**DOM traversal and manipulation**

**Objective:** By the end of this checkpoint, you can traverse and manipulate the DOM using jQuery.

To create an interactive web app, we need a way to programmatically alter the HTML that's displayed to users when certain events occur. For instance, we might want to display validation errors in a form or load additional articles to display as a user scrolls down in a result list.

In this checkpoint, we focus on using the JavaScript library [**jQuery**](https://api.jquery.com/) to alter the HTML displayed to end users. As we'll see, this is called **DOM (document object model) manipulation**.

At the end of this checkpoint, you’ll complete 1 short assignment:

1. Jot down at least 5 questions that you have about jQuery and interactive web apps so far. As you work through the rest of this module, see if you can find the answers to each of your questions. If you get to the end of the module and still have unanswered questions, be sure to share them with your mentor.

**Key Terms**

* DOM (document object model)
* jQuery
* Traversing
* Manipulating

**The DOM**

The [**DOM, or document object model**](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction) is the browser's representation of the current state of the HTML content of a page. The DOM is not to be confused with the HTML file for a page. When the browser first loads an HTML page, the DOM will look a lot like the HTML source code, but as JavaScript scripts begin to take over, HTML elements can be added and removed, regardless of whether or not they appeared in the HTML file.

You might not have realized it, but you worked with the DOM earlier in this curriculum. When you view HTML in the elements panel of Chrome DevTools, you are looking at a representation of the DOM.

**DOM Traversal and Manipulation**

We'll be using jQuery to find particular elements in the DOM (which is called **traversing**) and update them (which is called **manipulating**). DOM traversal and manipulation can be done using plain JavaScript, but because of cross-browser compatibility issues, many developers prefer to use jQuery's API for DOM traversal and manipulation.

In jQuery, to select elements in the DOM, you use the same selectors that you use for writing CSS. For instance, let paragraphElements = $('p') would give you access to all paragraph elements in the DOM. The $ in that snippet is the jQuery object, which has methods for DOM manipulation and traversal, among other things.

Let's look at a simple hello world app to see the idea of traversal and manipulation in action. In the JavaScript code below, we traverse the DOM to find all elements with the .js-hello-world class, and we then manipulate those elements so their inner text says "Hello world from JS".

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/jquery-hello-world-example).

Although the user never sees it, when the HTML first loads the page, it does so with an h1 element that says "Hello from original HTML". Looking at the HTML file, we can see that initially, the unordered list has a single empty list item in it.

Looking at the output panel, in the final result, we see the text "hello world from js" in 2 places.

Looking at the JavaScript code, in doHelloWorld, we first find all elements in the DOM that have the '.js-hide-it' class on them, and using jQuery's [**addClass method**](https://api.jquery.com/addclass/), we add the '.hidden' class to all matching elements. In that line, we have both **traversal** and **manipulation**. We traverse the DOM to find elements with the '.js-hide-it' class and we manipulate them by adding the '.hidden' class to them.

In the next line, we find elements with the '.js-hello-world' class, and using jQuery's [**text method**](http://api.jquery.com/text/), we change the inner text of matching elements to be "hello world from js". Again, we have 2 steps, traversing and manipulating.

Note the deliberate prefixing of 'js-' on the class names our JavaScript code targets. Any time you add class names that are there not for styling purposes but as a target for DOM manipulation, it's best practice to let the class name reflect this. You don't want your JavaScript layer to be dependent on the same class names being used for styling purposes because those might change, and your JavaScript code should continue to work even when styles change.

Finally, note the order of our <script> tags in index.html. First, we include the jQuery library, and then we link to index.js. Recall that the browser reads our HTML file top to bottom, so if you have an index.js file that depends on jQuery, that index.js file needs to come after jQuery.

**Traversal and Manipulation Methods**

When you use the $() jQuery method to target a set of elements, the object you get back is a jQuery object, and it has [**numerous traversal methods**](https://api.jquery.com/category/traversing/) that you can use to traverse the selected element(s).

The following example demonstrates 2 commonly used jQuery traversal methods: [**.find()**](https://api.jquery.com/find/) and [**.parent()**](https://api.jquery.com/parent/). .find() is used to traverse the elements contained in a jQuery selection, using a filter condition. .parent() is used to target the first parent element of a jQuery object that passes a filter condition.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/jquery-traversal-example).

In the initial HTML for this example, we've applied a hidden class to the div element containing a to-do list, making it invisible. Note that one of the list items has the class .js-complete.

