

What is JQuery?

JQuery is a small, light-weight and fast JavaScript library. It is cross-platform and supports different types of browsers. It is also referred as? write less do more? Because it takes a lot of common tasks that requires many lines of JavaScript code to accomplish, and binds them into methods that can be called with a single line of code whenever needed. It is also very useful to simplify a lot of the complicated things from JavaScript, like AJAX calls and DOM manipulation.

- JQuery is a small, fast and lightweight JavaScript library.
- JQuery is platform-independent.
- JQuery means "write less do more".
- JQuery simplifies AJAX call and DOM manipulation.

JQuery Features

1. HTML manipulation
2. DOM manipulation
3. DOM element selection
4. CSS manipulation
5. Effects and Animations
6. Utilities
7. AJAX
8. HTML event methods
9. JSON Parsing
10. Extensibility through plug-ins

Why JQuery is required?

1. It is very fast and extensible.
2. It facilitates the users to write UI related function codes in minimum possible lines.
3. It improves the performance of an application.
4. Browser's compatible web applications can be developed.
5. It uses mostly new features of new browsers.

So, we can say that out of the lot of JavaScript frameworks, jQuery is the most popular and the most extendable. Many of the biggest companies on the web use jQuery.

Some of these companies are:

1. Microsoft
2. Google
3. IBM
4. Netflix

What jQuery does?

The jQuery library provides a general-purpose abstraction layer for common web scripting, and it is therefore useful in almost every scripting situation. Its extensible nature means that we could never cover all the possible uses and functions in a single session, as plugins are constantly being developed to add new abilities. The core features, though, assist us in accomplishing the following tasks:

Access elements in a document: Without a JavaScript library, web developers often need to write many lines of code to traverse the Document Object Model (DOM) tree and locate specific portions of an HTML document's structure. With jQuery, developers have a robust and efficient selector mechanism at their disposal, making it easy to retrieve the exact piece of the document that needs to be inspected or manipulated.

```
$('div.content').find('p');
```

Modify the appearance of a web page: CSS offers a powerful method of influencing the way a document is rendered, but it falls short when not all web browsers support the same standards. With jQuery, developers can bridge this gap, relying on the same standards support across all browsers. In addition, jQuery can change the classes or individual style properties applied to a portion of the document even after the page has been rendered.

```
$('ul > li:first').addClass('active');
```

Alter the content of a document: Not limited to mere cosmetic changes, jQuery can modify the content of a document itself with a few keystrokes. Text can be changed, images can be inserted or swapped, lists can be reordered, or the entire structure of the HTML can be rewritten and extended--all with a single easy-to-use Application Programming Interface (API).

```
$('#container').append('<a href="more.html">more</a>');
```

Respond to a user's interaction: Even the most elaborate and powerful behaviors are not useful if we can't control when they take place. The jQuery library offers an elegant way to intercept a wide variety of events, such as a user clicking on a link, without the need to clutter the HTML code itself with event handlers.

```
$('button.show-details').click(() => {  
    $('div.details').show();  
});
```

Animate changes being made to a document: To effectively implement such interactive behaviours, a designer must also provide visual feedback to the user. The jQuery library facilitates this by providing an array of effects such as fades and wipes, as well as a toolkit for crafting new ones.

```
$('div.details').slideDown();
```

Retrieve information from a server without refreshing a page: This pattern is known as Ajax, which originally stood for Asynchronous JavaScript and XML, but has since come to represent a

much greater set of technologies for communicating between the client and the server. The jQuery library removes the browser-specific complexity from this process, allowing developers to focus on the server-side functionality.

```
$('div.details').load('more.html #content');
```

Why jQuery works well?

Leverage knowledge of CSS: By basing the mechanism for locating page elements on CSS selectors, jQuery inherits a terse yet legible way of expressing a document's structure. The jQuery library becomes an entry point for designers who want to add behaviours to their pages because a prerequisite for doing professional web development is knowledge of CSS syntax.

Support extensions: In order to avoid "feature creep", jQuery relegates special-case uses to plugins. The method for creating new plugins is simple and well documented, which has spurred the development of a wide variety of inventive and useful modules. Even most of the features in the basic jQuery download are internally realized through the plugin architecture and can be removed if desired, yielding an even smaller library.

Abstract away browser quirks: An unfortunate reality of web development is that each browser has its own set of deviations from published standards. A significant portion of any web application can be relegated to handling features differently on each platform. While the ever-evolving browser landscape makes a perfectly browser-neutral codebase impossible for some advanced features, jQuery adds an abstraction layer that normalizes the common tasks, reducing the size of code while tremendously simplifying it.

Always work with sets: When we instruct jQuery to find all elements with the class collapsible and hide them, there is no need to loop through each returned element. Instead, methods such as `.hide()` are designed to automatically work on sets of objects instead of individual ones. This technique, called implicit iteration, means that many looping constructs become unnecessary, shortening code considerably.

Allow multiple actions in one line: To avoid overuse of temporary variables or wasteful repetition, jQuery employs a programming pattern called chaining for the majority of its methods. This means that the result of most operations on an object is the object itself, ready for the next action to be applied to it.

Setting up jQuery in an HTML document

There are three pieces to most examples of jQuery usage: the HTML document (poem.html), CSS files to style it (01.css), and JavaScript files to act on it (01.js).

```
$(() => {  
  $('div.poem-stanza').addClass('highlight')  
});
```

Finding the poem text

The fundamental operation in jQuery is selecting a part of the document. This is done with the `$()` function. Typically, it takes a string as a parameter, which can contain any CSS selector expression. In this case, we wish to find all of the `<div>` elements in the document that have the `poem-stanza` class applied to them, so the selector is very simple.

When called, the `$()` function returns a new jQuery object instance, which is the basic building block we will be working with from now on. This object encapsulates zero or more DOM elements and allows us to interact with them in many different ways. In this case, we wish to modify the appearance of these parts of the page and we will accomplish this by changing the classes applied to the poem text.

Injecting the new class

The `.addClass()` method, like most jQuery methods, is named self descriptively; it applies a CSS class to the part of the page that we have selected. Its only parameter is the name of the class to add. This method, and its counterpart, `.removeClass()`, will allow us to easily observe jQuery in action as we explore the different selector expressions available to us. For now, our example simply adds the `highlight` class, which our stylesheet has defined as italicized text with a gray background and a border.

Executing the code

Taken together, `$()` and `.addClass()` are enough for us to accomplish our goal of changing the appearance of the poem text. However, if this line of code is inserted alone in the document header, it will have no effect. JavaScript code is run as soon as it is encountered in the browser, and at the time the header is being processed, no HTML is yet present to style. We need to delay the execution of the code until after the DOM is available for our use.

With the `$(() => {})` construct (passing a function instead of a selector expression), jQuery allows us to schedule function calls for firing once the DOM is loaded, without necessarily waiting for images to fully render. While this event scheduling is possible without the aid of jQuery, `$(() => {})` provides an especially elegant cross-browser solution that includes the following features:

- It uses the browser's native DOM-ready implementations when available and adds a **window.onload** event handler as a safety net

- It executes functions passed to `$()` even if it is called after the browser event has already occurred
- It handles the event scheduling asynchronously to allow scripts to delay if necessary

The `$()` function's parameter can accept a reference to an already defined function.

```
function addHighlightClass() {  
    $('div.poem-stanza').addClass('highlight');  
}  
  
$(addHighlightClass);
```

The method can also accept an anonymous function:

```
$(( ) =>  
    $('div.poem-stanza').addClass('highlight')  
);
```

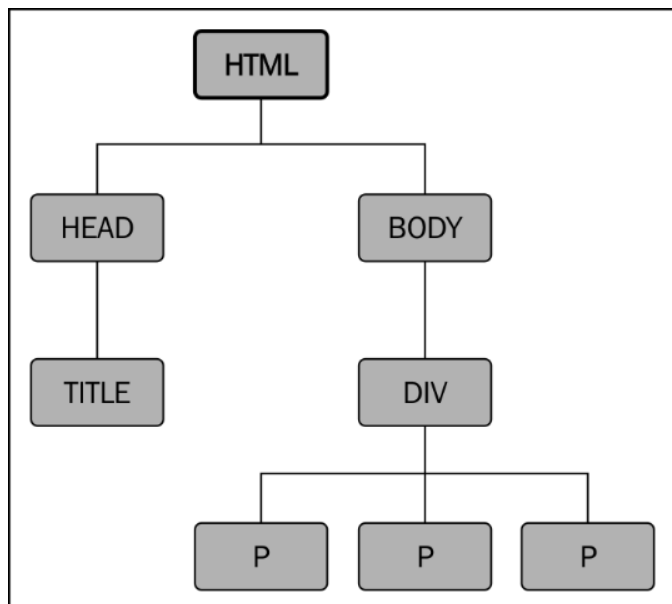
Understanding the DOM

One of the most powerful aspects of jQuery is its ability to make selecting elements in the DOM easy. The DOM serves as the interface between JavaScript and a web page; it provides a representation of the source HTML as a network of objects rather than as plain text.

This network takes the form of a family tree of elements on the page. When we refer to the relationships that elements have with one another, we use the same terminology that we use when referring to family relationships: parents, children, siblings, and so on. A simple example can help us understand how the family tree metaphor applies to a document:

```
<html>  
  <head>  
    <title>the title</title>  
  </head>  
  <body>  
    <div>  
      <p>This is a paragraph.</p>  
      <p>This is another paragraph.</p>  
      <p>This is yet another paragraph.</p>  
    </div>  
  </body>  
</html>
```

Here, **<html>** is the ancestor of all the other elements; in other words, all the other elements are descendants of **<html>**. The **<head>** and **<body>** elements are not only descendants, but children of **<html>** as well. Likewise, in addition to being the ancestor of **<head>** and **<body>**, **<html>** is also their parent. The **<p>** elements are children (and descendants) of **<div>**, descendants of **<body>** and **<html>**, and siblings of each other.



To help visualize the family tree structure of the DOM, we can use the browser's developer tools to inspect the DOM structure of any page. This is especially helpful when you're curious about how some other application works, and you want to implement something similar.

With this tree of elements at our disposal, we'll be able to use jQuery to efficiently locate any set of elements on the page. Our tools to achieve this are **jQuery selectors** and **traversal methods**.

Using the `$()` function

The resulting set of elements from jQuery's selectors and methods is always represented by a jQuery object. These objects are very easy to work with when we want to actually do something with the things that we find on a page. We can easily bind events to these objects and add visual effects to them, as well as chain multiple modifications or effects together.

In order to create a new jQuery object, we use the `$()` function. This function typically accepts a CSS selector as its sole parameter and serves as a factory, returning a new jQuery object pointing to the corresponding elements on the page.

The three primary building blocks of selectors are **tag name**, **ID**, and **class**. They can be used either on their own or in combination with others. The following simple examples illustrate how these three selectors appear in code:

Selector type	CSS	jQuery	What it does
Tag name	<code>p { }</code>	<code>\$('p')</code>	This selects all paragraphs in the document.
ID	<code>#some-id { }</code>	<code>\$('#some-id')</code>	This selects the single element in the document that has an ID of <code>some-id</code> .
Class	<code>.some-class { }</code>	<code>\$('.some-class')</code>	This selects all elements in the document that have a class of <code>some-class</code> .

CSS selectors

The jQuery library supports nearly all the selectors included in CSS specifications 1 through 3, as outlined on the World Wide Web Consortium's site: <http://www.w3.org/Style/CSS/specs>.

To begin learning how jQuery works with CSS selectors, we'll use a structure that appears on many websites, often for navigation--the nested unordered list:

```
<ul id="selected-plays">
  <li>Comedies
    <ul>
      <li><a href="/asyoulikeit/">As You Like It</a></li>
      <li>All's Well That Ends Well</li>
      <li>A Midsummer Night's Dream</li>
      <li>Twelfth Night</li>
    </ul>
  </li>
  <li>Tragedies
    <ul>
      <li><a href="hamlet.pdf">Hamlet</a></li>
      <li>Macbeth</li>
      <li>Romeo and Juliet</li>
    </ul>
  </li>
  <li>Histories
    <ul>
      <li>Henry IV (<a href="mailto:henryiv@king.co.uk">email</a>)
        <ul>
          <li>Part I</li>
          <li>Part II</li>
        </ul>
      <li><a href="http://www.shakespeare.co.uk/henryv.htm">Henry V</a></li>
      <li>Richard II</li>
    </ul>
  </li>
</ul>
```

Note that the first **** has an ID of **selected-plays**, but none of the **** tags have a class associated with them. Without any styles applied, the list looks like this:

Selected Shakespeare Plays

- Comedies
 - [As You Like It](#)
 - All's Well That Ends Well
 - A Midsummer Night's Dream
 - Twelfth Night
- Tragedies
 - [Hamlet](#)
 - Macbeth
 - Romeo and Juliet
- Histories
 - Henry IV ([email](#))
 - Part I
 - Part II
 - [Henry V](#)
 - Richard II

The nested list appears as we would expect it to--a set of bulleted items arranged vertically and indented according to their level.

Styling list-item levels

Let's suppose that we want the top-level items, and only the top-level items--Comedies, Tragedies, and Histories--to be arranged horizontally. We can start by defining a horizontal class in the stylesheet:

```
.horizontal {  
  float: left;  
  list-style: none;  
  margin: 10px;  
}
```

The **horizontal** class floats the element to the left-hand side of the one following it, removes the bullet from it if it's a list item, and adds a 10-pixel margin on all sides of it.

Rather than attaching the **horizontal** class directly in our HTML, we'll add it dynamically to the top-level list items only, to demonstrate jQuery's use of selectors:

```
$((() => {  
  $('#selected-plays > li')  
    .addClass('horizontal');  
}));
```

The second line uses the child combinator (**>**) to add the horizontal class to all the top-level items only. In effect, the selector inside the `$()` function is saying, "Find each list item (`li`) that is a **child (>)** of the element with an ID of `selected-plays` (**`#selected-plays`**)".

