

jQuery

learn.jquery.com
DOCUMENTATION

jQuery Learning Center

<http://learn.jquery.com/>

eBook compiled from the source

by [@offDocs](#)

[LeanPub](#) - [Lektu](#)

Date: Friday, 19-May-23 09:28:56 CEST

Follow this book at [LeanPub](#) or [Lektu](#) to get the latests updates.

Contents

- [About This Site](#)
- [Contributing](#)
- [About jQuery](#)
- [How jQuery Works](#)
- [Additional jQuery Support](#)
- [Using jQuery Core](#)
- [\\$ vs \\$\(\)](#)
- [\\$\(document \).ready\(\)](#)
- [Avoiding Conflicts with Other Libraries](#)
- [Attributes](#)
- [Selecting Elements](#)
- [Working with Selections](#)
- [Manipulating Elements](#)
- [The jQuery Object](#)
- [Traversing](#)
- [CSS, Styling, & Dimensions](#)
- [Data Methods](#)
- [Utility Methods](#)
- [Iterating over jQuery and non-jQuery Objects](#)
- [Using jQuery's .index\(\) Function](#)
- [Frequently Asked Questions](#)
- [How do I select an item using class or ID?](#)
- [How do I select elements when I already have a DOM element?](#)
- [How do I test whether an element has a particular class?](#)
- [How do I test whether an element exists?](#)
- [How do I determine the state of a toggled element?](#)
- [How do I select an element by an ID that has characters used in CSS notation?](#)
- [How do I disable/enable a form element?](#)
- [How do I check/uncheck a checkbox input or radio button?](#)
- [How do I get the text value of a selected option?](#)
- [How do I replace text from the 3rd element of a list of 10 items?](#)
- [How do I pull a native DOM element from a jQuery object?](#)
- [Events](#)
- [jQuery Event Basics](#)
- [Event Helpers](#)
- [Introducing Events](#)
- [Handling Events](#)
- [Inside the Event Handling Function](#)
- [Understanding Event Delegation](#)
- [Triggering Event Handlers](#)
- [History of jQuery Events](#)
- [Introducing Custom Events](#)
- [jQuery Event Extensions](#)
- [Effects](#)
- [Introduction to Effects](#)
- [Custom Effects with .animate\(\)](#)
- [Queue & Dequeue Explained](#)
- [Ajax](#)
- [Key Concepts](#)
- [jQuery's Ajax-Related Methods](#)
- [Ajax and Forms](#)
- [Working with JSONP](#)
- [Ajax Events](#)
- [Plugins](#)
- [Finding & Evaluating Plugins](#)
- [How to Create a Basic Plugin](#)
- [Advanced Plugin Concepts](#)
- [Writing Stateful Plugins with the jQuery UI Widget Factory](#)
- [Performance](#)
- [Append Outside of Loops](#)
- [Cache Length During Loops](#)
- [Detach Elements to Work with Them](#)

[Don't Act on Absent Elements](#)
[Optimize Selectors](#)
[Use Stylesheets for Changing CSS on Many Elements](#)
[Don't Treat jQuery as a Black Box](#)
[Code Organization](#)
[Code Organization Concepts](#)
[Beware Anonymous Functions](#)
[Keep Things DRY](#)
[Feature & Browser Detection](#)
[Deferreds](#)
[jQuery Deferreds](#)
[Deferred examples](#)
[jQuery UI](#)
[Getting Started with jQuery UI](#)
[How jQuery UI Works](#)
[Theming jQuery UI](#)
[Using jQuery UI ThemeRoller](#)
[Widget Factory](#)
[Why Use the Widget Factory?](#)
[How To Use the Widget Factory](#)
[Widget Method Invocation](#)
[Extending Widgets with the Widget Factory](#)
[Using the classes Option](#)
[Using jQuery UI](#)
[Using jQuery UI with AMD](#)
[Using jQuery UI with Bower](#)
[jQuery Mobile](#)
[Getting Started with jQuery Mobile](#)
[Creating a Custom Theme with ThemeRoller](#)

About This Site

Learning how and when to use jQuery is a different process for each and every web developer, depending largely on experience with the primary tools for front-end development (HTML, CSS, and JavaScript) and knowledge of general programming principles. Over the years developers of all stripes have come to rely on our [API documentation](#) for help figuring out how to do exactly what they need to do.