When our JavaScript code runs, it targets elements with the .js-to-dos class. We call .parent('.js-parent-demo') on the resulting jQuery object to find the first parent with the class .js-parent-demo, and we remove the .hidden class from that element. Here, .js-parent-demo is the filter condition.

We use the .find('.js-complete') traversal method to target children of the .js-to-dos jQuery object that have the class .js-complete, and add the .complete CSS class to those elements in order to get the strikethrough text effect.

There are several other useful jQuery [**traversal**](https://api.jquery.com/category/traversing/) and [**manipulation methods**](https://api.jquery.com/category/manipulation/). We will not cover these methods here, but you should spend a half hour or so reviewing what's available.

**Accessibility and interactive web apps**

Before moving on to learn about event listeners, we need to briefly touch on accessibility concerns in the context of DOM manipulation.

When you're writing a modern, interactive web application, it's important to keep in mind the diverse types of users that may visit your site. Some of the questions you should be asking are:

* When new information is added to the page, will all users know it's there?
* Will they know when information is *removed*?
* Is the information easily understood by users of all abilities?

We'll soon go into more detail about the process of designing an application, but it's important to ask exploratory questions about *who* can use your app and *how* so that you're less likely to exclude someone when you start writing code.

Also remember that DOM manipulation is all about creating, modifying, and deleting HTML. That means that whatever you do to your HTML should be as clean, logical, and semantic as possible. If you need to generate a new button for the user to click on, make sure it's a semantic <button>; if you're listing some search results, you probably want a <ul> or <ol>. ARIA will also come into play in a live application, but we'll cover this and other complex issues later. For now, keep your users in mind and write the best HTML you can, and your work will go a long way. For more comprehensive recommendations on making your site accessible, view the A11Y Project's [**accessibility checklist**](https://a11yproject.com/checklist/).

**Assignment**

Create a document that you can share with your mentor (like a [**Google Doc**](https://www.google.com/docs/about/), [**DropBox Paper**](https://www.dropbox.com/paper), or [**Gist**](https://gist.github.com/)).

In your document, jot down at least 5 questions that you have about jQuery and interactive web apps so far. As you work through the rest of this module, see if you can find the answers to each of your questions. If you get to the end of the module and still have unanswered questions, be sure to share them with your mentor.

Submit a link to your document below. The link will be shared with your mentor. Be sure to adjust any sharing permissions on the doc so that your mentor can view it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | **<!doctype html>**  <html lang="en">  <head>  <meta charset="utf-8">  <title>text demo</title>  <style>  p {  color: blue;  margin: 8px;  }  b {  color: red;  }  </style>  <script src="https://code.jquery.com/jquery-3.5.0.js"></script>  </head>  <body>    <p><b>Test</b> Paragraph.</p>  <p></p>    <script>  **var** str = $( "p" ).first().text();  $( "p" ).last().html( str );  </script>    </body>  </html> |

Output

**Test** Paragraph.

Test Paragraph.

Another Example

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width">

<title>jquery hello world</title>

<link rel="shortcut icon" href="#">

<link href="index.css" rel="stylesheet" type="text/css" />

</head>

<body>

<div class='hidden js-parent-demo'>

<ul class='js-to-dos'>

<li class="js-complete">Buy milk</li>

<li>mail rent check</li>

<li>write great american novel</li>

</ul>

</div>

<script

src="https://code.jquery.com/jquery-3.2.1.min.js"

integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="

crossorigin="anonymous"></script>

<script src="index.js"></script>

</body>

</html>

.hidden {

display: none;

}

.complete {

text-decoration: line-through;

}

function doIt() {

const toDos = $('.js-to-dos');

// this will grab the first parent of toDos that has the `.js-parent-demo`

// class

toDos.parent('.js-parent-demo').removeClass('hidden');

// this will look at all children of toDos that have the `.js-complete class`

toDos.find('.js-complete').addClass('complete');

}

doIt();

output:

* ~~Buy milk~~
* mail rent check
* write great american novel

# Event listeners

**Objective:** By the end of this checkpoint, you can create event listeners to trigger DOM manipulation.

In the previous checkpoint, you learned how to traverse and manipulate the DOM, but that ability alone is not very valuable. In order to take advantage of DOM manipulation, you need to be able to alter the DOM when [**events**](https://developer.mozilla.org/en-US/docs/Web/Events) happen. Some common use cases are:

* Listening for when a user submits a form, then validating the data they submitted and displaying helpful error messages if they need to modify their entries
* Listening for when a user types a search term into an input and displaying type-ahead text
* Listening for when a user clicks an element on the page and launching an animation

jQuery provides a streamlined, cross-browser compatible [**interface for setting up event listeners**](https://learn.jquery.com/events/event-basics/), and this checkpoint will teach you what you need to know to use it.