Output:

Selected Shakespeare Plays

Comedies

- [As You Like It](#)
- All's Well That Ends Well
- A Midsummer Night's Dream
- Twelfth Night

Tragedies

- [Hamlet](#)
- Macbeth
- Romeo and Juliet

Histories

- Henry IV ([email](#))
 - Part I
 - Part II
- [Henry V](#)
- Richard II

Styling all the other items--those that are not in the top level--can be done in a number of ways. Since we have already applied the **horizontal** class to the top-level items, one way to select all sub-level items is to use a **negation pseudo-class** to identify all list items that do not have a class of **horizontal**:

```
$((() => {  
  $('#selected-plays > li')  
    .addClass('horizontal');  
}));
```



```
$('#selected-plays li:not(.horizontal)')
.addClass('sub-level');
});
```

```
.sub-level {
  background: #ccc;
}
```

Selected Shakespeare Plays

Comedies

- [As You Like It](#)
- All's Well That Ends Well
- A Midsummer Night's Dream
- Twelfth Night

Tragedies

- [Hamlet](#)
- Macbeth
- Romeo and Juliet

Histories

- Henry IV ([email](#))
 - Part I
 - Part II
- [Henry V](#)
- Richard II

Attribute selectors

Attribute selectors are a particularly helpful subset of CSS selectors. They allow us to specify an element by one of its HTML attributes, such as a **link's title attribute** or an **image's alt attribute**. For example, to select all images that have an alt attribute, we write the following:

```
$('img[alt]')
```

Styling links

Attribute selectors accept a wildcard syntax inspired by regular expressions for identifying the value at the beginning (^) or end (\$) of a string. They can also take an asterisk (*) to indicate the value at an arbitrary position within a string or an exclamation mark (!) to indicate a negated value.

Let's say we want to have different styles for different types of links. We first define the styles in our stylesheet:

```
a {
  color: #00c;
}
a.mailto {
  background: url(images/email.png) no-repeat right top;
  padding-right: 18px;
}
a.pdfink {
  background: url(images/pdf.png) no-repeat right top;
  padding-right: 18px;
}
```

```
a.henrylink {
  background-color: #fff;
  padding: 2px;
  border: 1px solid #000;
}
```

Then, we add the three classes--**mailto**, **pdflink**, and **henrylink**--to the appropriate links using jQuery.

To add a class for all e-mail links, we construct a selector that looks for all anchor elements (**a**) with an **href** attribute (**[href]**) that begins with **mailto:** (**^="mailto:"**), as follows:

```
$((() => {
  $('a[href^="mailto:"]')
    .addClass('mailto');
}));
```

Because of the rules defined in the page's stylesheet, an envelope image appears after the **mailto:** link on the page.

Selected Shakespeare Plays

Comedies

- [As You Like It](#)
- [All's Well That Ends Well](#)
- [A Midsummer Night's Dream](#)
- [Twelfth Night](#)

Tragedies

- [Hamlet](#)
- [Macbeth](#)
- [Romeo and Juliet](#)

Histories

- [Henry IV \(email !\[\]\(b6d55d0b173caf9b2505126db01e6158_img.jpg\)](#))
 - [Part I](#)
 - [Part II](#)
- [Henry V](#)
- [Richard II](#)

To add a class for all the links to PDF files, we use the dollar sign rather than the caret symbol. This is because we're selecting links with an **href** attribute that ends with **.pdf**:

```
$((() => {
  $('a[href^="mailto:"]')
    .addClass('mailto');
  $('a[href$=".pdf"]')
    .addClass('pdflink');
}));
```


The stylesheet rule for the newly added **pdflink** class causes an Adobe Acrobat icon to appear after each link to a PDF document, as shown in the following screenshot:

Selected Shakespeare Plays

Comedies

- [As You Like It](#)
- [All's Well That Ends Well](#)
- [A Midsummer Night's Dream](#)
- [Twelfth Night](#)

Tragedies

- [Hamlet](#) 
- [Macbeth](#)
- [Romeo and Juliet](#)

Histories

- [Henry IV \(email !\[\]\(79de0df6c6ddd2d4eb74f1cc5f48ec50_img.jpg\)\)](#)
 - [Part I](#)
 - [Part II](#)
- [Henry V](#)
- [Richard II](#)

Attribute selectors can be combined as well. We can, for example, add the class **henrylink** to all links with an **href** value that both starts with **http** and contains **henry** anywhere:

```
$(() => {  
  $('a[href^="mailto:"]')  
    .addClass('mailto');  
  $('a[href$=".pdf"]')  
    .addClass('pdflink');  
  $('a[href^="http"][href*="henry"]')  
    .addClass('henrylink');  
});
```


With the three classes applied to the three types of links, we should see the following:

Selected Shakespeare Plays

Comedies

- [As You Like It](#)
- [All's Well That Ends Well](#)
- [A Midsummer Night's Dream](#)
- [Twelfth Night](#)

Tragedies

- [Hamlet](#) 
- [Macbeth](#)
- [Romeo and Juliet](#)

Histories

- [Henry IV \(email !\[\]\(a16a19bbc0e991a431a3f945e52ea4ee_img.jpg\)\)](#)
 - [Part I](#)
 - [Part II](#)
- [Henry V](#)
- [Richard II](#)

Custom selectors

To the wide variety of CSS selectors, jQuery adds its own custom selectors. These custom selectors enhance the capabilities of CSS selectors to locate page elements in new ways.

Most of the custom selectors allow us to choose one or more elements from a collection of elements that we have already found. The custom selector syntax is the same as the CSS pseudo-class syntax, where the selector starts with a colon (:). For example, to select the second item from a set of `<div>` elements with a class of `horizontal`, we write this:

```
$('.div.horizontal:eq(1)')
```

Note that `:eq(1)` selects the second item in the set because JavaScript array numbering is zero-based, meaning that it starts with zero. In contrast, CSS is one-based, so a CSS selector such as `$('div:nth-child(1)')` would select all div selectors that are the first child of their parent.

Styling alternate rows

Two very useful custom selectors in the jQuery library are `:odd` and `:even`. Let's take a look at how we can use one of them for basic table striping given the following tables:

```
<h2>Shakespeare's Plays</h2>
<table>
  <tr>
    <td>As You Like It</td>
    <td>Comedy</td>
    <td></td>
  </tr>
  <tr>
    <td>All's Well that Ends Well</td>
    <td>Comedy</td>
    <td>1601</td>
  </tr>
  <tr>
    <td>Hamlet</td>
    <td>Tragedy</td>
    <td>1604</td>
  </tr>
  <tr>
    <td>Macbeth</td>
    <td>Tragedy</td>
    <td>1606</td>
  </tr>
  <tr>
    <td>Romeo and Juliet</td>
    <td>Tragedy</td>
    <td>1595</td>
  </tr>
  <tr>
    <td>Henry IV, Part I</td>
    <td>History</td>
    <td>1596</td>
  </tr>
  <tr>
    <td>Henry V</td>
    <td>History</td>
    <td>1599</td>
  </tr>
</table>
<h2>Shakespeare's Sonnets</h2>
<table>
  <tr>
    <td>The Fair Youth</td>
    <td>1-126</td>
  </tr>
  <tr>
    <td>The Dark Lady</td>
    <td>127-152</td>
  </tr>
  <tr>
    <td>The Rival Poet</td>
```

```
<td>78-86</td>
</tr>
</table>
```

With minimal styles applied from our stylesheet, these headings and tables appear quite plain. The table has a solid white background, with no styling separating one row from the next.

Shakespeare's Plays

As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Shakespeare's Sonnets

The Fair Youth	1-126
The Dark Lady	127-152
The Rival Poet	78-86

Now, we can add a style to the stylesheet for all the table rows and use an alt class for the odd rows:

```
tr {
  background-color: #fff;
}
.alt {
  background-color: #ccc;
}
```

Finally, we write our jQuery code, attaching the class to the odd-numbered table rows (`<tr>` tags):

```
$((() => {
  $('tr:even').addClass('alt');
}));
```

But wait! Why use the `:even` selector for **odd-numbered** rows? Well, just as with the `:eq()` selector, the `:even` and `:odd` selectors use JavaScript's native zero-based numbering. Therefore, the first row counts as zero (even) and the second row counts as one (odd), and so on. With this in mind, we can expect our simple bit of code to produce tables that look like this:

Shakespeare's Plays

As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Shakespeare's Sonnets

The Fair Youth	1–126
The Dark Lady	127–152
The Rival Poet	78–86

Note that for the second table, this result may not be what we intend. Since the last row in the plays table has the alternate **gray** background, the first row in the **Sonnets** table has the plain white background. One way to avoid this type of problem is to use the `:nth-child()` selector instead, which counts an element's position relative to its parent element rather than relative to all the elements selected so far. This selector can take a number, **odd** or **even** as its argument:

```
$((() => {  
  $('tr:nth-child(odd)').addClass('alt');  
});
```

Note that `:nth-child()` is the only jQuery selector that **is one based**. To achieve the same row striping as we did earlier--except with consistent behavior for the second table--we need to use **odd** rather than **even** as the argument. With this selector in place, both tables are now striped nicely, as shown in the following screenshot:

Shakespeare's Plays

As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Shakespeare's Sonnets

The Fair Youth	1–126
The Dark Lady	127–152
The Rival Poet	78–86

Finding elements based on textual content

For one final **custom selector**, let's suppose for some reason we want to highlight any table cell that referred to one of the **Henry plays**. All we have to do--after adding a class to the stylesheet

to make the text bold and italicized (`.highlight {font-weight:bold; font-style: italic;}`)--is add a line to our jQuery code using the `:contains()` selector:

```
$(() => {  
  $('tr:nth-child(odd)')  
    .addClass('alt');  
  $('td:contains(Henry)')  
    .addClass('highlight');  
});
```

So, now we can see our lovely striped table with the **Henry** plays prominently featured:

Shakespeare's Plays

As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Shakespeare's Sonnets

The Fair Youth	1–126
The Dark Lady	127–152
The Rival Poet	78–86

Form selectors

The capabilities of custom selectors are not limited to locating elements based on their position. For example, when working with forms, jQuery's custom selectors and complementary CSS3 selectors can make short work of selecting just the elements we need. The following table describes a handful of these form selectors:

Selector	Match
<code>:input</code>	Input, text area, select, and button elements
<code>:button</code>	Button elements and input elements with a <code>type</code> attribute equal to <code>button</code>
<code>:enabled</code>	Form elements that are enabled
<code>:disabled</code>	Form elements that are disabled
<code>:checked</code>	Radio buttons or checkboxes that are checked
<code>:selected</code>	Option elements that are selected

As with the other selectors, form selectors can be combined for greater specificity. We can, for example, `select all checked radio buttons (but not checkboxes) with` `$('input[type="radio"]:checked')` or `select all password inputs and disabled text inputs with` `$('input[type="password"], input[type="text"]:disabled')`. Even with custom selectors, we can use the same basic principles of CSS to build the list of matched elements.

DOM traversal methods

The jQuery selectors that we have explored so far allow us to select a set of elements as we navigate across and down the DOM tree and filter the results. If this were the only way to select elements, our options would be somewhat limited. There are many occasions when selecting a parent or ancestor element is essential; that is where jQuery's DOM traversal methods come into play. With these methods, we can go up, down, and all around the DOM tree with ease.

Some of the methods have a nearly identical counterpart among the selector expressions. For example, the line we first used to add the alt class, `$('tr:even').addClass('alt')`, could be rewritten with the `.filter()` method as follows:

```
$('#tr')
  .filter(':even')
  .addClass('alt');
```

The `.filter()` method in particular has enormous power because it can take a function as its argument. The function allows us to create complex tests for whether elements should be included in the matched set. Let's suppose, for example, that we want to add a class to all external links:

```
a.external {
  background: #fff url(images/external.png) no-repeat 100% 2px;
  padding-right: 16px;
}
```

jQuery has no selector for this sort of thing. Without a filter function, we'd be forced to explicitly loop through each element, testing each one separately. With the following filter function, however, we can still rely on jQuery's implicit iteration and keep our code compact:

```
$('#a')
  .filter((i, a) =>
    a.hostname && a.hostname !== location.hostname
  )
  .addClass('external');
```

The supplied function filters the set of `<a>` elements by two criteria:

They must have an `href` attribute with a domain name (`a.hostname`). We use this test to exclude mailto links, for instance.

The domain name that they link to (again, `a.hostname`) must not match (`!==`) the domain name of the current page (`location.hostname`).

More precisely, the `.filter()` method iterates through the matched set of elements, calling the function once for each and testing the return value. If the function returns `false`, the element is removed from the matched set. If it returns `true`, the element is kept.

With the `.filter()` method in place, the [Henry V](#) link is styled to indicate it is external:

Selected Shakespeare Plays

Comedies

- [As You Like It](#)
- All's Well That Ends Well
- A Midsummer Night's Dream
- Twelfth Night

Tragedies

- [Hamlet](#)
- Macbeth
- Romeo and Juliet

Histories

- Henry IV ([email](#))
 - Part I
 - Part II
- [Henry V](#)
- Richard II

Styling specific cells

Earlier, we added a highlight class to all cells containing the text Henry. To instead style the cell next to each cell containing Henry, we can begin with the selector that we have already written and simply call the `.next()` method on the result:

```
$((() => {  
  $('td:contains(Henry)')  
    .next()  
    .addClass('highlight');  
}));
```

Shakespeare's Plays

As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Shakespeare's Sonnets

The Fair Youth	1-126
The Dark Lady	127-152
The Rival Poet	78-86

The `.next()` method selects only the **very next sibling** element. To highlight all of the cells following the one containing [Henry](#), we could use the `.nextAll()` method instead:

```
$((() => {  
  $('td:contains(Henry)')  
    .nextAll()  
    .addClass('highlight');  
}));
```

Since the cells containing **Henry** are in the first column of the table, this code causes the rest of the cells in these rows to be highlighted:

Shakespeare's Plays

As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Shakespeare's Sonnets

The Fair Youth	1-126
The Dark Lady	127-152
The Rival Poet	78-86

As we might expect, the `.next()` and `.nextAll()` methods have counterparts: `.prev()` and `.prevAll()`. Additionally, `.siblings()` selects all other elements at the same DOM level, regardless of whether they come before or after the previously selected element.

