivity and Animations part 1

> Now let's have a look at some different things we can do in D3 in terms of interactivity.

### Mouse Events

Mouse events in D3 are used to add interactivity to visualizations, such as highlighting elements or showing tooltips. Here is an example of adding a mouseover event to a circle:

```
// Select container element and append an SVG with a height and
        width
const mouseChartElement = d3
  .select('#mouse-events')
  .append('svg')
  .attr('width', 500)
  .attr('height', 200)


// Create the circle
const mouseChartSVG = mouseChartElement
  .append("circle")
  .attr("cx", 100)
  .attr("cy", 50)
  .attr("r", 20)
  .style("fill", "red")


// Add a mouseover event to the circle
mouseChartSVG
  .on("mouseover", function() {
    d3.select(this)
      .style("fill", "blue")
  })
  .on("mouseout", function() {
    d3.select(this)
      .style("fill", "red")
  })
```

In this example, we add a mouseover event to a circle by using the on() method and passing in the event name and a function to execute when the event is triggered. We then use d3.select(this) to select the element that

triggered the event, and change its fill color.
>

# Interactivity and Animations part 2

## Animations

Animations in D3 are used to add visual interest and guide the viewer's attention to specific parts of the visualization.
>
#### Transitions
Transitions are used to animate changes to visual elements, such as position or size. Here is an example of animating a circle:

```
// Select container element and append an SVG with a height and
        width
const transitionChartElement = d3
  .select('#transition-easing')
  .append('svg')
  .attr('width', 500)
  .attr('height', 200)


// Create the circle
const transitionChartSVG = transitionChartElement
  .append("circle")
  .attr("cx", 100)
  .attr("cy", 50)
  .attr("r", 20)
  .style("fill", "red")


// Animate the circle
transitionChartSVG
  .transition()
  .duration(10000)
  .attr("cx", 300)
  .style("fill", "blue")
```

In this example, we animate a circle by using the transition() method and passing in a duration in milliseconds. We then use the attr() and style() methods to modify the position and fill color of the circle during the transition.
>
#### Easing Functions
Easing functions in D3.js are used to control the rate of change during a transition, such as speeding up or slowing down at specific points.

>

Here's an example of adding a bounce-out easing to the animated circle from above:

```
// Add easing
  .ease(d3.easeBounceOut)
```

In this example, we append the ease() method to the method chain in order to apply the `d3.easeBounceOut` easing function to the transition, which causes the circle to bounce out at the end of the animation.

>

# Data Manipulation part 1

> Data manipulation in D3 is used to filter and sort data to create more complex visualizations.
>
> ### Filtering Data
>
> Filtering data is used to remove unwanted data from a dataset before creating a visualization.
>
> Here is an example of filtering data:

```
// Select container element and append an SVG with a height and
        width
const filterChartElement = d3
  .select('#filtering')
  .append('svg')
  .attr('width', 500)
  .attr('height', 200)


// Filter the data based on a condition
const filterExampleData = [10, 20, 30, 40, 50];
const filteredData = filterExampleData.filter(function(d) {
        return d > 30; });

// Use the filtered data to create a visualization
filterChartElement
  .selectAll("rect")
  .data(filteredData)
  .enter()
  .append("rect")
  .attr("x", function(d, i) { return i * 50 })
  .attr("y", (d) => 200 - d)
  .attr("width", 40)
  .attr("height", function(d, i) { return d })
```

In this example, we filter the data array to only include values greater than 30. We then use the filtered data to create a visualization.
>

# Data Manipulation part 2

> ### Sorting Data
>
> Sorting data is used to reorder the elements in a dataset before creating a visualization.
>
> Here is an example of sorting data:

```
// Select container element and append an SVG with a height, a
        width, and a little extra top margin
const sortChartElement = d3
  .select('#sorting')
  .append('svg')
  .attr('width', 500)
  .attr('height', 200)
  .style('margin-top', "3rem")


// Sort the data in ascending order
const sortExampleData = [50, 10, 30, 40, 20]
const sortedData = sortExampleData.sort(function(a, b) { return
        a - b })


// Use the sorted data to create a visualization
sortChartElement
  .selectAll("rect")
  .data(sortedData)
  .enter()
  .append("rect")
  .attr("x", function(d, i) { return i * 50 })
  .attr("y", d => d)
  .attr("width", 40)
  .attr("height", (d) => 200 - d)
  .attr("fill", "#fff")
  .attr("stroke", "#000")
```