The key idea of this checkpoint is **event-driven DOM manipulation**, which in plain English, just means writing instructions that tell the computer to listen for when specific kinds of events happen and then do something in response. Whenever the event occurs, the computer executes a **callback function**, which is a set of instructions about what to do whenever the event occurs.

For instance, you could listen for when users click a submit button on a signup form (that would be the event), and that could trigger a form validation callback function that validates password requirements. If the form is valid, its contents get posted to the server. If not, the page gets updated to display validation errors to the user, which we hope will prompt the user to update the invalid inputs and resubmit.

At the end of this checkpoint, you’ll complete 2 assignments:

1. Work through a set of 2 drills, in which you'll practice creating event listeners.
2. Build an interactive shopping list app based on a design spec.

**Key Terms**

* Event listener
* Callback function
* Event objects
* Event delegation

## Basic event handling

An event listener consists of 2 parts. First, you have to specify which event to listen for. Second, you have to provide a **callback function** that runs whenever the event occurs. Let's look at some examples.

This app listens for when users click the "Click me" button and responds by updating the DOM with the number of times the button has been clicked:

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/event-handling-example).

To implement this behavior, we use 2 event listeners. The first event we need to know about is when the page is fully loaded. At the bottom of the JavaScript file, we call jQuery ($()), passing in a callback function that we want to run when the page loads:

$(handleClicks);

When the browser finishes loading the DOM for a page, it emits a "ready" event. This event happens once per page load. jQuery will run the callback function we pass to $() after this ready event happens. You should start using the $(callbackFn) function to activate your application.

Be aware that in addition to the $(callbackFn) syntax for document ready events, earlier versions of (jQuery < 3.0) used the built-in [**.ready() method**](https://api.jquery.com/ready/). .ready has been deprecated (programmer talk for 'no longer preferred, but still allowed'), and you shouldn't use it in your own code, but you should understand that it does the same thing as $(callbackFn), as you're likely to encounter legacy code with the older syntax.

We pass handleClicks as our callback function to run when the DOM has loaded. At the top of handleClicks, we create a variable to track the number of times the user has clicked the button.

Next, we traverse the DOM to find any elements with the .js-click-counter class and set the text on those elements to the current value of click count.

After that, we have our second event listener. Any time an element with the .js-clicker class on it gets clicked, we increment our clickCount variable, then update the .js-click-counter elements with the new value.

## Event objects

Let's imagine that you are using jQuery to make a puzzle game. When the player presses the "w" key, you want their character to move forwards. When they press "a" and "d" you want them to move left and right. And when the player right-clicks on an item you want to open up a menu letting them interact with the item. So far you have learned how to call a function when the user clicks on a DOM element. But how would you work out whether the "a" or "d" key was pressed, or whether the left or right button was clicked?

The solution is to use DOM event objects. The callback functions in your event listeners always get access to an object representing the triggering event. This contains information about where you typed in the text, what keys you pressed, or which buttons were clicked. In fact, just about every piece of information you could ever want to know about how you performed an action is contained within the event object.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/jquery-event-object).

In the repl.it above, try typing in the text box and left and right-clicking the button. You should see the output <div> being filled with information about the action you performed. If you left-click, it will tell you that button 1 was clicked, and if you right-click, it will tell you that button 3 was clicked. And if you type some text, it will tell you which keys were pressed.

Let's look at the JavaScript for the repl.it:

$(**function**() {

$('button').mousedown(event =>

$('.output').text(`Button clicked: ${event.which}`)

);

$('input').keydown(event =>

$('.output').text(`Key pressed: ${event.key}`)

);

});

This code tells the browser to wait until the page loads, then listen for when a user clicks on any button element or presses a key in any input element. When one of those happens, you run a callback function (i.e., the blocks of behavior we define below). When the user clicks or types a key, the browser generates some data about the click event. The object representing that data is the event you see in function(event).

[**event.which**](https://api.jquery.com/event.which/) contains information about which button was pressed in a mousedown event and the key property contains information about which key was pressed in a keydown event. Different types of events will contain different information. For example, it wouldn't make sense for there to be a key property when you had clicked a mouse button.