To include the original cell (the one that contains **Henry**) along with the cells that follow, we can add the `.addBack()` method:

```
$(() => {
  $('td:contains(Henry)')
    .nextAll()
    .addBack()
    .addClass('highlight');
});
```

With this modification in place, all of the cells in the row get their styles from the **highlight** class

Shakespeare's Plays

As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Shakespeare's Sonnets

The Fair Youth	1-126
The Dark Lady	127-152
The Rival Poet	78-86

There are a multitude of selector and traversal-method combinations by which we can select the same set of elements. Here, for example, is another way to select every cell in each row where at least one of the cells contains **Henry**:

```
$(() => {  
  $('td:contains(Henry)')  
    .parent()  
    .children()  
    .addClass('highlight');  
});
```

Rather than traversing across to sibling elements, we travel up one level in the DOM to the `<tr>` tag with `.parent()` and then select all of the row's cells with `.children()`

Chaining

The traversal method combinations that we have just explored illustrate jQuery's chaining capability. With jQuery, it is possible to select **multiple sets of elements** and do multiple things with them, all within a single line of code. This chaining not only helps keep jQuery code concise, but it can also improve a script's **performance** when the alternative is to respecify a selector.

It is also possible to break a single line of code into multiple lines for greater readability. For example, a single chained sequence of methods could be written in one line:

```
$('td:contains(Henry)').parent().find('td:eq(1)')  
  .addClass('highlight').end().find('td:eq(2)')  
    .addClass('highlight');
```

This same sequence of methods could also be written in seven lines:

```
$('td:contains(Henry)') // Find every cell containing "Henry"  
  .parent() // Select its parent  
  .find('td:eq(1)') // Find the 2nd descendant cell  
  .addClass('highlight') // Add the "highlight" class  
  .end() // Return to the parent of the cell containing "Henry"  
  .find('td:eq(2)') // Find the 3rd descendant cell  
  .addClass('highlight'); // Add the "highlight" class
```

Chaining can be like speaking a whole paragraph's worth of words in a single breath--it gets the job done quickly, but it can be hard for someone else to understand. Breaking it up into multiple lines and adding judicious comments can save more time in the long run.

Accessing DOM elements

Every selector expression and most jQuery methods return a **jQuery object**. This is almost always what we want because of the implicit iteration and chaining capabilities that it affords.

Still, there may be points in our code when we need to access a DOM element directly. For example, we may need to make a resulting set of elements available to another JavaScript library, or we might need to access an element's **tag** name, which is available as a property of the DOM element. For these admittedly rare situations, jQuery provides the **.get()** method. To access the first DOM element referred to by a jQuery object, for example, we would use **.get(0)**. So, if we want to know the tag name of an element with an ID of **my-element**, we would write:

```
$('#my-element').get(0).tagName;
```

For even greater convenience, jQuery provides a shorthand for **.get()**. Instead of writing the previous line, we can use **square brackets** immediately following the selector:

```
$('#my-element')[0].tagName;
```

Event Handling

JavaScript has several built-in ways of reacting to user interaction and other events. To make a page dynamic and responsive, we need to harness this capability so that we can, at the appropriate times, use the jQuery techniques you learned so far and the other tricks you'll learn later. While we could do this with vanilla JavaScript, jQuery enhances and extends the basic event-handling mechanisms to give them a more elegant syntax while making them more powerful at the same time.

Performing tasks on page load

We have already seen how to make jQuery react to the loading of a web page. The `$(() => {})` event handler can be used to run code that depends on HTML elements, but there's a bit more to be said about it.

Timing of code execution

We noted that `$(() => {})` was jQuery's primary way to perform tasks on page load. It is not, however, the only method at our disposal. The native `window.onload` event can do the same thing. While the two methods are similar, it is important to recognize their difference in timing.

The `window.onload` event fires when a document is **completely** downloaded to the browser. This means that every element on the page is ready to be manipulated by JavaScript, which is a boon for writing feature-rich code without worrying about load order.

On the other hand, a handler registered using `$(() => {})` is invoked when the DOM is completely ready for use. This also means that all elements are accessible by our scripts, but does not mean that every **associated file** has been downloaded. As soon as the HTML file has been downloaded and parsed into a DOM tree, the code can run.

Consider, for example, a page that presents an image gallery; such a page may have many large images on it, which we **can hide, show, move, and otherwise manipulate with jQuery**. If we set up our interface using the `onload` event, users will have to wait until each and every image is completely downloaded before they can use those features. Even worse, if behaviors are not yet attached to elements that have default behaviors (such as links), user interactions could produce unintended outcomes. However, when we use `$(() => {})` for the setup, the interface is ready to be used earlier with the correct behavior.

Handling multiple scripts on one page

The traditional mechanism for registering event handlers through JavaScript (rather than adding handler attributes right in the HTML content) is to assign a function to the DOM element's corresponding property. For example, suppose we had defined the following function:

```
function doStuff() {  
  // Perform a task...  
}
```

We could then either assign it within our HTML markup:

```
<body onload="doStuff();">
```

Or, we could assign it from within JavaScript code:

```
window.onload = doStuff;
```

Both of these approaches will cause the function to execute when the page is loaded. The advantage of the second is that the behavior is cleanly separated from the markup.

With one function, this strategy works quite well. However, suppose we have a second function as follows:

```
function doOtherStuff() {  
  // Perform another task...  
}
```

We could then attempt to assign this function to run on page load:

```
window.onload = doOtherStuff;
```

However, this assignment **trumps** the first one. The **.onload** attribute can only store **one** function reference at a time, so we can't add to the existing behavior.

The **\$(() => {})** mechanism handles this situation gracefully. Each call adds the new function to an internal queue of behaviors; when the page is loaded, all of the functions will execute. The functions will run in the order in which they were registered.

jQuery doesn't have a monopoly on workarounds to this issue. We can write a JavaScript function that calls the existing **onload** handler, then calls a passed-in handler. This approach avoids conflicts between rival handlers like **\$(() => {})** does, but lacks some of the other benefits we have discussed. In modern browsers, the **DOMContentLoaded** event can be triggered with the W3C standard **document.addEventListener()** method. However, the **\$(() => {})** is more concise and elegant.

Passing an argument to the document ready callback

In some cases, it may prove useful to use more than one JavaScript library on the same page. Since many libraries make use of the **\$** identifier (since it is short and convenient), we need a way to prevent collisions between libraries.

Fortunately, jQuery provides a method called **jQuery.noConflict()** to return control of the **\$** identifier back to other libraries. Typical usage of **jQuery.noConflict()** follows the following pattern:

```

<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>
    jQuery.noConflict();
</script>
<script src="myscript.js"></script>

```

First, the other library (**prototype.js** in this example) is included. Then, **jquery.js** itself is included, taking over **\$** for its own use. Next, a call to **.noConflict()** frees up **\$**, so that control of it reverts to the first included library (**prototype.js**). Now in our custom script, we can use both libraries, but whenever we want to use a jQuery method, we need to write **jQuery** instead of **\$** as an identifier.

The **\$(() => {})** document ready handler has one more trick up its sleeve to help us in this situation. The **callback** function we pass to it can take a single parameter--the jQuery object itself. This allows us to effectively rename it without fear of conflicts using the following syntax:

```

jQuery(($) => {
    // In here, we can use $ like normal!
});

```

Handling simple events

There are other times, apart from the loading of the page, at which we might want to perform a task. Just as JavaScript allows us to intercept the page load event with **<body onload="">** or **window.onload**, it provides similar hooks for user-initiated events such as mouse clicks (**onclick**), form fields being modified (**onchange**), and windows changing size (**onresize**). When assigned directly to elements in the DOM, these hooks have similar drawbacks to the ones we outlined for **onload**. Therefore, jQuery offers an improved way of handling these events as well.

A simple style switcher

To illustrate some event handling techniques, suppose we wish to have a single page rendered in several different styles based on user input; **we will present buttons that allow the user to toggle between a normal view, a view in which the text is constrained to a narrow column, and a view with large print for the content area.**

```

<div id="switcher" class="switcher">
  <h3>Style Switcher</h3>
  <button id="switcher-default">
    Default
  </button>
  <button id="switcher-narrow">
    Narrow Column
  </button>
  <button id="switcher-large">
    Large Print
  </button>
</div>

```

Style Switcher

Default

Narrow Column

Large Print

A Christmas Carol

In Prose, Being a Ghost Story of Christmas

by Charles Dickens

Preface

I HAVE endeavoured in this Ghostly little book, to raise the Ghost of an Idea, which shall not put my readers out of humour with themselves, with each other, with the season, or with me. May it haunt their houses pleasantly, and no one wish to lay it.

Their faithful Friend and Servant,

C. D.

December, 1843.

Stave I: Marley's Ghost

To begin with, we'll make the **Large Print** button operate. We need a bit of CSS to implement our alternative view of the page as follows:

```
body.large .chapter {  
  font-size: 1.5em;  
}
```

Our goal, then, is to apply the large class to the `<body>` tag. This will allow the stylesheet to reformat the page appropriately.

```
$('#body').addClass('large');
```

However, we want this to occur when the button is clicked, not when the page is loaded as we have seen so far. To do this, we'll introduce the `.on()` method. This method allows us to specify any DOM event and to attach a behavior to it. In this case, the event is called click, and the behavior is a function consisting of our previous one liner:

```
$(() => {  
  $('#switcher-large')  
    .on('click', () => {  
    $('#body').addClass('large');  
  });  
});
```


Style Switcher

Default

Narrow Column

Large Print

A Christmas Carol

In Prose, Being a Ghost Story of Christmas

by Charles Dickens

Preface

I HAVE endeavoured in this Ghostly little book, to raise the Ghost of an Idea, which shall not put my readers out of humour with themselves, with each other, with the season, or with me. May it haunt their houses pleasantly, and no one wish to lay it.

Their faithful Friend and Servant,

That's all there is to binding a behavior to an event. The advantages we discussed with the `$((() => {}))` document ready handler apply here as well. Multiple calls to `.on()` coexist nicely, appending additional behaviors to the same event as necessary.

This isn't necessarily the most elegant or efficient way to accomplish this task. As we proceed through, we will extend and refine this code into something we can be proud of.

Enabling the other buttons

We now have a **Large Print** button that works as advertised, but we need to apply similar handling to the other two buttons (**Default** and **Narrow Column**) to make them perform their tasks. This is straightforward: we use `.on()` to add a click handler to each of them, removing and adding classes as necessary. The new code reads as follows:

```
$(() => {
  $('#switcher-default')
    .on('click', () => {
      $('body')
        .removeClass('narrow')
        .removeClass('large');
    });

  $('#switcher-narrow')
    .on('click', () => {
      $('body')
        .addClass('narrow')
        .removeClass('large');
    });

  $('#switcher-large')
    .on('click', () => {
      $('body')
        .removeClass('narrow')
```

```
        .addClass('large');  
    });  
});
```

This is combined with a CSS rule for the narrow class:

```
body.narrow .chapter {  
    width: 250px;  
}
```

Making use of event handler context

Our switcher is behaving correctly, but we are not giving the user any feedback about which button is currently active. Our approach for handling this will be to apply the selected class to the button when it is clicked, and to remove this class from the other buttons. The selected class simply makes the button's text bold:

```
.selected {  
    font-weight: bold;  
}
```

We could accomplish this class modification as we did previously by referring to each button by ID and applying or removing classes as necessary, but, instead, we'll explore a more elegant and scalable solution that exploits the context in which event handlers run.

When any event handler is triggered, the keyword **this** refers to the DOM element to which the behavior was attached. Earlier we noted that the **\$()** function could take a DOM element as its argument; this is one of the key reasons why that facility is available. By writing **\$(this)** within the event handler, we create a jQuery object corresponding to the element, and we can act on it just as if we had located it with a CSS selector.

With this in mind, we can write the following:

```
$(this).addClass('selected');
```

Placing this line in each of the three handlers will add the class when a button is clicked. To remove the class from the other buttons, we can take advantage of jQuery's implicit iteration feature, and write:

```
$('#switcher button').removeClass('selected');
```

This line removes the class from every button inside the style switcher.

We should also add the class to the Default button when the document is ready. So, placing these in the correct order, the code is as follows:

```
$((() => {  
    $('#switcher-default')
```

```

.addClass('selected')
.on('click', function() {
  $('body')
    .removeClass('narrow');
    .removeClass('large');
  $('#switcher button')
    .removeClass('selected');
  $(this)
    .addClass('selected');
});

$('#switcher-narrow')
.on('click', function() {
  $('body')
    .addClass('narrow')
    .removeClass('large');
  $('#switcher button')
    .removeClass('selected');
  $(this)
    .addClass('selected');
});

$('#switcher-large')
.on('click', function() {
  $('body')
    .removeClass('narrow')
    .addClass('large');
  $('#switcher button')
    .removeClass('selected');
  $(this)
    .addClass('selected');
});
});

```

Now the style switcher gives appropriate feedback.