However, API documentation alone cannot serve as a guide to solving problems and fostering a true understanding of web development. Over the years, an ecosystem of blog posts, books, support forums, and channels has grown to help cover the **hows** and **whys** of developing with jQuery, as well as explaining best practices, techniques, and workarounds for common problems. This type of documentation has been an invaluable resource for millions of people, but the experience of navigating these waters can be frustrating as often as it is fulfilling, as developers struggle to identify trustworthy resources, determine whether what they're reading is actually up to date, and figure out those magical search keywords that are *just right*!

This site represents the jQuery Foundation's ongoing effort to consolidate and curate this information in order to provide this crucial "narrative documentation" to our community and serve the following goals:

1. Provide our **users** with a digestible reference on all aspects of using jQuery, from the basics of getting started and performing common tasks to more advanced topics like approaches to structuring code and where jQuery fits into modern web application development.
2. Provide our **contributors** a central, open place to collaborate and provide a dependable, highly sharable resource that will improve our users' support experiences.
3. Foster an environment by which users are encouraged to become contributors and build the skills to help them work on jQuery – or any other open source project!

In order to achieve these goals, all of [this site's content is maintained publicly on GitHub](#) and is licensed under the [MIT License](#). To learn more about how the site works, take a look at our [contributing guide](#).

History

The jQuery Learning site has its roots in two primary places.

The first is Rebecca Murphey's [jQuery Fundamentals](#), a free, open source book on jQuery basics she originally released in 2010. Seeking a better home where the information could be both maintained going forward, and consumed in a more piecemeal fashion, Rebecca donated the content to the jQuery Foundation to form the basis of what was then an abstract idea for some sort of "learning center."

The second is [docs.jquery.com](#), that erstwhile chestnut still living out its final days before it will be shut down in early 2013. Since we've moved the API documentation for jQuery Core off that domain, we needed a place that could serve a similar need – documentation (that anyone can contribute to) that gets into the "how-to" and FAQs – without clumsy barriers to entry like finding the right person to set you up with a special wiki account and forcing all authoring into a <textarea>.

Contributing

This project wouldn't have been possible without all our [awesome contributors](#). If you feel like something on the site is open for improvements, you can contribute [on GitHub](#). Feel free to check out [the contributing guide](#) for more in-depth information.

About the Beta

Though this resource will never truly be "done," the current version of this site should still be considered something of a preview. We still have a number of improvements we want to make to the content, user experience, and site build before we're ready to call it a "final release." At the same time, however, it's important for us to open the doors now so we can begin providing better docs to people who need them right away and spread the word about this effort. If

you're interested in helping us reach the finish line, we invite you to please read more about how [you can get involved with contributing](#)!

Contributing

Depending on your level of experience with some of the workflows common to many open source projects, e.g. git/GitHub, the command line, and setting up a local development environment, contributing to this site may be a breeze or come with a bit of a learning curve. If you fit into the former group, great! Jump ahead to learn how to get started.

But if you think you're part of the second group, and have had trouble participating in open source because of a lack of comfort with the tools, **you're still welcome!** Beyond providing a resource for learning jQuery, a major goal of this site is to provide an encouraging environment for you to develop these skills, while still making a contribution that matters. Many people think that the only way to get involved with a programming project like jQuery is to solve intricate bugs that require a nuanced understanding of the codebase, or to propose enhancements that may or may not be in scope with the development team's plans. The fact is that there's way more: improving documentation, working on web properties, and supporting other users are crucial aspects where more help is always needed. If you're willing to share your time and expertise to help other developers, we're willing to [help you get up to speed with the tools](#) you'll need.

Why Contribute?

If you've ever looked for help with jQuery – or with web development in general – you know the hunt can sometimes be challenging. It can be a process of wading through a number of different posts until you find that article that's the right combination of trustworthy, timely, and helpful for your particular problem. And if you're one of those authors – thanks! – then you are probably familiar with the frustrating feeling of putting a useful tip out there, and then wondering if it's actually making its way to the people who need it, and what to do with that old post years and versions down the road. You're invited to share that energy to help us bring that ecosystem together and grow it further!

If you've ever helped anyone, colleague or stranger, with a particular problem, then you know the value of having a reference you can quickly link to that says "here's how you do it." This site is intended to be that compendium, but there's always more to refine and add, and we need your help too!