## event.currentTarget

One of the most powerful properties of an event object is the currentTarget. This contains information about which DOM element the user has interacted with.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/eventcurrentTarget-example).

In the repl.it above, try clicking on the light blue section, then each of the dashed-bordered rectangles. The text inside of the clicked element gets displayed in the top section. In non-technical language, when you click on one of these sections, it's as though the yellow section at the top says "Hey which section was it that got clicked? And what's the text in that section?", and the clicked section responds "My internal text is 'fee'" (or whatever its internal text happens to be).

Here's the JavaScript for the repl.it:

$(**function**() {

$(".foo, p, ul, li").click(**function**(event) {

event.stopPropagation();

$("h2").text("event.currentTarget's text is: " + $(event.currentTarget).text());

});

});

This code tells the browser to wait until the page loads, then listen for the user clicking any element matched by the selectors .foo, p, ul, li. When one of those is clicked, a function is run (with an event object provided).

We'll circle back to event.stopPropagation() in a moment. First let's see how event.currentTarget works. In the line $("h2").text(..., we're telling the browser to find an h2 element and set its text content to be "event.currentTarget's text is: " plus whatever $(event.currentTarget).text() computes to. From observing the behavior of the repl.it, you already have a sense of what's going on here: event.currentTarget refers to the element being clicked, and we retrieve the text of the currently clicked element.

Why then do we need to do $(event.currentTarget).text() and not just event.currentTarget.text()? What's the jQuery dollar sign doing? Recall that jQuery is a JavaScript library that makes it much easier to make our pages interactive. .text() is a method from the jQuery library. In order to use the methods from jQuery, you first need to select something to act upon, using the selector ($(event.currentTarget)). The browser will find that element and then pass it to the method you want to use. In JavaScript, event.currentTarget refers to the element, as discussed above, but jQuery selectors actually have different methods and data than a non-jQuery selected element. Because of that, we need to turn event.currentTarget into a jQuery object, so the library code can add the extra methods (including .text()).

Finally, let's see what event.stopPropagation() is about. To do that, let's comment out the event.stopPropagation() line:

$(**function**() {

$(".foo, p, ul, li").click(**function**(event) {

*// event.stopPropagation();*

$("h2").text("event.currentTarget's text is: " + $(event.currentTarget).text());

});

});

Now try clicking on the dashed-bordered box that has the text "fee" in it. The text in the top yellow section changes to:

*event.currentTarget's text is: Lorem ipsum dolor sit amet, consectetur adipisicing elit. fee fie foh fum*

That's the text of the entire div.foo! What happened?

The issue is that the li element we clicked is inside of a ul, inside of a div.foo, and we're watching for events on all of those containers. When we click on an li, we're also clicking on a ul and a div.foo because those elements are the parents of the li. In practice, what happens is that the $("h2").text("event.currentTarget's text is: " + $(event.currentTarget).text()); runs once where event.currentTarget is the li, again with event.currentTarget as the ul and yet again with .foo. The last time it runs, we get the inner text content of all of div.foo.

By using event.stopPropagation(), we're telling the browser to stop the event from "bubbling up" the DOM. It will run for the innermost clicked element (in this case, the li that we've clicked), but it won't run again for the ul or div.foo.

## Preventing default element behaviors

One frequent use case for jQuery is listening for a user to submit an HTML form and then doing something in response. By default, when the browser sees that a user submits a form, it tries to submit the form to a server (as described in [**this StackOverflow answer**](https://stackoverflow.com/a/1132015)).

Later in this module, when you build your shopping list app and quiz app, you'll listen for form submission events in order to know when users add an item to the shopping list or answer a question in the quiz. In both cases, you won't want to submit form data to a server.

We can use [**jQuery's preventDefault method**](https://api.jquery.com/event.preventdefault/) to stop the default submission behavior for forms (and it can also be used to stop the default behavior for anchor tags).

In the following example, we have an example of using event.preventDefault to stop the browser's default form submission behavior.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/eventpreventDefault-demo-jQuery).

Try running this repl.it. When you enter text in the input and submit it, you should see the submitted text appear in a paragraph below the input. The text input will also be emptied out so you can submit a new entry.

Looking at the JavaScript, we have:

$('.js-form').submit(event => {

*// this stops the default form submission behavior*

event.preventDefault();

**const** userTextElement = $(event.currentTarget).find('#user-text');

$(".js-display-user-text").text(`user text is: ${userTextElement.val()}`);

userTextElement.val("");

});

This says that when a .js-form element is submitted, we should stop the default behavior (event.preventDefault()); then we should start at the thing that caused this event (event.currentTarget, or the form itself in this case) and find the element that we used to enter text. From there, we display the text the user submitted and empty the input.