Generalizing the statements by using the handler context allows us to be yet more efficient. We can factor the highlighting routine out into a separate handler, as shown in below code, because it is the same for all three buttons:

```

$(() => {
  $('#switcher-default')
    .addClass('selected')
    .on('click', function() {
      $('body')
        .removeClass('narrow')
        .removeClass('large');
    });
  $('#switcher-narrow')
    .on('click', () => {
      $('body')
        .addClass('narrow')
        .removeClass('large');
    });
  $('#switcher-large')
    .on('click', () => {

```

```

    $('body')
      .removeClass('narrow')
      .addClass('large');
  });

  $('#switcher button')
    .on('click', function() {
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
    });
});

```

Consolidating code using event context

The code optimization we've just completed is an example of refactoring--modifying existing code to perform the same task in a more efficient or elegant way. To explore further refactoring opportunities, let's look at the behaviors we have bound to each button. The `.removeClass()` method's parameter is optional; when omitted, it removes all classes from the element. We can streamline our code a bit by exploiting this as follows:

```

$(() => {
  $('#switcher-default')
    .addClass('selected')
    .on('click', () => {
      $('body').removeClass();
    });
  $('#switcher-narrow')
    .on('click', () => {
      $('body')
        .removeClass()
        .addClass('narrow');
    });

  $('#switcher-large')
    .on('click', () => {
      $('body')
        .removeClass()
        .addClass('large');
    });

  $('#switcher button')
    .on('click', function() {
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
    });
});

```

Note that the order of operations has changed a bit to accommodate our more general class removal; we need to execute `.removeClass()` first so that it doesn't undo the call to `.addClass()`, which we perform in the same breath.

Now we are executing some of the same code in each of the button's handlers. This can be easily factored out into our general button click handler:

```
$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher button')
    .on('click', function() {
      $('body')
        .removeClass();
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
    });

  $('#switcher-narrow')
    .on('click', () => {
      $('body')
        .addClass('narrow');
    });

  $('#switcher-large')
    .on('click', () => {
      $('body')
        .addClass('large');
    });
});
```

Note that we need to move the general handler above the specific ones now. The `.removeClass()` call needs to happen before `.addClass()` executes, and we can count on this because jQuery always triggers event handlers in the order in which they were registered.

Finally, we can get rid of the specific handlers entirely by, once again, exploiting event context. Since the context keyword `this` gives us a DOM element rather than a jQuery object, we can use native DOM properties to determine the ID of the element that was clicked. We can thus bind the same handler to all the buttons, and within the handler perform different actions for each button:

```
$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher button')
    .on('click', function() {
      const bodyClass = this.id.split('-')[1];
      $('body')
        .removeClass()
        .addClass(bodyClass);
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
    });
});
```

The value of the `bodyClass` variable will be `default`, `narrow`, or `large`, depending on which button is clicked. Here, we are departing somewhat from our previous code; in that we are adding a default class to `<body>` when the user clicks on `<button id="switcher-default">`. While we do not need this class applied, it isn't causing any harm either, and the reduction of code complexity more than makes up for an unused class name.

Shorthand events

Binding a handler for an event (such as a simple `click` event) is such a common task that jQuery provides an even terser way to accomplish it; shorthand event methods work in the same way as their `.on()` counterparts with fewer keystrokes.

For example, our style switcher could be written using `.click()` instead of `.on()` as follows:

```
$(() => {
  $('#switcher-default')
    .addClass('selected');

  $('#switcher button')
    .click(function() {
      const bodyClass = this.id.split('-')[1];
      $('body')
        .removeClass()
        .addClass(bodyClass);
      $('#switcher button')
        .removeClass('selected');
      $(this)
        .addClass('selected');
    });
});
```

Shorthand event methods such as the previous one exist for the other standard DOM events such as `blur`, `keydown`, and `scroll` as well. Each shortcut method `binds` a handler to the event with the corresponding name.

Showing and hiding page elements

Suppose that we wanted to be able to hide our style switcher when it is not needed. One convenient way to hide page elements is to make them `collapsible`. We will allow one `click` on the label to hide the buttons, leaving the label alone. Another click on the label will restore the buttons. We need another class that will hide buttons:

```
.hidden {
  display: none;
}
```

We could implement this feature by storing the current state of the buttons in a variable and checking its value each time the label is clicked to know whether to add or remove the `hidden` class on the buttons. However, jQuery provides an easy way for us to add or remove a class depending on whether that class is already present--the `.toggleClass()` method:

```
$(() => {  
  $('#switcher h3')  
    .click(function() {  
      $(this)  
        .siblings('button')  
        .toggleClass('hidden');  
    });  
});
```

Event propagation

In illustrating the ability of the click event to operate on normally non-clickable page elements, we have crafted an interface that doesn't indicate that the style switcher label--just an `<h3>` element--is actually a live part of the page awaiting user interaction. To remedy this, we can give it a rollover state, making it clear that it interacts in some way with the mouse:

```
.hover {  
  cursor: pointer;  
  background-color: #afa;  
}
```

The CSS specification includes a pseudo-class called `:hover`, which allows a stylesheet to affect an element's appearance when the user's mouse cursor hovers over it. This would certainly solve our problem in this instance, but instead, we will take this opportunity to introduce jQuery's `.hover()` method, which allows us to use JavaScript to change an element's styling--and indeed, perform any arbitrary action--both when the mouse cursor enters the element and when it leaves the element.

The `.hover()` method takes two function arguments, unlike the simple event methods we have so far encountered. The first function will be executed when the mouse cursor enters the selected element, and the second is fired when the cursor leaves. We can modify the classes applied to the buttons at these times to achieve a rollover effect:

```
$(() => {  
  $('#switcher h3')  
    .hover(function() {  
      $(this).addClass('hover');  
    }, function() {  
      $(this).removeClass('hover');  
    });  
});
```

We once again use implicit iteration and event context for short and simple code. Now when hovering over the `<h3>` element, we see our class applied:

Style Switcher

[Default](#)[Narrow Column](#)[Large Print](#)

A Christmas Carol

In Prose, Being a Ghost Story of Christmas

by Charles Dickens

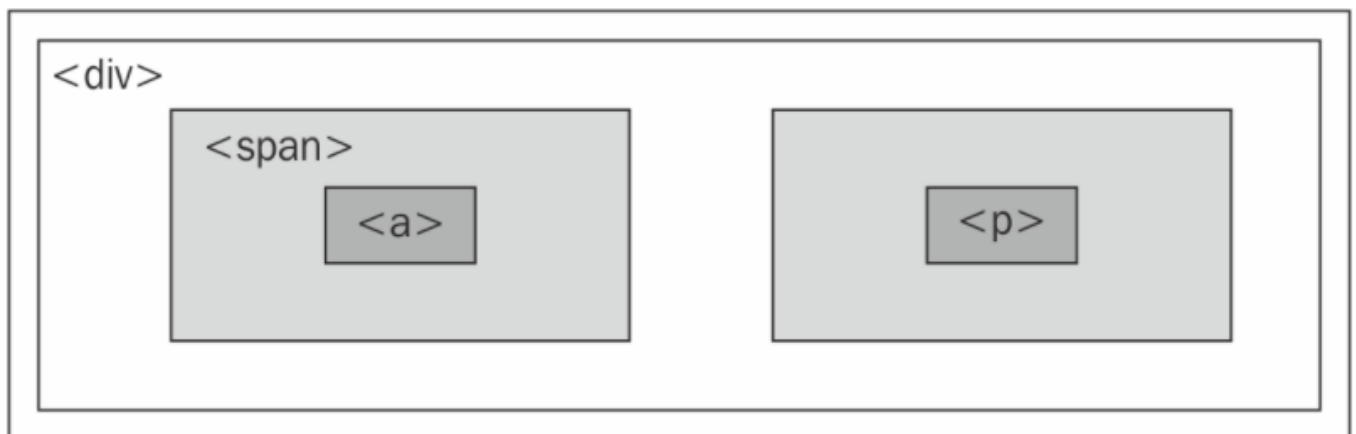
Preface

The journey of an event

When an event occurs on a page, an entire hierarchy of DOM elements gets a chance to handle the event. Consider a page model like the following:

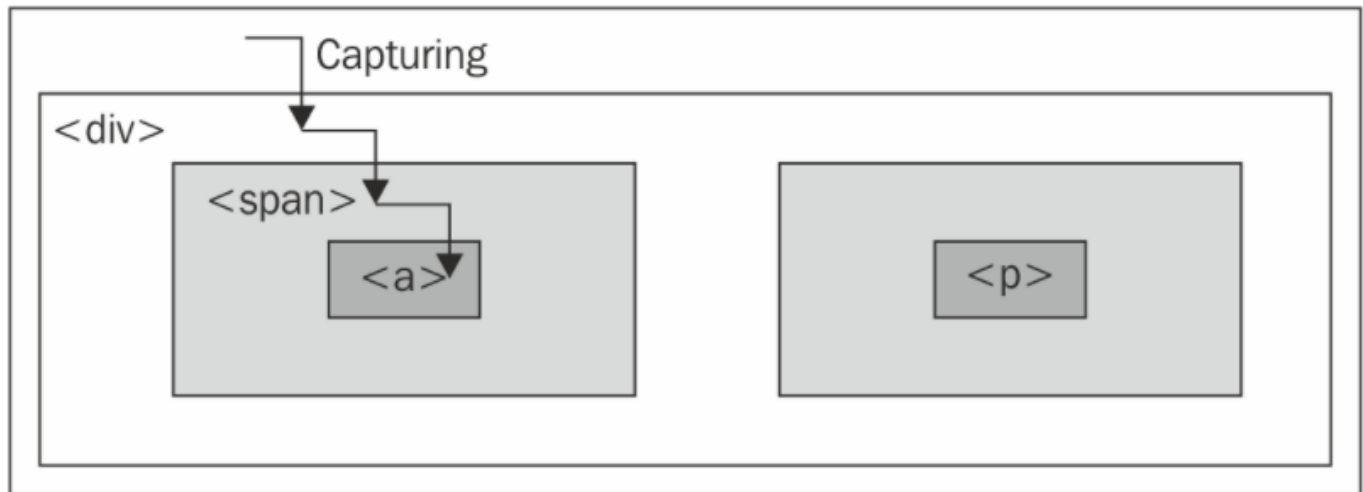
```
<div class="foo">
  <span class="bar">
    <a href="http://www.example.com/">
      The quick brown fox jumps over the lazy dog.
    </a>
  </span>
  <p>
    How razorback-jumping frogs can level six piqued gymnasts!
  </p>
</div>
```

We then visualize the code as a set of nested elements:

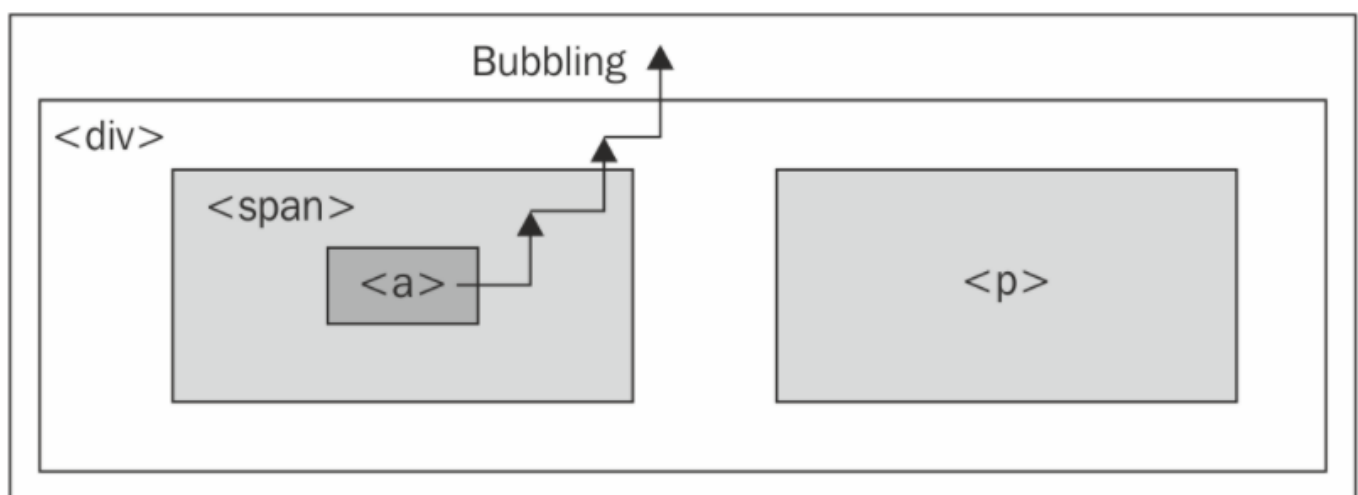


For any event, there are multiple elements that could logically be responsible for reacting. When the link on this page is clicked, for example, the `<div>`, ``, and `<a>` elements should all get the opportunity to respond to the click. After all, these three elements are all under the user's mouse cursor at the time. The `<p>` element, on the other hand, is not part of this interaction at all.

One strategy for allowing multiple elements to respond to a user interaction is called **event capturing**. With event capturing, the event is first given to the most all-encompassing element, and then to progressively more specific ones. In our example, this means that first the `<div>` element gets passed the event, then the `` element, and finally the `<a>` element, as shown in the following figure:



The opposite strategy is called **event bubbling**. The event gets sent to the most specific element, and after this element has an opportunity to react, the event bubbles up to more general elements. In our example, the `<a>` element would be handed the event first, and then the `` and `<div>` elements in that order, as shown in the following figure:



The DOM standard that was eventually developed thus specified that both strategies should be used: first the event is captured from general elements to specific ones, and then the event bubbles back up to the top of the DOM tree. Event handlers can be registered for either part of the process.

To provide consistent and easy-to-understand behavior, jQuery always registers event handlers for the bubbling phase of the model. We can always assume that the most specific element will get the first opportunity to respond to any event.

Side effects of event bubbling

Event bubbling can cause unexpected behavior, especially when the wrong element responds to a `mouseover` or `mouseout` event. Consider a `mouseout` event handler attached to the `<div>`

element in our example. When the user's mouse cursor exits the `<div>` element, the `mouseout` handler is run as anticipated. Since this is at the top of the hierarchy, no other elements get the event. On the other hand, when the cursor exits the `<a>` element, a `mouseout` event is sent to that. This event will then bubble up to the `` element and then to the `<div>` element, firing the same event handler. This **bubbling** sequence is unlikely to be desired.