In this example, we sort the data array in ascending order using the sort() method and a comparison function. We then use the sorted data to create a bar chart by appending rect elements and setting their position and size based on the data.
>

# Layouts

> Layouts in D3 are used to transform data into a specific format for creating visualizations, such as a hierarchical tree or a pie chart which are created using functions (e.g. - d3.tree() or d3.pie()). In this lesson, we'll just have a look at pie charts.

### Creating a Pie Chart

Pie charts are created using the `d3.pie()` layout and the `d3.arc()` generator. Here is an example of creating a pie chart:

```
// Select container element and append an SVG with a height and
        width
const pieChartElement = d3
  .select('#pie-charts')
  .append('svg')
  .attr('width', 500)
  .attr('height', 200)


// Create the input data
const pieChartData = [
  { name: "A", value: 10 },
  { name: "B", value: 20 },
  { name: "C", value: 30 }
]


// Create a pie layout
const pie = d3.pie()
  .value(function(d) { return d.value })


// Transform the data using the pie layout
const pieData = pie(pieChartData)


// Use the transformed data to create a pie chart
pieChartElement
  .selectAll("path")
  .data(pieData)
  .enter()
  .append("path")
  .attr("d", d3.arc()
    .innerRadius(0)
    .outerRadius(50))
  .attr("transform", "translate(100, 100)")
  .style("fill", function(d) { return
        d3.schemeCategory10[d.index] })
```

In this example, we create a pie chart by using the d3.pie() function to create a pie layout, which transforms the input data into a format that can be used to create a pie chart. We then use the transformed data to create a pie chart by appending path elements and setting their position, size, and fill color based on the data.

>

# Conclusion & Takeaways

> In this lesson, we have covered several more of the key concepts and techniques for creating visualizations using D3.js. We started by going over the basics of scales, axes, and labels, and then moved on to more advanced topics such as interactivity and animations, data manipulation, and layouts. Along the way, we went through several code examples and practice exercises to help you develop your skills. Now, we're going to work on a couple mini projects to help solidify many of the concepts we learned this week.

# Mini Project: Introduction

> ### Goals
>
> By the end of this lesson you will:
> - Have gotten more practice in D3 by building a mini project
> - Have begun applying some of what you learned to add to your personal website
>
> ### Introduction
>
> In this last section for the week, we will use what we've learned about D3 and get some more practice creating a visualization (since D3 is the more difficult of three libraries we've looked at this week). Then, once that's done, you'll set to work thinking about how to apply any or all of the concepts you learned this week to your personal website.

# Population Bar Chart part 1

> ## Getting Set Up
>
> ### Modify HTML
>
> First let's go into **index.html** and enter a new title. You can write something like "Country Populations" or similar. Then, replace the div with the id "app" with the following:

```
<h1>Country Populations</h1>
  <h3>(in millions)</h3>
  <div class="container">
    <div id="chart"></div>
  </div>
```

Note: By now you've noticed that any changes to the html need to be in the document above the javascript that affects that html code.

### Add A Few Basic Styles

Now go into **style.css** and add the following:

```
@import url('https://fonts.googleapis.com/css2?
        family=Poppins:ital,wght@0,100;0,200;0,300;0,400;0,500;0,600;0,700;0,800;0,900;1,100;1,200;1,
        );



*, *::before, *::after {
  box-sizing: border-box;
  margin: 0;
  padding: 0;
}

body {
  font-family: "Poppins", sans-serif;
}

h1 {
  text-align: center;

}

h3 {
  text-align: center;
  margin-bottom: 5rem;
}

.container {
  max-width: 900px;
  margin: 0 auto;
}

#chart {
  display: flex;
  justify-content: space-around;
  align-items: flex-end;
  width: 100%;
  height: auto;
  padding: 20px;
  background-color: #f5f5f5;
}

.bar {
  width: 50px;
  margin: 0 5px;
}
```