Try commenting out the event.preventDefault() line and then re-running the repl.it. When you submit the form, you may briefly see your submitted text display below the form, but then it will disappear. This is because the default form submission behavior kicks in and an attempt is made to submit the form data.

## The this keyword

When writing event listeners using jQuery, the this keyword plays an important role in the callback functions that run when an event happens. Inside a callback function on an event listener, this refers to event.currentTarget. So anywhere that you have been typing out event.currentTarget, you can replace it with the this keyword.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/jquery-event-this-demo).

The repl.it above is exactly the same as the one you looked at in the event.currentTarget section, except that event.currentTarget has been replaced with this. There is no difference in how the code works. All that has changed is that you are using the this keyword to refer to the element which was clicked, rather than finding out this information from the event object.

It is more common to use this in jQuery code than event.currentTarget, so you will see that style used throughout the remainder of the course.

Remember that this only refers to the element within the callback function. Outside of the callback function, it could refer to something completely different.

Finally, note that in the context of callback functions, this will not behave as expected if you use ES6 arrow functions. If you need this to refer to the event object, stick with the function keyword.

## Event delegation

According to [**the official jQuery docs**](https://learn.jquery.com/events/event-delegation/), **event delegation**

*allows us to attach a single event listener, to a parent element, that will fire for all descendants matching a selector, whether those descendants exist now or are added in the future.*

This statement is dense but crucial to understand if you're going to be proficient with jQuery. We'll unpack it in a moment, but first let's go over 2 examples that will make the idea of event delegation clear.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/event-delegation-1).

Take a moment to play with this repl.it. First, click the box that says "tiger", and you'll see that it's removed. Next click on the "Make a pet!" button, and you'll see a blue box with a new pet appear below the existing ones. Finally, click on the box of the new pet you just created and... wait nothing happened! Why is that?

Looking at the JavaScript code for this repl.it, we have:

$(**function**() {

$('button').click(**function**(event) {

$('ul').append(

'<li>' +

['cat', 'dog', 'rock'][Math.floor(Math.random()\*3)] + '</li>'

);

});

$('li').click(**function**(event) {

**this**.remove();

});

});

This code says: when the page has loaded, be on the lookout for any clicks on the button ($('button').click()) or an <li> element ($('li').click()). If the user clicks the button, we append to the bottom of the unordered list. Specifically, we append a new <li> element with a random choice of text from the options "cat", "dog", and "rock". Don't worry about understanding the details of the ['cat', 'dog', 'rock'][Math.floor(Math.random()\*3)] line. You'll better understand how this works later in the course. For now, just know that it randomly chooses one of the strings between the brackets.

When we tested the repl.it above, that functionality worked fine, so the problem must not be there.

Moving on to the listener for the <li> click events, we have:

$('li').click(**function**(event) {

**this**.remove();

});

We just learned about this, so we know that when the code runs, this will refer to the <li> that was just clicked. When we clicked on one of the pre-existing pets on the page, they were removed as expected, but when we click on one of the pets that was added after the page loaded, it's not removed, even though this code clearly says it should be.

This behavior is a side effect of how the browser processes JavaScript files. When the page loads, the browser reads through any linked JS files in the order they were linked, each one top to bottom. When it encounters listeners like $("li").click() above, it finds all the current DOM elements targeted by the jQuery selector and listens for clicks on those specific elements. We say that the browser binds listeners to these elements, and this binding process happens once and only once: when the JavaScript code is first processed by the browser. Because of this, event listeners are effectively deaf to events related to any elements created after initial binding.

This means that if we want to detect when new pets get clicked in our repl.it example, we'll need a different approach: **event delegation**. Recall how event propagation works. When you click on the li, the event information will bubble up the DOM. Instead of watching for clicks on all <li> elements (regardless of when they were created), we'll watch for clicks on a parent element that we know existed at the time the page loaded. Since the event from the <li> will propagate to the parent, we can catch and handle it there. The repl.it below demonstrates the solution.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/event-delegation-2).

Try making a new pet, then clicking on it, and you'll see that our app works as expected: the new pet is removed.