The `mouseenter` and `mouseleave` events, either bound individually or combined in the `.hover()` method, are aware of these **bubbling issues** and, when we use them to attach events, we can ignore the problems caused by the wrong element getting a `mouseover` or `mouseout` event.

Altering the journey - the event object

We have already seen one situation in which event bubbling can cause problems. To show a case in which `.hover()` does not help our cause, we'll alter the collapsing behavior that we implemented earlier.

Suppose we wish to expand the **clickable** area that triggers the collapsing or expanding of the style **switcher**. One way to do this is to move the event handler from the label, `<h3>`, to its containing `<div>` element. previously, we added a click handler to `#switcher h3`; we will attempt this change by attaching the handler to `#switcher` instead:

```
$(() => {  
  $('#switcher')  
    .click(() => {  
      $('#switcher button').toggleClass('hidden');  
    });  
});
```

This alteration makes the entire area of the style switcher clickable to toggle its visibility. The downside is that clicking on a button also collapses the style switcher after the style on the content has been altered. This is due to event bubbling; the event is first handled by the buttons, then passed up through the DOM tree until it reaches the `<div id="switcher">` element, where our new handler is activated and hides the buttons.

To solve this problem, **we need access to the event object**. This is a DOM construct that is passed to each element's event handler when it is invoked. It provides information about the event, such as where the mouse cursor was at the time of the **event**. It also provides some methods that can be used to affect the progress of the event through the DOM.

To use the event object in our handlers, we only need to add a parameter to the function:

```
$(() => {  
  $('#switcher')  
    .click(function(event) {  
      $('#switcher button').toggleClass('hidden');  
    });  
});
```

```
});
```

Note that we have named this parameter **event** because it is descriptive, not because we need to. Naming it **flapjacks** or anything else for that matter would work just as well.

Event targets

Now we have the event object available to us as **event** within our handler. The property **event.target** can be helpful in controlling where an event takes effect. This property is a part of the DOM API, but is not implemented in some older browser versions; jQuery extends the **event** object as necessary to provide the property in every browser. With **.target**, we can determine which element in the DOM was the first to receive the event. In the case of a click **event**, this will be the actual item clicked on. Remembering that this gives us the DOM element handling the event, we can write the following code:

```
$(() => {
  $('#switcher')
    .click(function(event) {
      if (event.target == this) {
        $(this)
          .children('button')
          .toggleClass('hidden');
      }
    });
});
```

This code ensures that the item clicked on was **<div id="switcher">**, not one of its sub-elements. Now, clicking on buttons will not collapse the style switcher, but clicking on the switcher's background will. However, clicking on the label, **<h3>**, now does nothing, because it, too, is a sub-element. Instead of placing this check here, we can modify the behavior of the buttons to achieve our goals.

Stopping event propagation

The event object provides the **.stopPropagation()** method, which can halt the bubbling process completely for the event. Like **.target**, this method is a basic DOM feature, but using the jQuery implementation will hide any browser inconsistencies from our code.

We'll remove the **event.target == this** check we just added, and instead add some code in our buttons' click handlers:

```
$(() => {
  $('#switcher')
    .click((e) => {
      $(e.currentTarget)
        .children('button')
        .toggleClass('hidden');
    });
});
```

```

});
$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher button')
    .click((e) => {
      const bodyClass = e.target.id.split('-')[1];

      $('body')
        .removeClass()
        .addClass(bodyClass);
      $(e.target)
        .addClass('selected')
        .removeClass('selected');

      e.stopPropagation();
    });
});

```

As before, we need to add an event parameter to the function we're using as the `click` handler: `e`. Then, we simply call `e.stopPropagation()` to prevent any other DOM element from responding to the event. Now our click is handled by the buttons, and only the buttons; clicks anywhere else on the style switcher will collapse or expand it.

Preventing default actions

If our click event handler was registered on a link element (`<a>`) rather than a generic `<button>` element outside of a form, we would face another problem. When a user clicks on a link, the browser loads a new page. This behavior is not an event handler in the same sense as the ones we have been discussing; instead, this is the default action for a click on a link element.

Similarly, when the Enter key is pressed while the user is editing a form, the submit event may be triggered on the form, but then the form submission actually occurs after this.

If these default actions are undesired, calling `.stopPropagation()` on the event will not help. These actions don't occur in the normal flow of event propagation. Instead, the `.preventDefault()` method serves to stop the event in its tracks before the default action is triggered.

Event propagation and default actions are independent mechanisms; either of them can be stopped while the other still occurs. If we wish to halt both, we can return `false` at the end of our event handler, which is a shortcut for calling both `.stopPropagation()` and `.preventDefault()` on the event.

Delegating events

Event bubbling isn't always a hindrance; we can often use it to great benefit. One great technique that exploits bubbling is called `event delegation`. With it, we can use an event handler on a single element to do the work of many.

In our example, there are just three `<button>` elements that have attached `click` handlers. But what if there were many more than three? This is more common than you might think. Consider, for example, a large table of information in which each row has an interactive item requiring a click handler. Implicit iteration makes assigning all of these click handlers easy, but performance can suffer because of the looping being done internally to jQuery, and because of the memory footprint of maintaining all the handlers.

Instead, we can assign a single click handler to an ancestor element in the DOM. An uninterrupted click event will eventually reach the ancestor due to event bubbling, and we can do our work there.

As an example, let's apply this technique to our style switcher (even though the number of items does not demand the approach). As seen previously, we can use the `e.target` property to check which element is under the mouse cursor when the click event occurs.

```
$(() => {
  $('#switcher')
    .click((e) => {
      if ($(event.target).is('button')) {
        const bodyClass = e.target.id.split('-')[1];

        $('body')
          .removeClass()
          .addClass(bodyClass);
        $(e.target)
          .addClass('selected')
          .removeClass('selected');

        e.stopPropagation();
      }
    });
});
```

We've used a new method here called `.is()`. This method accepts the selector expressions we investigated in the previous topics and tests the current jQuery object against the selector. If at least one element in the set is matched by the selector, `.is()` returns `true`. In this case, `$(e.target).is('button')` asks whether the element clicked is a `<button>` element. If so, we proceed with the previous code, with one significant alteration: the keyword `this` now refers to `<div id="switcher">`, so every time we are interested in the clicked button, we must now refer to it with `e.target`.

We have an unintentional side-effect from this code, however. When a button is clicked now, the switcher collapses, as it did before we added the call to `.stopPropagation()`. The handler for the switcher visibility toggle is now bound to the same element as the handler for the buttons, so halting the event bubbling does not stop the toggle from being triggered. To sidestep this issue, we can remove the `.stopPropagation()` call and instead add another `.is()` test. Also, since we're making the entire switcher `<div>` element clickable, we ought to toggle the `hover` class while the user's mouse is over any part of it:

```

$(() => {
  const toggleHover = (e) => {
    $(e.target).toggleClass('hover');
  };

  $('#switcher')
    .hover(toggleHover, toggleHover);
});

$(() => {
  $('#switcher')
    .click((e) => {
      if (!$.is(e.target, 'button')) {
        $(e.currentTarget)
          .children('button')
          .toggleClass('hidden');
      }
    });
});

$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher')
    .click((e) => {
      if ($.is(e.target, 'button')) {
        const bodyClass = e.target.id.split('-')[1];

        $('body')
          .removeClass()
          .addClass(bodyClass);
        $(e.target)
          .addClass('selected')
          .siblings('button')
          .removeClass('selected');
      }
    });
});

```

Using built-in event delegation capabilities

Because event delegation can be helpful in so many situations, jQuery includes a set of tools to aid developers in using this technique. The `.on()` method we have already discussed can perform event delegation when provided with appropriate parameters:

```

$(() => {
  $('#switcher-default')
    .addClass('selected');
  $('#switcher')
    .on('click', 'button', (e) => {
      const bodyClass = e.target.id.split('-')[1];

      $('body')
        .removeClass()
        .addClass(bodyClass);
      $(e.target)
        .addClass('selected')
    });
});

```

```

        .siblings('button')
        .removeClass('selected');

    e.stopPropagation();
  })
  .on('click', (e) => {
    $(e.currentTarget)
      .children('button')
      .toggleClass('hidden');
  });
});

```

This is looking pretty good now. We have two really simple handlers for all click events in switcher feature. We added a selector expression to the `.on()` method as the second argument. Specifically, we want to make sure that any elements that bubble click events up to `#switch` are in fact button elements. This is better than writing a bunch of logic in the event handler to determine how to handle the event based on the element that generated it.

We did have to add a call to `e.stopPropagation()`. The reason is so that the second click handler, the one that handles toggling the button visibility, doesn't have to worry about checking where the event came from. It's often easier to prevent propagation than it is to introduce edge case handling into event handler code.

With a few minor trade offs, we now have a single button click handler function that works with 3 buttons, or with 300 buttons. It's the little things like this that make jQuery code scale well.

Removing an event handler

There are times when we will be done with an event handler we previously registered. Perhaps the state of the page has changed such that the action no longer makes sense. It is possible to handle this situation with conditional statements inside our event handlers, but it is more elegant to unbind the handler entirely.

Suppose that we want our collapsible style switcher to remain expanded whenever the page is not using the normal style. While the `Narrow Column` or `Large Print` button is selected, clicking on the background of the style switcher should do nothing. We can accomplish this by calling the `.off()` method to remove the collapsing handler when one of the non-default style switcher buttons is clicked:

```

$(() => {
  $('#switcher')
    .click((e) => {
      if (!$.e.target).is('button')) {
        $(e.currentTarget)
          .children('button')
          .toggleClass('hidden');
      }
    });
});

```

```
$('#switcher-narrow, #switcher-large')
  .click(() => {
    $('#switcher').off('click');
  });
});
```

Now when a button such as **Narrow Column** is clicked, the click handler on the style switcher **<div>** is removed, and clicking on the background of the box no longer collapses it. However, the buttons don't work anymore! They are affected by the click event of the style switcher **<div>** as well, because we rewrote the button-handling code to use event delegation. This means that when we call `$('#switcher').off('click')`, both behaviors are removed.

Giving namespaces to event handlers

We need to make our `.off()` call more specific so that it does not remove both of the click handlers we have registered. One way of doing this is to use event **namespacing**. We can introduce additional information when an event is bound that allows us to identify that particular handler later. To use **namespaces**, we need to return to the non-shorthand method of binding event handlers, the `.on()` method itself.

The first parameter we pass to `.on()` is the name of the event we want to watch for. We can use a special syntax here, though, that allows us to subcategorize the event:

```
$(() => {
  $('#switcher')
    .on('click.collapse', (e) => {
      if (!$(e.target).is('button')) {
        $(e.currentTarget)
          .children('button')
          .toggleClass('hidden');
      }
    });
  $('#switcher-narrow, #switcher-large')
    .click(() => {
      $('#switcher').off('click.collapse');
    });
});
```

The **.collapse** suffix is invisible to the event handling system; **click** events are handled by this function, just as if we wrote `.on('click')`. However, the addition of the namespace means that we can unbind just this handler without affecting the separate click handler we wrote for the buttons.

Rebinding events

Now clicking on the **Narrow Column** or **Large Print** button causes the style switcher collapsing functionality to be disabled. However, we want the behavior to return when the Default button is pressed. To do this, we will need to rebind the handler whenever Default is clicked.

First, we should give our handler function a name so that we can use it more than once without repeating ourselves:

```
$(() => {
  const toggleSwitcher = (e) => {
    if (!$.e.target.is('button')) {
      $.e.currentTarget
        .children('button')
        .toggleClass('hidden');
    }
  };

  $('#switcher')
    .on('click.collapse', toggleSwitcher);
  $('#switcher-narrow, #switcher-large')
    .click((e) => {
      $('#switcher').off('click.collapse');
    });
});
```

Recall that we are passing `.on()` a function reference as its second argument. It is important to remember when referring to a function that we must omit parentheses after the function name; parentheses would cause the function to be called rather than referenced.

Now that the `toggleSwitcher()` function can be referenced, we can bind it again later, without repeating the function definition:

```
$(() => {
  const toggleSwitcher = (e) => {
    if (!$.e.target.is('button')) {
      $.e.currentTarget
        .children('button')
        .toggleClass('hidden');
    }
  };

  $('#switcher').on('click.collapse', toggleSwitcher);
  $('#switcher-narrow, #switcher-large')
    .click(() => {
      $('#switcher').off('click.collapse');
    });
  $('#switcher-default')
    .click(() => {
      $('#switcher').on('click.collapse', toggleSwitcher);
    });
});
```

Now the toggle behavior is bound when the document is loaded, unbound when **Narrow Column or Large Print** is clicked, and rebound when **Default** is clicked after that.

Since we have named the function, we no longer need to use **namespacing**. The `.off()` method can take a function as a second argument; in this case, it unbinds only that specific handler. However, we have run into another problem. Remember that when a handler is bound to an event in jQuery, previous handlers remain in effect. In this case, each time **Default** is clicked, another copy of the `toggleSwitcher` handler is bound to the style switcher. In other words, the

function is called an extra time for each additional click until the user clicks **Narrow** or **Large Print**, which unbinds all of the **toggleSwitcher** handlers at once.

When an even number of **toggleSwitcher** handlers are bound, clicks on the style switcher (but not on a button) appear to have no effect. In fact, the hidden class is being toggled multiple times, ending up in the same state it was when it began. To remedy this problem, we can unbind the handler when a user clicks on any button, and rebind only after ensuring that the clicked button's ID is **switcher-default**:

```
$(() => {
  const toggleSwitcher = (e) => {
    if (!$.e.target.is('button')) {
      $(e.currentTarget)
        .children('button')
        .toggleClass('hidden');
    }
  };

  $('#switcher')
    .on('click', toggleSwitcher);
  $('#switcher button')
    .click((e) => {
      $('#switcher').off('click', toggleSwitcher);

      if (e.target.id == 'switcher-default') {
        $('#switcher').on('click', toggleSwitcher);
      }
    });
});
```

Simulating user interaction

At times, it is convenient to execute code that we have bound to an event, even if the event isn't triggered directly by user input. For example, suppose we wanted our style switcher to begin in its collapsed state. We could accomplish this by hiding buttons from within the stylesheet, or by adding our hidden class or calling the **.hide()** method from a **\$(() => {})** handler. Another way would be to simulate a click on the style switcher so that the toggling mechanism we've already established is triggered.