Of course, we'll also give you credit for your work! The **Contributors** section for each article is generated from the git commit logs on the file, so you'll be publicly acknowledged for your help.

How Does It Work?

Content

The content in this site is maintained in [this GitHub repository](#) as a collection of [Markdown](#) files in the `page` directory. The order in which chapters and articles are presented is controlled by the [order.json](#) file.

Design

The site's layout and design is controlled by our [jquery-wp-content](#), a custom [WordPress](#) configuration that runs (or will run in the near future) all of the sites run by the jQuery Foundation. The [master theme](#) controls most of the layout for all of our sites, and there is a [child theme](#) that controls the templates and styles specific to the learn site.

[jquery-wp-content](#) powers our sites in production and staging environments, and can be set up for local development relatively easily.

Build

The static content in the `page` directory is deployed to a [jquery-wp-content](#) instance using [grunt](#), specifically with two grunt plugins we've created:

- [grunt-jquery-content](#) – pre-processes content in a variety of formats (HTML, Markdown, XML) into HTML, applying syntax highlighting and some simple partial

- support, preparing it for processing by:
 - [grunt-wordpress](#) – syncs static content to WordPress using [XML-RPC](#)

How Can I Help?

The simplest and least complicated way to help is to [file issues](#) if you notice mistakes that should be fixed, improvements that can be made, or if you have ideas for new articles. We'll use the issues to continue discussion and track progress on anything you point out.

If you'd like to go a step further and contribute new articles, make edits to existing ones, or work on the site itself, the first thing you'll need is a [fork](#). When you have changes you'd like to have reviewed for integration into the site, submit a [pull request](#).

(If you're unfamiliar with Git, you can still contribute by using features in GitHub's web interface. You can edit files directly via [GitHub's in-browser editor](#). You can [create and delete branches directly from your fork](#), so you can also submit new articles as well. Either way, we still encourage you to [learn how to use Git and GitHub](#) as soon as you can.)

Local Development

In order to preview your changes locally, work on design/layout issues, or work on other jQuery sites' content, and generally contribute most effectively, we recommend that you set up a local development environment. You can learn how to get set up from our [documentation on contributing to jQuery Foundation web sites](#).

- **Windows note:** Line endings need to be Unix-style (line-feed only). Make sure your text editor creates new files with Unix-style line endings. In addition, the following setting to your git config will keep the Unix-style line endings when pulling from the repository:

```
$ git config --global core.autocrlf true
```

Working with Content

Once you've gotten your environment working, here are the general steps you should follow to make your changes:

1. Create a new "feature" branch based on master: `git branch <feature/issue name/number>`
2. Move onto that branch: `git checkout <feature/issue name/number>`
3. Work on your awesome contribution.
4. As you work and want to preview your changes, use `grunt` to deploy them to your site. You can also use `grunt watch` to have the site monitor the `page` directory for any changes and automatically have the changes deployed every time you save.
5. When you're done, stage the new/modified preparation for commit: `git add page/faq/how-do-i-add-a-new-article-to-the-learn-site.md`
6. Commit the files to your local repo: `git commit -m "add a relevant message describing the change"`
7. Push the files to your GitHub remote: `git push origin <feature/issue name/number>`
8. Go to your fork on GitHub and submit a new [pull request](#).

For more advice on managing your fork and submitting pull requests to the jQuery Foundation, read our [Commits and Pull Requests](#) guide.

Adding a New Article

1. Add the file to the right folder in the page folder.
2. Add the slug name (the filename without the extension) to the desired location in `order.json`
3. Run `grunt`
4. You should now be able to navigate to the file.

Formatting Articles

Yes! Take a look at our [style guide](#) for more information on authoring and formatting conventions.

Getting Help

If you're struggling to get any part of the site working properly, or have any questions, we're here to help.

The best place to get help is on [IRC](#), in the `#jquery-content` channel on [Freenode](#). If you're unfamiliar with IRC, you can use the [webchat gateway](#).

In addition, the jQuery Content Team holds a [public, biweekly meeting](#) on Wednesday, at 1PM Eastern time in the `#jquery-meeting` channel on Freenode.

If IRC is not your thing, but you still want or need to get in touch, please use the site's [GitHub repo](#) or send us an e-mail to `content at jquery dot org`.