Looking at our JavaScript for this repl.it, we have:

$('ul').on('click', 'li', **function**(event) {

**this**.remove();

});

This code uses the [**.on() method from jQuery**](http://api.jquery.com/on/). Up until now, you’ve seen .click(), and .hover() which are shortcuts for .on(‘click… and .on(‘hover’. The code above says that when the <ul> element is clicked (i.e., on any click of a <ul>), if that click is specifically on an <li> element within the <ul>, then run the callback function to remove the clicked element. Because the <ul> was there at the time our JavaScript code first ran, the browser is able to bind the event listener that will respond to all click events that happen inside that <ul>. By using .on('click', 'li', function(){}), we filter all click events within the <ul> so that the function runs only if the clicked item was an <li>.

In other words (specifically, the words of the jQuery docs):

*Event delegation allows us to attach a single event listener, to a parent element, that will fire for all descendants matching a selector, whether those descendants exist now or are added in the future.*

The moral of the story moving forward is this: if you need to use jQuery to watch for events on dynamically generated elements (that is, elements that get added to the page after the JavaScript has been loaded), don't use .click(). Instead use .on() and target the specific event you're watching for (in this case "click"). You can then, optionally, filter the targeted events for that parent element by a more specific selector (in this case "li").

## The a11y benefits of starting with HTML

It's easy to think that HTML is an inherently static language and that the work of handling user interaction must be done entirely in JavaScript. In reality, this is only partly true. Some elements, like form inputs, are meant to be interactive — to an extent. If we start with as much interactivity as HTML can give us, we'll be well on our way to creating web applications that are accessible and user-friendly before we write any JavaScript.

Consider a common real-world interface: a toggle button. Below, we've turned the image of a lightbulb — marked up in an <img> tag — into a kind of toggle button.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/lessimggreater-input-switches).

In this example, we've employed [**jQuery's .toggleClass() method**](http://api.jquery.com/toggleclass/) to turn the lightbulbs yellow when they are clicked. Unlike regular buttons, toggle buttons stay in one state until they are activated. This means that our button stays white ("off") until clicked the first time, and will stay yellow ("on") until it's clicked again. At first glance, this is fine. The light turns on and off like it should!

This mouse functionality, however, does not translate to keyboard users. Because <img> elements are not inherently interactive, keyboard users have no way to navigate to them, much less trigger the button-like behavior we are trying to give them. Try clicking on the result panel in the repl.it above, and then hitting the Tab button on your keyboard. You'll see that... nothing happens. For this UI to be accessible to keyboard users, it needs to enable users to move from lightbulb to lightbulb using Tab, and to turn the lightbulbs on and off with Space.

Since screen reader users are also often primarily keyboard users, you can be sure that an interface unavailable to a sighted keyboard user will be even more problematic to a visually impaired keyboard user. Screen readers give users a way to navigate to images, but all the user will hear is "Lightbulb one, image". In that alone, there is no indication that the user should interact with the image, much less how to do it.

So, you might be asking, what kind of image is interactive? Once again, form elements come to our rescue. The repl.it below replaces our wannabe <img> switches with [**input type="image"**](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/image).

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/ARIA-Friendly-Lightbulbs).

Image-typed inputs are graphical buttons, and as such you get all the benefits of a button: you can cycle through the switches with the Tab key, and when you do, your browser indicates which element is in focus.

Because our new inputs are available to the keyboard, they're also available to visually impaired users who rely on screen readers. Input-type images are treated like buttons, but we add aria-pressed="false" to make sure they're announced as toggle buttons, specifically. This means a screen reader user will hear Lightbulb, toggle button when they arrive at one of these buttons.

As useful as all this HTML is, we still need JavaScript to do the toggling for us. We use [**jQuery's .attr() method**](https://api.jquery.com/attr/) to set aria-pressed to true or false as needed when the buttons are activated. Now, where you can see a yellow color and know a lightbulb is on, a screen reader user will hear something like Lightbulb two, selected, toggle button.

Go ahead and test our buttons! You can use the mouse, of course, but be sure to tab to one of the buttons and press Space. It turns on as if we've clicked it! Button elements like this one (and the more familiar <button>) interpret the space key as a standard click event, no extra JavaScript required.

Although we've made considerable accessibility improvements with a few small changes, accessible interfaces go well beyond the scope of this example. Your job is not to learn everything overnight! Keep in mind that good HTML is the heart of an accessible web application, and as you build, keep asking yourself what your content means and who it is for. These kinds of questions will lead you to better solutions.

## Assignment

### Drills

The following drills will give you practice with event listeners and DOM manipulation and traversal. You won't need to alter the CSS in any of the drills, but you will need to either use existing classes in the HTML as hooks for jQuery or add your own, and you'll need to write JavaScript code that achieves the requirements of the drill.