The **.trigger()** method allows us to do just this:

```
$(() => {
  $('#switcher').trigger('click');
});
```

Now when the page loads, the switcher is collapsed just as if it had been clicked.

If we were hiding content that we wanted people without JavaScript enabled to see, this would be a reasonable way to implement **graceful degradation**. Although, this is very uncommon these days.

The `.trigger()` method provides the same set of shortcut methods that `.on()` does. When these shortcuts are used with no arguments, the behavior is to trigger the action rather than bind it:

```
$(() => {  
  $('#switcher').click();  
});
```

Reacting to keyboard events

As another example, we can add keyboard shortcuts to our style switcher. When the user types the first letter of one of the display styles, we will have the page behave as if the corresponding button was clicked. To implement this feature, we will need to explore keyboard events, which behave a bit differently from mouse events.

There are two types of keyboard events: **those that react to the keyboard directly** (`keyup` and `keydown`) and **those that react to text input** (`keypress`). A single character entry event could correspond to several keys, for example, when the **Shift key** in combination with the **X key** creates the capital letter **X**. While the specifics of implementation differ from one browser to the next (unsurprisingly), a safe rule of thumb is: **if you want to know what key the user pushed, you should observe the `keyup` or `keydown` event; if you want to know what character ended up on the screen as a result, you should observe the `keypress` event**. For this feature, we just want to know when the user presses the D, N, or L key, so we will use `keyup`.

Next, we need to determine which element should watch for the event. This is a little less obvious than with mouse events, where we have a visible mouse cursor to tell us about the event's target. Instead, the target of a keyboard event is the element that currently has the keyboard focus. The element with focus can be changed in several ways, including using mouse clicks and pressing the Tab key. Not every element can get the focus, either; only items that have default **keyboard-driven** behaviors such as form **fields, links, and elements** with a `tabIndex` property are candidates.

In this case, we don't really care what element has the focus; we want our switcher to work whenever the user presses one of the keys. **Event bubbling** will once again come in handy, as we can bind our `keyup` event to the document element and have assurance that eventually any key event will bubble up to us.

Finally, we will need to know which key was pressed when our `keyup` handler gets triggered. We can inspect the event object for this. The `.which` property of the event contains an identifier for the key that was pressed, and for alphabetic keys, this identifier is the ASCII value of the uppercase letter. With this information, we can now create an object literal of letters and their corresponding buttons to **click**. When the user presses a key, we'll see if its identifier is in the map, and if so, trigger the click:

```
$(() => {  
  const triggers = {  
    D: 'default',  
    N: 'narrow',  
    L: 'large'
```

```

});

$(document)
  .keyup((e) => {
    const key = String.fromCharCode(e.which);

    if (key in triggers) {
      $('#switcher-${triggers[key]}').click();
    }
  });
});

```

As an alternative to using `.trigger()` to simulate this click, let's explore how to factor out code into a function so that more than one handler can call it--in this case, both `click` and `keyup` handlers. While not necessary in this case, this technique can be useful in eliminating code redundancy:

```

$(() => {
  // Enable hover effect on the style switcher
  const toggleHover = (e) => {
    $(e.target).toggleClass('hover');
  };

  $('#switcher').hover(toggleHover, toggleHover);

  // Allow the style switcher to expand and collapse.
  const toggleSwitcher = (e) => {
    if (!$.is(e.target, 'button')) {
      $(e.currentTarget)
        .children('button')
        .toggleClass('hidden');
    }
  };

  $('#switcher')
    .on('click', toggleSwitcher)
    // Simulate a click so we start in a collapsed state.
    .click();

  // The setBodyClass() function changes the page style.
  // The style switcher state is also updated.
  const setBodyClass = (className) => {
    $('body')
      .removeClass()
      .addClass(className);

    $('#switcher button').removeClass('selected');
    $('#switcher-${className}').addClass('selected');
    $('#switcher').off('click', toggleSwitcher);

    if (className === 'default') {
      $('#switcher').on('click', toggleSwitcher);
    }
  };

  // Begin with the switcher-default button "selected"
  $('#switcher-default').addClass('selected');

  // Map key codes to their corresponding buttons to click

```

```

const triggers = {
  D: 'default',
  N: 'narrow',
  L: 'large'
};

// Call setBodyClass() when a button is clicked.
$('#switcher')
  .click((e) => {
    if ($(e.target).is('button')) {
      setBodyClass(e.target.id.split('-')[1]);
    }
  });

// Call setBodyClass() when a key is pressed.
$(document)
  .keyup((e) => {
    const key = String.fromCharCode(e.which);

    if (key in triggers) {
      setBodyClass(triggers[key]);
    }
  });
});

```

This final revision consolidates all the previous code examples . We have moved the entire block of code into a single `$((() => {}))` handler and made our code less redundant.

Manipulating the DOM

Manipulating attributes and properties

Till now, we have been using the `.addClass()` and `.removeClass()` methods to demonstrate how we can change the appearance of elements on a page. Although we discussed these methods informally in terms of manipulating the class **attribute**, jQuery actually modifies a DOM property called **className**. The `.addClass()` method creates or adds to the property, while `.removeClass()` deletes or shortens it. Add to these the `.toggleClass()` method, which alternates between adding and removing class names, and we have an efficient and robust way of handling classes. These

methods are particularly helpful in that they avoid adding a class if it already exists on an element (so we don't end up with `<div class="first first">`, for example), and correctly handle cases where multiple classes are applied to a single element, such as `<div class="first second">`.

Non-class attributes

We may need to access or change several other attributes or properties from time to time. For manipulating attributes such as `id`, `rel`, and `href`, jQuery provides the `.attr()` and `.removeAttr()` methods. These methods make changing an attribute a simple matter

For example, we can easily set the `id`, `rel`, and `title` attributes for links all at once. Let's start with some sample HTML:

```
<h1 id="f-title">Flatland: A Romance of Many Dimensions</h1>
<div id="f-author">by Edwin A. Abbott</div>
<h2>Part 1, Section 3</h2>
<h3 id="f-subtitle">
  Concerning the Inhabitants of Flatland
</h3>
<div id="excerpt">an excerpt</div>
<div class="chapter">
  <p class="square">Our Professional Men and Gentlemen are
    Squares (to which class I myself belong) and Five-Sided
    Figures or <a
      href="http://en.wikipedia.org/wiki/Pentagon">Pentagons
    </a>.
  </p>
  <p class="nobility hexagon">Next above these come the
    Nobility, of whom there are several degrees, beginning at
    Six-Sided Figures, or <a
      href="http://en.wikipedia.org/wiki/Hexagon">Hexagons</a>,
    and from thence rising in the number of their sides till
    they receive the honourable title of <a
      href="http://en.wikipedia.org/wiki/Polygon">Polygonal</a>,
    or many-Sided. Finally when the number of the sides
    becomes so numerous, and the sides themselves so small,
    that the figure cannot be distinguished from a <a
      href="http://en.wikipedia.org/wiki/Circle">circle</a>, he
    is included in the Circular or Priestly order; and this is
    the highest class of all.
  </p>
  <p><span class="pull-quote">It is a <span class="drop">Law
    of Nature</span> with us that a male child shall have
    <strong>one more side</strong> than his father</span>, so
    that each generation shall rise (as a rule) one step in
    the scale of development and nobility. Thus the son of a
    Square is a Pentagon; the son of a Pentagon, a Hexagon;
    and so on.
  </p>
<!-- . . . code continues . . . -->
</div>
```

Now, we can iterate through each of the links inside `<div class="chapter">` and apply attributes to them one by one. If we need to set a single attribute value for all of the links, we can do so with a single line of code within our `$(() => {})` handler:

```
$(() => {  
  $('div.chapter a').attr({ rel: 'external' });  
});
```

Much like the `.css()` method, `.attr()` can accept a pair of parameters, the first specifying the attribute name and the second being its new value. More typically, though, we supply an object of key-value pairs. The following syntax allows us to easily expand our example to modify multiple attributes at once:

```
$(() => {  
  $('div.chapter a')  
    .attr({  
      rel: 'external',  
      title: 'Learn more at Wikipedia'  
    });  
});
```

Value callbacks

The straightforward technique for passing `.attr()` a simple object is sufficient when we want the attribute or attributes to have the same value for each matched element. Often, though, the attributes we add or change must have different values each time. One common example is that for any given document, each `id` value must be unique if we want our JavaScript code to behave predictably. To set a unique `id` value for each link, we can harness another feature of jQuery methods such as `.css()` and `.each()`--**value callbacks**.

A **value callback** is simply a function that is supplied instead of the value for an argument. This function is then invoked once per element in the matched set. Whatever data is returned from the function is used as the new value for the attribute. For example, we can use the following technique to generate a different `id` value for each element:

```
$(() => {  
  $('div.chapter a')  
    .attr({  
      rel: 'external',  
      title: 'Learn more at Wikipedia',  
      id: index => `wikilink-${index}`  
    });  
});
```

Each time our **value callback** is fired, it is passed an **integer** indicating the iteration count; we're using it here to give the first link an `id` value of `wikilink-0`, the second `wikilink-1`, and so on.

We are using the **title** attribute to invite people to learn more about the linked term at Wikipedia. In the HTML tags we have used so far, all of the **links** point to Wikipedia. However, to account for other types of links, we should make the selector expression a little more specific:

```
$(() => {
  $('div.chapter a[href*="wikipedia"]')
    .attr({
      rel: 'external',
      title: 'Learn more at Wikipedia',
      id: index => `wikilink-${index}`
    });
});
```

To complete our tour of the **.attr()** method, we'll enhance the **title** attribute of these links to be more specific about the link destination. Once again, a **value callback** is the right tool for the job:

```
$(() => {
  $('div.chapter a[href*="wikipedia"]')
    .attr({
      rel: 'external',
      title: function() {
        return `Learn more about ${$(this).text()} at Wikipedia.`;
      },
      id: index => `wikilink-${index}`
    });
});
```

This time we've taken advantage of the context of **value callbacks**. Just as with event handlers, the keyword **this** points to the DOM element we're manipulating each time the callback is invoked. Here, we're wrapping the element in a jQuery object so that we can use the **.text()** method to retrieve the textual content of the link. This makes each link title different from the rest, as we can see in the following screenshot:

Flatland: A Romance of Many Dimensions

by Edwin A. Abbott

Part 1, Section 3

Concerning the Inhabitants of Flatland

an excerpt

Our Professional Men and Gentlemen are Squares (to which class I myself belong) and Five-Sided Figures or [Pentagons](#).

Next above these come the Nobility, of whom there are several degrees, beginning at Six-Sided Figures, or [Hexagons](#), and from thence rising in the number of their sides till they receive the

Learn more about Pentagons at Wikipedia.

Data attributes

HTML5 data attributes allow us to attach arbitrary data values to page elements. Our jQuery code can then use these values, as well as modify them. The reason for using data attributes is so that we can separate DOM attributes that control how they're displayed and how they behave, from data that's specific to our application.

We'll use the `data()` jQuery method to read data values and to change data values. Let's add some new functionality that allows the user to mark a paragraph as read by clicking on it. We'll also need a checkbox that hides paragraphs that have been marked as read. We'll use data attributes to help us remember which paragraphs have been marked as read:

```
$(() => {  
  $('#hide-read')  
    .change((e) => {  
      if ($(e.target).is(':checked')) {  
        $('.chapter p')  
          .filter((i, p) => $(p).data('read'))  
          .hide();  
      } else {  
        $('.chapter p').show();  
      }  
    })  
});  
  
$('.chapter p')  
  .click((e) => {  
    const $elm = $(e.target);  
  
    $elm  
      .css(  
        'textDecoration',  
        $elm.data('read') ? 'none' : 'line-through'  
      )  
      .data('read', !$elm.data('read'));  
  });  
});
```

When we click on a paragraph, the text is marked with a line through it to indicate that it has been read:

Flatland: A Romance of Many Dimensions

by Edwin A. Abbott

Part 1, Section 3

Concerning the Inhabitants of Flatland

an excerpt

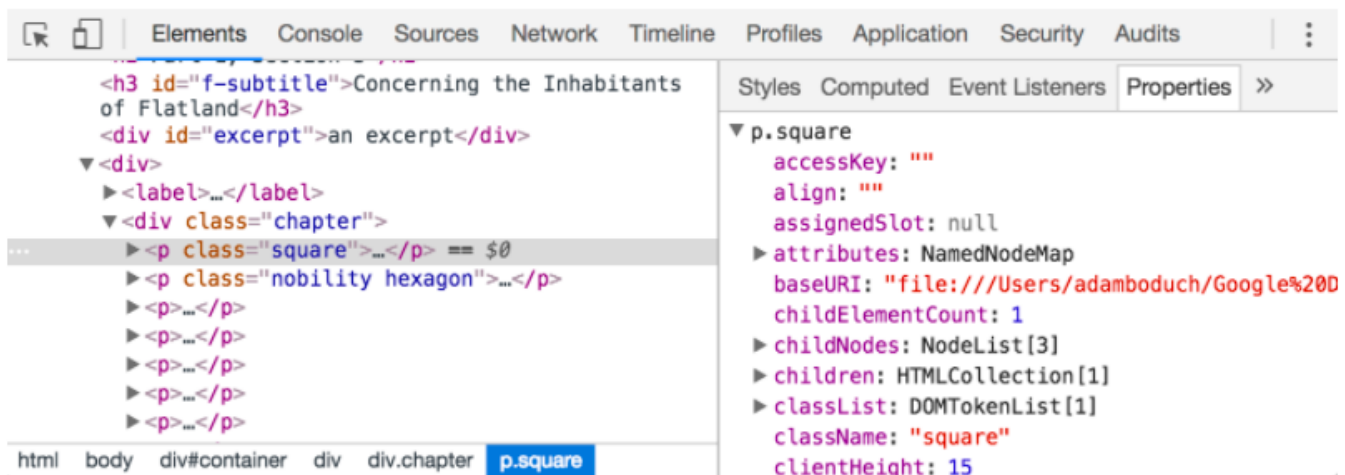
Our Professional Men and Gentlemen are Squares (to which class I myself belong) and Five-Sided Figures or [Pentagons](#).