About jQuery

Getting started with jQuery can be easy or challenging, depending on your experience with JavaScript, HTML, CSS, and programming concepts in general. In addition to these articles, you can read about the [history of jQuery](#) and the [licensing terms](#) that apply to jQuery projects. You can also [make a donation](#) to help the [jQuery team](#) continue to improve jQuery.

One important thing to know is that jQuery is just a **JavaScript library**. All the power of jQuery is accessed via JavaScript, so having a strong grasp of JavaScript is essential for understanding, structuring, and debugging your code. While working with jQuery regularly can, over time, improve your proficiency with JavaScript, it can be hard to get started writing jQuery without a working knowledge of JavaScript's built-in constructs and syntax. Therefore, if you're new to JavaScript, we recommend checking out the [JavaScript basics tutorial](#) on the Mozilla Developer Network (MDN).

About jQuery

How jQuery Works

jQuery: The Basics

This is a basic tutorial, designed to help you get started using jQuery. If you don't have a test page setup yet, start by creating the following HTML page:

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>Demo</title>
</head>
<body>
    <a href="http://jquery.com/">jQuery</a>
    <script src="jquery.js"></script>

    // Your code goes here.

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

The `src` attribute in the `<script>` element must point to a copy of jQuery. Download a copy of jQuery from the [Downloading jQuery](http://jquery.com/download/) page and store the `jquery.js` file in the same directory as your HTML file.

****Note**:** When you download jQuery, the file name may contain a version number, e.g., `jquery-x.y.z.js`. Make sure to either rename this file to `jquery.js` or update the `src` attribute of the `<script>` element to match the file name.

Launching Code on Document Ready

To ensure that their code runs after the browser finishes loading the document, many JavaScript programmers wrap their code in an `onload` function:

```
window.onload = function() {
    alert( "welcome" );
};
```

Unfortunately, the code doesn't run until all images are finished downloading, including banner ads. To run code as soon as the document is ready to be manipulated, jQuery has a statement known as the [ready event](#):

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

****Note**:** The jQuery library exposes its methods and properties via two properties of the `window` object called `jQuery` and `$`. `$` is simply an alias for `jQuery` and it's often employed because it's shorter and faster to write.

For example, inside the `ready` event, you can add a click handler to the link:

```
$( document ).ready(function() {
    $( "a" ).click(function( event ) {
        alert( "Thanks for visiting!" );
    });
});
```

Copy the above jQuery code into your HTML file where it says `// Your code goes here.` Then, save your HTML file and reload the test page in your browser. Clicking the link should now first display an alert pop-up, then continue with the default behavior of navigating to <http://jquery.com>.

For `click` and most other [events](#), you can prevent the default behavior by calling `event.preventDefault()` in the event handler:

```
$( document ).ready(function() {
    $( "a" ).click(function( event ) {
        alert( "As you can see, the link no longer took you to jquery.com"
    );
        event.preventDefault();
    });
});
```

Try replacing your first snippet of jQuery code, which you previously copied in to your HTML file, with the one above. Save the HTML file again and reload to try it out.

Complete Example

The following example illustrates the click handling code discussed above, embedded directly in the HTML `<body>`. Note that in practice, it is usually better to place your code in a separate JS file and load it on the page with a `<script>` element's `src` attribute.

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>Demo</title>
</head>
<body>
    <a href="http://jquery.com/">jQuery</a>
    <script src="jquery.js"></script>

    $( document ).ready(function() {
        $( "a" ).click(function( event ) {
            alert( "The link will no longer take you to jquery.com" );
            event.preventDefault();
        });
    });

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

Adding and Removing an HTML Class

****Important:**** You must place the remaining jQuery examples inside the ``ready`` event so that your code executes when the document is ready to be worked on.

Another common task is adding or removing a class.

First, add some style information into the `<head>` of the document, like this:

```
<style>
a.test {
    font-weight: bold;
}
</style>
```

Next, add the [.addClass\(\)](#) call to the script:

```
$( "a" ).addClass( "test" );
```

All `<a>` elements are now bold.

To remove an existing class, use [.removeClass\(\)](#):

```
$( "a" ).removeClass( "test" );
```

Special Effects

jQuery also provides some handy [effects](#) to help you make your web sites stand out. For example, if you create a click handler of:

```
$( "a" ).click(function( event ) {
    event.preventDefault();
```

```

        $( this ).hide( "slow" );
    });

```

Then the link slowly disappears when clicked.