Make sure you're logged into repl.it. For each drill, click on the fork button at the top of the repl.it to save a copy to your account.

Limit yourself to 30 minutes per drill maximum. If you're stuck after half an hour, seek enlightenment from the example solutions at the bottom of this assignment (which you should look at even if you do complete the assignment to compare your solution).

Also, note that repl.it uses its own console. If you want to view the logs from your repl.it, you'll need to click on the console tab.

#### Cat carousel

In this challenge, you need to write JavaScript that provides the following experience: when a user selects one of the thumbnail images using the mouse or keyboard, that image should be displayed in the full-size image container at the top. Updating the image should also update its alt attribute. To test keyboard interactivity, you can Tab to one of the thumbnails and press the Enter key.

Some of the methods we discussed in the previous assignment might be helpful here, but you should also use [**the jQuery docs**](https://api.jquery.com/) and Google if you're not sure what to do here. Researching on your own is a key skill, and it'll get easier with practice.

###### Updating the alt attributes of our images is not enough to make a carousel accessible! A robust, accessible carousel has [a lot of requirements](https://www.w3.org/WAI/tutorials/carousels/) beyond the scope of this drill. Your aim is to practice simple DOM interactivity for now.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/Cat-carousel-jQuery).

#### FizzBuzz

In this challenge, you'll implement **[FizzBuzz](https://en.wikipedia.org/wiki/Fizz_buzz" \t "_blank)**. FizzBuzz is a game used to teach multiples/divisors. The player is given a number to count to. Then, they count from one up to that number. For numbers divisible by 3, they substitute the word "fizz", for numbers divisible by 5, they substitute the word "buzz", and for numbers divisible by both 3 and 5, they substitute the word "fizzbuzz".

You'll create an event listener that listens for when a user submits a form indicating how high to count. This event should cause the program to create the FizzBuzz sequence up to the number requested by the user.

For each element in the FizzBuzz sequence, your code should insert an element into the .js-results div that looks like this:

<div class="fizz-buzz-item">

<span>1</span>

</div>

If the element's content is the word "fizz", the div with .fizz-buzz-item should also get the class .fizz applied. So:

<div class="fizz-buzz-item fizz">

<span>fizz</span>

</div>

If the element's content is the word "buzz" it should get the class ".buzz" applied:

<div class="fizz-buzz-item buzz">

<span>buzz</span>

</div>

If its content is the word "fizzbuzz" it should get the class "fizzbuzz" applied:

<div class="fizz-buzz-item fizzbuzz">

<span>fizzbuzz</span>

</div>

Your function should append the resulting HTML to the .js-results DOM element.

If you have problems accessing the embedded repl.it below, please [**try opening it in a separate browser tab by clicking here**](https://repl.it/@thinkful/return-of-fizz-buzz).

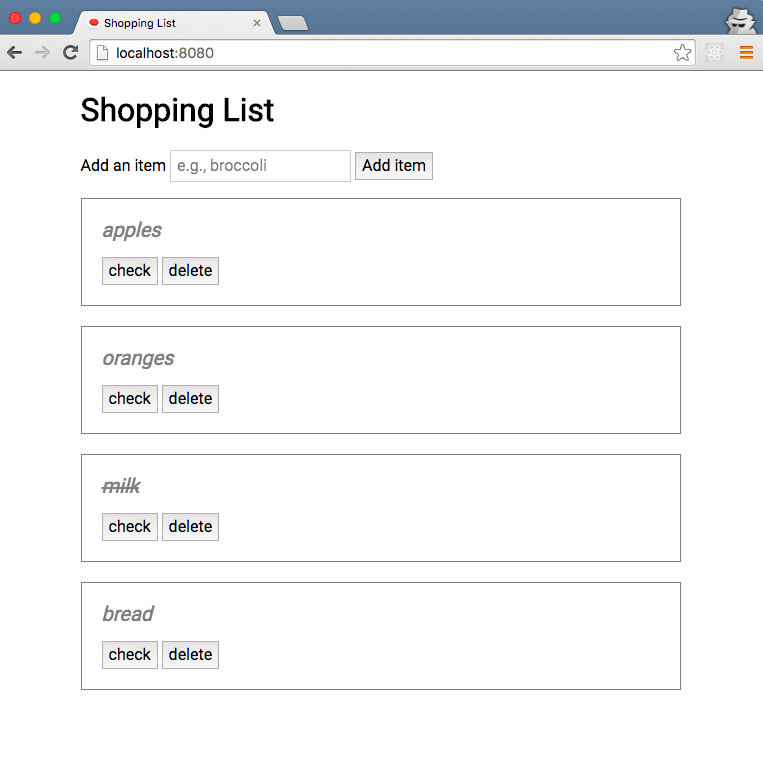
#### Solutions

Remember, spend up to a half hour on each drill before looking at these example solutions. Be sure to compare and contrast your own solution with the ones provided below.