Next above these come the Nobility, of whom there are several degrees, beginning at Six-Sided Figures, or [Hexagons](#), and from thence rising in the number of their sides till they receive the honourable title of [Polygonal](#), or many-Sided. Finally when the number of the sides becomes so numerous, and the sides themselves so small, that the figure cannot be distinguished from a [circle](#), he is included in the Circular or Priestly order; and this is the highest class of all.

It is a Law of Nature with us that a male child shall have **one more side** than his father, so that each generation shall rise (as a rule) one step in the scale of development and nobility. Thus the son of a Square is a Pentagon; the son of a Pentagon, a Hexagon; and so on.

DOM element properties

There is a subtle distinction between **HTML attributes** and **DOM properties**. Attributes are the values given in quotation marks in the HTML source for the page, while properties are the values as accessed by JavaScript. We can observe attributes and properties easily in a developer tool like Chrome's:



The **Chrome Developer** Tool's elements inspector shows us that the highlighted `<p>` element has an attribute called `class` with the value `square`. In the right panel, we can see that this element has a corresponding property called `className` with the value `square`. This illustrates one of the rare situations in which an attribute and its equivalent property have different names.

In most cases, **attributes** and **properties** are functionally interchangeable, and jQuery takes care of the naming inconsistencies for us. However, at times we do need to be mindful of the differences between the two. Some DOM properties, such as `nodeName`, `nodeType`, `selectedIndex`, and `childNodes`, have no equivalent attribute and therefore are not accessible via

`.attr()`. Moreover, data types may differ: the `checked` attribute, for example, has a string value, while the `checked` property has a Boolean value. For these Boolean attributes, it is best to test and set the property rather than the attribute to ensure consistent cross-browser behavior.

We can get and set properties from jQuery using the `.prop()` method:

```
// Get the current value of the "checked" property
const currentlyChecked = $('#my-checkbox').prop('checked');

// Set a new value for the "checked" property
$('#my-checkbox').prop('checked', false);
```

The `.prop()` method has all the same features as `.attr()`, such as accepting an object of multiple values to set at the same time and taking **value callback** functions.

The value of form controls

Perhaps the most troublesome difference between **attributes and properties** arises when trying to get or set the value of a form control. For text inputs, the `value` attribute is equivalent to the `defaultValue` property, not the `value` property. For `select` elements, the value is usually obtained via the element's `selectedIndex` property or the `selected` property of its option elements.

Because of these discrepancies, we should avoid using `.attr()`--and, in the case of `select` elements, even `.prop()`--to get or set form element values. Instead, we can use the `.val()` method, which jQuery provides for these occasions:

```
// Get the current value of a text input
const inputValue = $('#my-input').val();
// Get the current value of a select list
const selectValue = $('#my-select').val();
//Set the value of a single select list
$('#my-single-select').val('value3');
// Set the value of a multiple select list
$('#my-multi-select').val(['value1', 'value2']);
```

As with `.attr()` and `.prop()`, `.val()` can take a function for its setter argument. With its multipurpose `.val()` method, jQuery yet again makes developing for the web much easier.

DOM tree manipulation

The `.attr()` and `.prop()` methods are very powerful tools, and with them we can make targeted changes to the document. We still haven't seen ways to change the overall structure of the document though. To actually manipulate the DOM tree, you'll need to learn a bit more about the function that lies at the very heart of the **jQuery** library.

The \$() function revisited

From the starting, we've been using the `$()` function to access elements in a document. As we've seen, this function acts as a factory, producing new jQuery objects that point to the elements described by CSS selectors.

This isn't all that the `$()` function can do. It can also change the contents of a page. Simply by passing a snippet of HTML code to the function, we can create an entirely **new DOM** structure.

Creating new elements

A feature commonly seen on FAQ pages is the **back to top** link that appears after each question-and-answer pair. It could be argued that these links serve no semantic purpose, and therefore they can be legitimately included via JavaScript as an enhancement for a subset of the visitors to a page. For our example, we'll add a **back to top** link after each paragraph, as well as the anchor to which the **back to top** links will take us. To begin, we simply create the new elements:

```
$(() => {  
    $('<a href="#top">back to top</a>');  
    $('<a id="top"></a>');  
});
```

We've created a **back to top** link in the first line of code and a target anchor for the link in the second line. However, no **back to top** links appear on the page yet.

While the two lines of code that we've written do indeed create the elements, they don't yet add the elements to the page. We need to tell the browser where these new elements should go. To do that, we can use one of the many **jQuery insertion methods**.

Inserting new elements

The jQuery library has a number of methods available for inserting elements into the document. Each one dictates the relationship the new content will have to the existing content. For example, we will want our **back to top** links to **appear after each paragraph**, so we'll use the appropriately named `.insertAfter()` method to accomplish this:

```
$(() => {  
    $('<a href="#top">back to top</a>')  
        .insertAfter('div.chapter p');  
});
```

So, now that we've actually inserted the links into the page (and into the DOM) after each paragraph that appears within `<div class="chapter">`, the **back to top** links will appear:

Flatland: A Romance of Many Dimensions

by Edwin A. Abbott

Part 1, Section 3

Concerning the Inhabitants of Flatland

an excerpt

Our Professional Men and Gentlemen are Squares (to which class I myself belong) and Five-Sided Figures or [Pentagons](#)

[back to top](#)

Next above these come the Nobility, of whom there are several degrees, beginning at Six-Sided Figures, or [Hexagons](#), and from thence rising in the number of their sides till they receive the honourable title of [Polygonal](#), or many-Sided. Finally when the number of the sides becomes so numerous, and the sides themselves so small, that the figure cannot be distinguished from a [circle](#), he is included in the Circular or Priestly order; and this is the highest class of all.

[back to top](#)

It is a Law of Nature with us that a male child shall have **one more side** than his father, so that each generation shall rise (as a rule) one step in the scale of development and nobility. Thus the son of a Square is a Pentagon; the son of a Pentagon, a Hexagon; and so on.

[back to top](#)

Note that the new links appear on their own line, not within the paragraph. This is because the `.insertAfter()` method, and its counterpart `.insertBefore()`, add content outside the specified element.

Unfortunately, the links won't work yet. We still need to insert the anchor with `id="top"`. This time, we'll use one of the methods that insert elements inside of other elements:

```
$(() => {  
  $('<a href="#top">back to top</a>')  
    .insertAfter('div.chapter p');  
  $('<a id="top"></a>')  
    .prependTo('body');  
});
```

This additional code inserts the anchor right at the beginning of the `<body>` tag; in other words, at the top of the page. Now, with the `.insertAfter()` method for the links and the `.prependTo()` method for the anchor, we have a fully functioning set of **back to top** links for the page.

Once we add the corresponding `.appendTo()` method, we now have a complete set of options for inserting new elements before and after other elements:

- `.insertBefore()`: Adds content outside of and before existing elements
- `.prependTo()`: Adds content inside of and before existing elements
- `.appendTo()`: Adds content inside of and after existing elements
- `.insertAfter()`: Adds content outside of and after existing elements

Moving elements

When adding the [back to top](#) links, we created new elements and inserted them on the page. It's also possible to take elements from one place on the page and insert them into another place. A practical application of this type of insertion is the dynamic placement and formatting of **footnotes**. One footnote already appears in the original Flatland text that we are using for this example, but we'll also designate a couple of other portions of the text as footnotes for the purpose of this demonstration:

```
<p>How admirable is the Law of Compensation! <span  
  class="footnote">And how perfect a proof of the natural  
  fitness and, I may almost say, the divine origin of the  
  aristocratic constitution of the States of Flatland!</span>  
  By a judicious use of this Law of Nature, the Polygons and  
  Circles are almost always able to stifle sedition in its  
  very cradle, taking advantage of the irrepressible and  
  boundless hopefulness of the human mind.&hellip;  
</p>
```

Our HTML document contains three **footnotes**; the previous paragraph contains one example. The footnote text is inside the paragraph text, set apart using ``. By marking up the HTML document in this way, we can preserve the context of the footnote. A CSS rule applied in the stylesheet italicizes the footnotes, so an affected paragraph initially looks like the following:

How admirable is the Law of Compensation! *And how perfect a proof of the natural fitness and, I may almost say, the divine origin of the aristocratic constitution of the States of Flatland!* By a judicious use of this Law of Nature, the Polygons and Circles are almost always able to stifle sedition in its very cradle, taking advantage of the irrepressible and boundless hopefulness of the human mind....

Now, we need to grab the **footnotes** and move them to the bottom of the document. Specifically, we'll insert them in between `<div class="chapter">` and `<div id="footer">`.

Since it's important to maintain the correct order of the **footnotes** in their new place on the page, we should use `.insertBefore('#footer')`. This will place each footnote directly before the `<div id="footer">` element so that the first footnote is placed between `<div class="chapter">` and `<div id="footer">`, the second footnote is placed between the first footnote and `<div id="footer">`, and so on. Using `.insertAfter('div.chapter')`, on the other hand, would cause the footnotes to appear in reverse order.

```
$((() => {  
  $('span.footnote').insertBefore('#footer');  
}));
```

The footnotes are in `` tags, which means they display inline by default, one right after the other with no separation. However, we've anticipated this in the CSS, giving `span.footnote` elements a display value of block when they are outside of `<div class="chapter">`. So, the footnotes are now beginning to take shape:

to mutual warfare, and perish by one another's angles. No less than one hundred and twenty rebellions are recorded in our annals, besides minor outbreaks numbered at two hundred and thirty-five; and they have all ended thus.

[back to top](#)

"What need of a certificate?" a Spaceland critic may ask: "Is not the procreation of a Square Son a certificate from Nature herself, proving the Equal-sidedness of the Father?" I reply that no Lady of any position will marry an uncertified Triangle. Square offspring has sometimes resulted from a slightly Irregular Triangle; but in almost every such case the Irregularity of the first generation is visited on the third; which either fails to attain the Pentagonal rank, or relapses to the Triangular.

The Equilateral is bound by oath never to permit the child henceforth to enter his former home or so much as to look upon his relations again, for fear lest the freshly developed organism may, by force of unconscious imitation, fall back again into his hereditary level.

And how perfect a proof of the natural fitness and, I may almost say, the divine origin of the aristocratic constitution of the States of Flatland!

Wrapping elements

To number the footnotes, we could explicitly add numbers in the markup, but here we can take advantage of the standard ordered list element that takes care of numbering for us. To do this, we need to create a containing `` element surrounding all of the footnotes and an `` element surrounding each one individually. To achieve this, we'll use wrapping methods.

When wrapping elements in another element, we need to be clear about whether we want each element wrapped in its own container or all elements wrapped in a single container. For our footnote numbering, we need both types of wrapper:

```
$((() => {  
  $('span.footnote')  
    .insertBefore('#footer')  
    .wrapAll('<ol id="notes"></ol>')  
    .wrap('<li></li>');  
});
```

Once we have inserted the **footnotes** before the footer, we wrap the entire set inside a single `` element using `.wrapAll()`. We then proceed to wrap each individual footnote inside its own `` element using `.wrap()`. We can see that this has created properly numbered footnotes:

have all ended thus.

[back to top](#)

1. *"What need of a certificate?" a Spaceland critic may ask: "Is not the procreation of a Square Son a certificate from Nature herself, proving the Equal-sidedness of the Father?" I reply that no Lady of any position will marry an uncertified Triangle. Square offspring has sometimes resulted from a slightly Irregular Triangle; but in almost every such case the Irregularity of the first generation is visited on the third; which either fails to attain the Pentagonal rank, or relapses to the Triangular.*
2. *The Equilateral is bound by oath never to permit the child henceforth to enter his former home or so much as to look upon his relations again, for fear lest the freshly developed organism may, by force of unconscious imitation, fall back again into his hereditary level.*
3. *And how perfect a proof of the natural fitness and, I may almost say, the divine origin of the aristocratic constitution of the States of Flatland!*

Now, we're ready to mark and number the place from which we're pulling the footnote. To do this in a straightforward manner, though, we need to rewrite our existing code so that it doesn't rely on implicit iteration.

Explicit iteration

The `.each()` method, which acts as an explicit iterator, is very similar to the `forEach` array iterator that was recently added to the JavaScript language. The `.each()` method can be employed when the code we want to use on each of the matched elements is too complex for the implicit iteration syntax. It is passed a callback function that will be called once for each element in the matched set.

```
$(() => {
  const $notes = $('<ol id="notes"></ol>')
    .insertBefore('#footer');

  $('span.footnote')
    .each((i, span) => {
      $(span)
        .appendTo($notes)
        .wrap('<li></li>');
    });
});
```

We use the `span` parameter to create a jQuery object pointing to a single footnote, ``, then we append the element to the notes ``, and finally wrap the footnote inside an `` element.

To mark the locations in the text from which the footnotes were pulled, we can take advantage of the `.each()` callback's parameter. This parameter provides the iteration count, starting at 0 and incrementing each time the callback is invoked. Therefore, this counter will always be 1 less than number of the footnote. We'll account for this fact when producing the appropriate labels in the text:

```
$(() => {
  const $notes = $('<ol id="notes"></ol>')
    .insertBefore('#footer');
```



```

$('span.footnote')
  .each((i, span) => {
    $('<sup>${i + 1}</sup>')
      .insertBefore(span);
    $(span)
      .appendTo($notes)
      .wrap('<li></li>');
  });
});

```

Now, before each footnote is pulled out of the text to be placed at the bottom of the page, we create a new `<sup>` element containing the footnote's number and insert it into the text. The order of actions is important here; we need to make sure that the marker is inserted before the footnote is moved, or else we lose track of its initial position.