**Don't forget to be sure the CSS file is linked to your HTML file!**

# Population Bar Chart part 2

> ### Write the JavaScript
>
> First, let's define the data we will use for our chart:

```
// Define the data to be used for the chart
const data = [
  { country: 'United States', population: 332 },
  { country: 'China', population: 1444 },
  { country: 'India', population: 1392 },
  { country: 'Indonesia', population: 276 },
  { country: 'Pakistan', population: 225 },
  { country: 'Brazil', population: 213 },
  { country: 'Nigeria', population: 211 },
  { country: 'Bangladesh', population: 166 },
  { country: 'Russia', population: 146 },
  { country: 'Mexico', population: 130 }
]
```

Now, we should set up some variables to help when defining dimensions and scales:

```
// Define variables for the chart dimensions and scales
const margin = { top: 20, right: 20, bottom: 60, left: 60 }
const width = 800 - margin.left - margin.right
const height = 500 - margin.top - margin.bottom
```

Having the margin variable ensures that we'll have a little extra breathing room on the edges of our visualization. Next, let's create scales for the two axes (band scale for **x**, and linear scale for **y**):
>

```
// Create a band scale for the x-axis
const x = d3.scaleBand()
  .domain(data.map(d => d.country))
  .range([0, width])
  .padding(0.2)


// Create a linear scale for the y-axis
const y = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.population)])
  .range([height, 0])
```

Okay, now we can define our outer SVG element and set it's dimensions using the **.attr()** method, and appending it to its respective container element with the ID "chart":

```
// Create an SVG element for the chart, define its width and
        height, and append a group element to it
const svg = d3.select('#chart')
  .append('svg')
  .attr('width', width + margin.left + margin.right)
  .attr('height', height + margin.top + margin.bottom + 20)
  .append('g')
  .attr('transform', `translate(${margin.left},${margin.top})`)
```

# Population Bar Chart part 3

Next we need to generate the rectangle elements which will be our bars, and we'll also go ahead and set them up so that when a mouse hovers over them their fill color lightens a little:

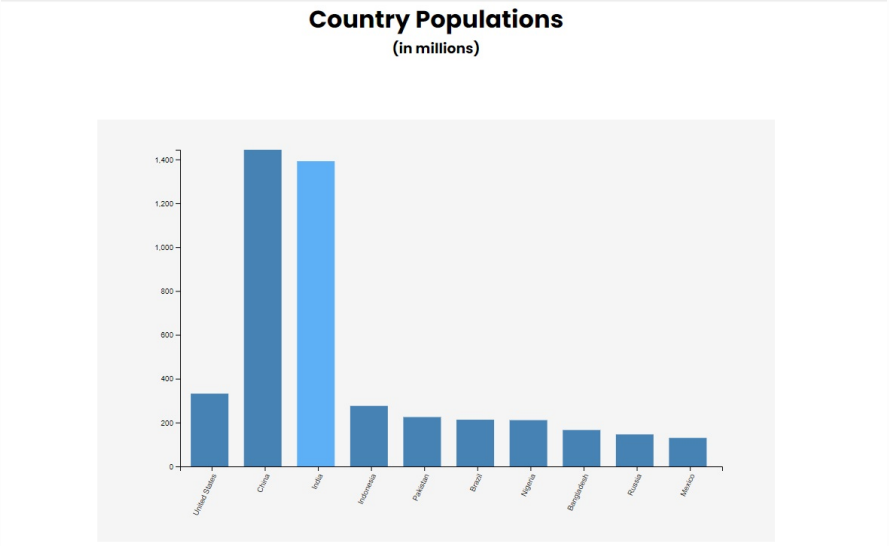
```
// Create rectangles for each data point and add interactivity
        with mouseover and mouseout events
svg.selectAll('.bar')
  .data(data)
  .enter()
  .append('rect')
  .attr('class', 'bar')
  .attr('x', d => x(d.country))
  .attr('width', x.bandwidth())
  .attr('y', d => y(d.population))
  .attr('height', d => height - y(d.population))
  .attr('fill', 'steelblue')
  .on("mouseover", function (d) {
    d3.select(this).style("fill", "#5db0f5")
  })
  .on("mouseout", function (d) {
    d3.select(this).style("fill", "steelblue")
  })
```