### Shopping List challenge

In this challenge, you'll take your first stab at creating an interactive app from the ground up. Your challenge is to create an application that allows users to add, check, uncheck, and remove items from a shopping list. You'll submit your work to the grading team for feedback.

Your starting point for this challenge is [**this GitHub repo**](https://github.com/Thinkful-Ed/shopping-list), which you should fork. This repo contains HTML and CSS starter files for a page that looks like this:



As you can see, we've intentionally gone with a minimal, underdeveloped design for this project. The reason for this is twofold. First, from a learning perspective, we want you to focus on this module's main skill: using jQuery to create an interactive web page. The HTML and CSS skeleton we've provided in the starter files allows you to get up and running quickly, without having to think about designing or writing HTML and CSS for now.

Second, when you're writing software professionally, it's almost always preferable to first create an entirely unpolished but functional prototype of the app. This allows you to connect the thread between the HTML, CSS, and JavaScript layer quickly, and to start getting feedback on basic functionality. You may discover that the underlying features of the app need to drastically change or you need to add new elements, and you won't have sunk time into perfecting the styling of your content, which now needs an overhaul. Getting rapid feedback on the underlying functionality of an app is much more important than perfecting the styling at first.

To complete this challenge requires:

* Using DOM manipulation and traversal to dynamically add and remove HTML elements and apply styles.
* Linking to an externally hosted library (jQuery) and a locally hosted JavaScript file (index.js).
* Linking to your application JavaScript file from the index.html page.
* Using this and event delegation

### Requirements

In terms of user experience, your shopping list app must allow users to:

* enter items they need to purchase by entering text and hitting "Return" or clicking the "Add item" button
* check and uncheck items on the list by clicking the "Check" button
* permanently remove items from the list

Additionally:

* You must use a CDN-hosted version of jQuery
* Put your application code in a file called index.js and link to it in index.html
* Be sure to put both script elements at the bottom of the <body> element.
* Do **not** alter index.html or main.css other than adding the links to the external JavaScript inside index.html. Write JavaScript code that works with the existing HTML and CSS to implement the required features.
* Hint: you may find it helpful to read up on and use the following jQuery methods: [**.submit()**](https://api.jquery.com/submit/), **[preventDefault()](https://api.jquery.com/event.preventdefault/" \t "_blank)**, [**toggleClass()**](http://api.jquery.com/toggleclass/), and [**closest()**](https://api.jquery.com/closest/).

When you're done, push your work up to GitHub and publish it on GitHub Pages. Submit a link to your repo below.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width">

<title>Event objects example</title>

<link rel="shortcut icon" href="#">

<link href="index.css" rel="stylesheet" type="text/css" />

</head>

<body>

<main>

<h3><span class="js-click-counter">0</span> clicks</h3>

<button class="js-clicker">Click me</button>

</main>

<script

src="https://code.jquery.com/jquery-3.2.1.min.js"

integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="

crossorigin="anonymous"></script>

<script src="index.js"></script>

</body>

</html>

// this function does two things.

// First it creates a variable

// called `clickCount`. This variable

// is meant to keep track of

// the number of times the user

// has clicked the "click me"

// button. Initially we set its value

// to zero.

function handleClicks() {

let clickCount = 0;

// this line sets the inner text

// of the `.js-click-counter`

// element to the current value

// of `clickCount` (which is 0)

$('.js-click-counter').text(clickCount);

// this line says when the

// `.js-clicker` element is

// clicked, run the instructions

// inside the anonymous function

// (that is, the instructions

// between the {...} brackets).

$('.js-clicker').click(function(event) {

// any time the user clicks,

// we add 1 to the value of

// `clickCount ...

clickCount += 1;

// ...and display the updated

// click count in the

// `.js-click-count` element.

$('.js-click-counter').text(clickCount);

});

}

// this code just says that when

// the browser is done loading the

// page, it should run the

// `handleClicks` function

// we've defined above.

$(handleClicks);

### 0 clicks

Click me