Looking at our page again, now we can see footnote markers where the inline footnotes used to be:

subject of rejoicing in our country for many furlongs round. After a strict examination conducted by the Sanitary and Social Board, the infant, if certified as Regular, is with solemn ceremonial admitted into the class of Equilaterals. He is then immediately taken from his proud yet sorrowing parents and adopted by some childless Equilateral.²

[back to top](#)

How admirable is the Law of Compensation!³ By a judicious use of this Law of Nature, the Polygons and Circles are almost always able to stifle sedition in its very cradle, taking advantage of the irrepressible and boundless hopefulness of the human mind....

Using inverted insertion methods

In the above, we inserted content before an element, then appended that same element to another place in the document. Typically, when working with elements in jQuery, we can use chaining to perform multiple actions succinctly and efficiently. We weren't able to do that here, though, because this is the target of `.insertBefore()` and the subject of `.appendTo()`. The inverted insertion methods will help us get around this limitation.

Each of the insertion methods, such as `.insertBefore()` or `.appendTo()`, has a corresponding inverted method. The inverted methods perform exactly the same task as the standard ones, but the subject and target are reversed. For example:

```

$('<p>Hello</p>').appendTo('#container');

```

Is the same as:

```
$('#container').append('<p>Hello</p>');
```

Using `.before()`, the inverted form of `.insertBefore()`, we can now re-factor our code to exploit chaining:

```
$(() => {
  const $notes = $('<ol id="notes"></ol>')
    .insertBefore('#footer');

  $('span.footnote')
    .each((i, span) => {
      $(span)
        .before(`<sup>${i + 1}</sup>`)
        .appendTo($notes)
        .wrap('<li></li>');
    });
})
```

We prepare our footnote marker using a **template string**. This is a very useful technique, but for joining a large number of strings it can start to look cluttered. Instead, we can use the array method `.join()` to construct the larger string. The following statements have the same effect:

```
var str = 'a' + 'b' + 'c';
var str = `${'a'}${'b'}${'c'}`;
var str = ['a', 'b', 'c'].join('');
```

While it requires a few more characters to type in this example, the `.join()` method can add clarity when otherwise difficult to read string concatenation or string templates. Let's look at our code again, this time using `.join()` to create the string:

```
$(() => {
  const $notes = $('<ol id="notes"></ol>')
    .insertBefore('#footer');

  $('span.footnote')
    .each((i, span) => {
      $(span)
        .before([
          '<sup>',
          i + 1,
          '</sup>'
        ].join(''))
        .appendTo($notes)
        .wrap('<li></li>');
    });
});
```

Using this technique, we can augment the footnote marker with a link to the bottom of the page as well as a unique **id** value. While we're at it, we'll also add an **id** for the `` element, so the link has a destination to point at, as shown in the following code snippet:

```

$(() => {
  const $notes = $('<ol id="notes"></ol>')
    .insertBefore('#footer');

  $('span.footnote')
    .each((i, span) => {
      $(span)
        .before([
          '<a href="#footnote-',
            i + 1,
            '" id="context-',
            i + 1,
            '" class="context">',
            '<sup>',
            i + 1,
            '</sup></a>'
        ])
        .join('')
        .appendTo($notes)
        .wrap('<li></li>');
    });
});

```

With the additional markup in place, each footnote marker now links down to the corresponding footnote at the bottom of the document. All that remains is to create a link back from the footnote to its context. For this, we can employ the inverse of the `.appendTo()` method, `.append()`:

```

$(() => {
  const $notes = $('<ol id="notes"></ol>')
    .insertBefore('#footer');

  $('span.footnote')
    .each((i, span) => {
      $(span)
        .before([
          '<a href="#footnote-',
            i + 1,
            '" id="context-',
            i + 1,
            '" class="context">',
            '<sup>',
            i + 1,
            '</sup></a>'
        ])
        .join('')
        .appendTo($notes)
        .append([
          '&nbsp;(<a href="#context-',
            i + 1,
            '">context</a>)'
        ])
        .join('')
        .wrap('<li></li>');
    });
});

```

Note that the `href` tag points back to the `id` value of the corresponding marker. In the following screenshot, you can see the footnotes again, except this time with the new link appended to each:

have all ended thus.

[back to top](#)

-
1. *"What need of a certificate?" a Spaceland critic may ask: "Is not the procreation of a Square Son a certificate from Nature herself, proving the Equal-sidedness of the Father?" I reply that no Lady of any position will marry an uncertified Triangle. Square offspring has sometimes resulted from a slightly Irregular Triangle; but in almost every such case the Irregularity of the first generation is visited on the third; which either fails to attain the Pentagonal rank, or relapses to the Triangular. ([context](#))*
 2. *The Equilateral is bound by oath never to permit the child henceforth to enter his former home or so much as to look upon his relations again, for fear lest the freshly developed organism may, by force of unconscious imitation, fall back again into his hereditary level. ([context](#))*
 3. *And how perfect a proof of the natural fitness and, I may almost say, the divine origin of the aristocratic constitution of the States of Flatland! ([context](#))*
-

Copying elements

So far, we have inserted newly created elements, moved elements from one location in the document to another, and wrapped new elements around existing ones. Sometimes, though, we may want to copy elements. For example, **a navigation menu that appears in the page's header could be copied and placed in the footer as well**. Whenever elements can be copied to enhance a page visually, we can let jQuery do the heavy lifting.

For copying elements, jQuery's `.clone()` method is just what we need; it takes any set of matched elements and creates a copy of them for later use. As in the case of the `$()` function's element creation process we explored earlier, the copied elements will not appear in the document until we apply one of the insertion methods.

For example, the following line creates a copy of the first paragraph inside `<div class="chapter">`:

```
$('#div.chapter p:eq(0)').clone();
```

This alone is not enough to change the content of the page. We can make the cloned paragraph appear before `<div class="chapter">` with an insertion method:

```
$('#div.chapter p:eq(0)')
  .clone()
  .insertBefore('div.chapter');
```

This will cause the first paragraph to appear twice. So, to use a familiar analogy, `.clone()` is related to the insertion methods just as **copy** is to **paste**.

Cloning for pull quotes

Many websites, like their print counterparts, use pull quotes to emphasize small portions of text and **attract the reader's eye**. A **pull quote** is simply an excerpt from the main document that is presented with a special graphical treatment alongside the text. We can easily accomplish this embellishment with the `.clone()` method. First, let's take another look at the third paragraph of our example text:

```
<p>
  <span class="pull-quote">It is a Law of Nature
  <span class="drop">with us</span> that a male child shall
  have <strong>one more side</strong> than his father</span>,
  so that each generation shall rise (as a rule) one step in
  the scale of development and nobility. Thus the son of a
  Square is a Pentagon; the son of a Pentagon, a Hexagon; and
  so on.
</p>
```

Note that the paragraph begins with ``. This is the class we will be targeting for **cloning**. Once the copied text inside that `` tag is pasted into another place, we need to modify its style properties to set it apart from the rest of the text.

To accomplish this type of styling, we'll add a pulled class to the copied ``. In our stylesheet, that class receives the following style rule:

```
.pulled {
  position: absolute;
  width: 120px;
  top: -20px;
  right: -180px;
  padding: 20px;
  font: italic 1.2em "Times New Roman", Times, serif;
  background: #e5e5e5;
  border: 1px solid #999;
  border-radius: 8px;
  box-shadow: 1px 1px 8px rgba(0, 0, 0, 0.6);
}
```

An element with this class is visually differentiated from the main content by applying style rules for **background**, **border**, **font**, and so on. Most importantly, it's absolutely positioned, 20 pixels above and 20 pixels to the right of the nearest (absolute or relative) positioned ancestor in the DOM. If no ancestor has positioning (other than static) applied, the **pull quote** will be positioned relative to the document `<body>`. Because of this, we need to make sure in the jQuery code that the cloned pull quote's parent element has `position:relative` set.

Now, we can consider the jQuery code needed to apply this style. We'll start with a selector expression to find all of the `` elements and apply the `position: relative` style to each parent element as we just discussed:

```
$((() => {
  $('span.pull-quote')
    .each((i, span) => {
      $(span)
```

```

        .parent()
        .css('position', 'relative');
    });
});

```

Next, we need to create the **pull quote** itself, taking advantage of the CSS we've prepared. We need to **clone** each **** tag, add the pulled class to the copy, and insert it into the beginning of its parent paragraph:

```

$(() => {
    $('span.pull-quote')
    .each((i, span) => {
        $(span)
        .clone()
        .addClass('pulled')
        .prependTo(
            $(span)
            .parent()
            .css('position', 'relative')
        );
    });
});

```

Because we're using absolute positioning for the **pull quote**, the placement of it within the paragraph is irrelevant. As long as it remains inside the paragraph, it will be positioned in relation to the top and the right of the paragraph, based on our CSS rules.

The **pull quote** now appears alongside its originating paragraph, as intended:

[back to top](#)

It is a Law of Nature with us that a male child shall have **one more side** than his father, so that each generation shall rise (as a rule) one step in the scale of development and nobility. Thus the son of a Square is a Pentagon; the son of a Pentagon, a Hexagon; and so on.

[back to top](#)

But this rule applies not always to the Tradesman, and still less often to

*It is a Law of Nature
with us that a male
child shall have **one
more side** than his
father*

Content getter and setter methods

It would be nice to be able to modify the **pull quote** a bit by dropping some words and replacing them with **ellipses** to keep the content brief. To demonstrate this, we have wrapped a few words of the example text in a `` tag.

The easiest way to accomplish this replacement is to directly specify the new HTML entity that is to replace the old one. The `.html()` method is perfect for this:

```
$(() => {  
  $('span.pull-quote')  
    .each((i, span) => {  
      $(span)  
        .clone()  
        .addClass('pulled')  
        .find('span.drop')  
          .html('&hellip;')  
          .end()  
        .prependTo(  
          $(span)  
            .parent()  
            .css('position', 'relative')  
        );  
    });  
});
```

The new lines above rely on the DOM traversal techniques we learned so far, Selecting Elements. We use `.find()` to search inside the **pull quote** for any `` elements, operate on them, and then return to the pull quote itself by calling `.end()`. In between these methods, we invoke `.html()` to change the content into an **ellipsis** (using the appropriate HTML entity).

When called without arguments, `.html()` returns a string representation of the HTML entity inside the matched element. With an argument, the contents of the element are replaced by the supplied HTML entity. We must take care to only specify a valid HTML entity, escaping special characters properly when using this technique.

The specified words have now been replaced by an ellipsis:

[back to top](#)

It is a Law of Nature with us that a male child shall have **one more side** than his father, so that each generation shall rise (as a rule) one step in the scale of development and nobility. Thus the son of a Square is a Pentagon; the son of a Pentagon, a Hexagon; and so on.

[back to top](#)

*It is a Law of Nature
... that a male child
shall have **one more**
side than his father*

Pull quotes typically do not retain their **original font formatting**, such as the **boldfaced** one more side text in this example. What we really want to display is the text of `` stripped of any ``, ``, `<a href>`, or other inline tags. To replace all of the **pull-quote** HTML entities with a **stripped, text-only** version, we can employ the `.html()` method's companion method, `.text()`.

Like `.html()`, the `.text()` method can either retrieve the content of the matched element or replace its content with a new string. Unlike `.html()`, however, `.text()` always gets or sets a plain text string. When `.text()` retrieves content, all of the included tags are ignored, and HTML entities are translated into plain characters. When it sets content, special characters such as `<` are translated into their HTML entity equivalents:

```
$(() => {
  $('span.pull-quote')
    .each((i, span) => {
      $(span)
        .clone()
        .addClass('pulled')
        .find('span.drop')
          .html('&hellip;')
          .end()
        .text((i, text) => text)
        .prependTo(
          $(span)
            .parent()
            .css('position', 'relative')
        );
    });
});
```

When you retrieve values using `text()`, the markup is removed. This is exactly what we're trying to accomplish with this example. As with some of the other jQuery functions that you've learned about so far, `text()` accepts a function. The return value is used to set the text of the element, while the current text is passed in as the second argument. So to **strip** tags from element text, just call `text((i, text) => text)`. Awesome!

Here's the results of this approach:

[back to top](#)

It is a Law of Nature with us that a male child shall have **one more side** than his father, so that each generation shall rise (as a rule) one step in the scale of development and nobility. Thus the son of a Square is a Pentagon; the son of a Pentagon, a Hexagon; and so on.

[back to top](#)

But this rule applies not always to the Tradesman, and still less often to

*It is a Law of Nature
... that a male child
shall have one more
side than his father*

DOM manipulation methods in a nutshell

The extensive DOM manipulation methods that jQuery provides vary according to their task and their target location

1. To create new elements from HTML, use the `$()` function
2. To insert new elements inside every matched element, use the following functions:
 - `.append()`
 - `.appendTo()`
 - `.prepend()`
 - `.prependTo()`
3. To insert new elements adjacent to every matched element, use the following functions:
 - `.after()`
 - `.insertAfter()`
 - `.before()`
 - `.insertBefore()`
4. To insert new elements around every matched element, use the following functions:
 - `.wrap()`
 - `.wrapAll()`
 - `.wrapInner()`
5. To replace every matched element with new elements or text, use the following functions:
 - `.html()`
 - `.text()`
 - `.replaceAll()`
 - `.replaceWith()`
6. To remove elements inside every matched element, use the following function:
 - `.empty()`
7. To remove every matched element and descendants from the document without actually deleting them, use the following functions:
 - `.remove()`
 - `.detach()`