Last but not least, let's add our two labels to the chart and then open it in a browser preview and see how it looks!

```
// Add x-axis labels to the chart
svg.append('g')
  .attr('transform', `translate(0,${height})`)
  .call(d3.axisBottom(x))
  .selectAll('text')
  .style('text-anchor', 'end')
  .attr('dx', '-.8em')
  .attr('dy', '.15em')
  .attr('transform', 'rotate(-65)')


// Add y-axis labels to the chart
svg.append('g')
  .call(d3.axisLeft(y))
```

> If we did everything correctly, this is what it should look like in the browser preview:




>

# Conclusion & Takeaways

> We have covered a lot this week. After being introduced to setting up development projects with the Vite build tool, we did deep dives into jQuery, Lodash, and D3.js (which is no small feat!). Hopefully you've seen how these libraries can help you in various ways (depending on what your project is), by providing you with a bunch of extra baked-in functionality that you can then leverage to write more feature-rich frontend applications.

# Project: Add to personal website

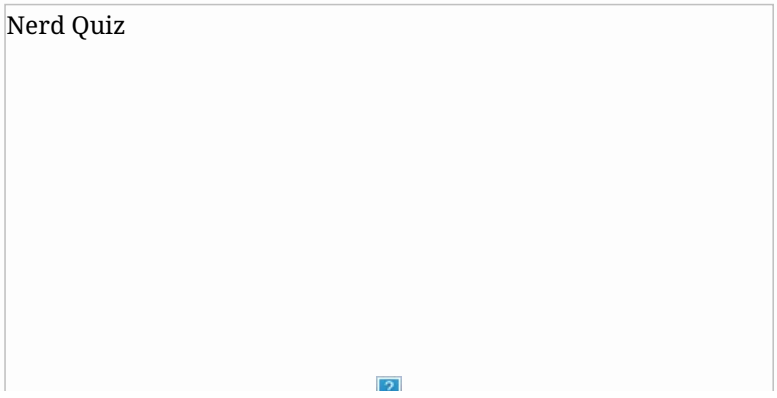
## Overview

For Project 12, you will get some practice creating a simple project from scratch that incorporates HTML, CSS, and JavaScript. Specifically, since jQuery was introduced this week, you'll write your JavaScript logic in jQuery! When completed, you'll add this project to GitHub and to your personal website's project section!

### Step One

Create a simple quiz app, which presents the user with at least 1 multiple choice question and gives some kind of feedback to the user about whether their selected answer was correct or incorrect.
>
**Requirements**:
>
- **HTML Structure** - The quiz should have a title, should provide 3 - 4 answer options, and should have an element to display feedback on the selected answer
>
- **CSS Styling** - The quiz should be centered on the page, applying suitable margin and padding for better readability and appearance. In addition, the answer options should employ some kind of hover effect (e.g. - changing color, etc.), and the feedback message should use distinct colors (one for a correct answer, another for an incorrect answer)
>
- **Functionality** - When an answer option is clicked, the other options should be disabled (unclickable), a determination should be made as to whether the answer is correct or not, and feedback should be output to the appropriate element. All the JS should be written in jQuery!

**Example**:



**Initial State**

Nerd Quiz - correct answer

**Correct Answer**

Nerd Quiz - incorrect answer

**Incorrect Answer**

> **NOTE**: This project does not need to be terribly complex or have amazing styling (unless you want to, of course). Some basic styling will be sufficient. Bottom line: don't overthink it, keep it simple, and prioritize functionality over aesthetics!

**Step Two**

Add all of your code to a GitHub repository, and add the quiz app to your website's project section. Include a screenshot, title, brief description, and a link to the repo.
>

When you're finished with both steps, submit the HTML, CSS, and jQuery for your quiz app below.
>
Happy coding!