Callbacks and Functions

Unlike many other programming languages, JavaScript enables you to freely pass functions around to be executed at a later time. A *callback* is a function that is passed as an argument to another function and is executed after its parent function has completed. Callbacks are special because they patiently wait to execute until their parent finishes. Meanwhile, the browser can be executing other functions or doing all sorts of other work.

To use callbacks, it is important to know how to pass them into their parent function.

Callback *without* Arguments

If a callback has no arguments, you can pass it in like this:

```
$.get( "myhtmlpage.html", myCallBack );
```

When [\\$.get\(\)](#) finishes getting the page myhtmlpage.html, it executes the myCallBack() function.

- **Note:** The second parameter here is simply the function name (but *not* as a string, and without parentheses).

Callback *with* Arguments

Executing callbacks with arguments can be tricky.

Wrong

This code example will *not* work:

```
$.get( "myhtmlpage.html", myCallBack( param1, param2 ) );
```

The reason this fails is that the code executes myCallBack(param1, param2) immediately and then passes myCallBack()'s *return value* as the second parameter to \$.get(). We actually want to pass the function myCallBack(), not myCallBack(param1, param2)'s return value (which might or might not be a function). So, how to pass in myCallBack() *and* include its arguments?

Right

To defer executing myCallBack() with its parameters, you can use an anonymous function as a wrapper. Note the use of function() {}. The anonymous function does exactly one thing: calls myCallBack(), with the values of param1 and param2.

```
$.get( "myhtmlpage.html", function() {
    myCallBack( param1, param2 );
});
```

When \$.get() finishes getting the page myhtmlpage.html, it executes the anonymous function, which executes myCallBack(param1, param2).

About jQuery

Additional jQuery Support

While we hope to cover most jQuery-related topics on this site, you may need additional or more immediate support. The following resources can prove useful.

Official Forums

<http://forum.jquery.com/>

There are many subforums where you can discuss jQuery, ask questions, talk about JavaScript, or announce your plugins.

- [Getting Started](#)
 - This is the best place to post if you are brand new to jQuery and JavaScript.
- [Using jQuery](#)
 - This is the best place to post if you have general questions or concerns.
 - If you've built a site that uses jQuery, or would like to announce a new plugin, this is the place to do it.
- [Using jQuery Plugins](#)
 - If you are a plugin author or user and you wish to discuss specific plugins, plugin bugs, new features, or new plugins.
- [Using jQuery UI](#)
 - This is the place to discuss use of [jQuery UI](#) Interactions, Widgets, and Effects
- [jQuery Mobile](#)
 - This is the place to discuss jQuery Mobile.
- [Developing jQuery Core](#)
 - This forum centers around development of the jQuery library itself.
 - Post here if you have questions about certain bugs, development with jQuery, features, or anything in the bug tracker or Git.
- [Developing jQuery Plugins](#)
 - This forum covers development of jQuery plugins.
- [Developing jQuery UI](#)
 - This is the place to discuss development of [jQuery UI](#) itself – including bugs, new plugins, and how you can help.
 - All jQuery UI svn commits are posted to this list to facilitate feedback, discussion, and review.
 - Also note that a lot of the development and planning of jQuery UI takes place on the [jQuery UI Development and Planning Wiki](#).
- [Developing jQuery Mobile](#)
 - This forum covers issues related to the development of jQuery Mobile.
- [QUnit and Testing](#)
 - This is the place to discuss JavaScript testing in general and QUnit in particular

At the bottom of each of the forums is an RSS feed you can subscribe to.

To ensure that you'll get a useful answer in no time, please consider the following advice:

- Ensure your markup is valid.
- Use Firebug/Developer Tools to see if you have an exception.
- Use Firebug/Developer Tools to inspect the HTML classes, CSS, etc.
- Try expected resulting HTML and CSS without JavaScript/jQuery and see if the problem could be isolated to those two.
- Reduce to a minimal test case (keep removing things until the problem goes away, etc.)
- Provide that test case as part of your mail. Either upload it somewhere or post it on jsbin.com.

In general, keep your question short and focused and provide only essential details – others can be added when required.

Mailing List Archives

The mailing lists existed before the forums were created, and were closed in early 2010.

There are two different ways of browsing the mailing list archives.

1. The official mailing list archives can be found here:

- [jQuery General Discussion Archives](#)
 - [jQuery Dev List Archives](#)
 - [jQuery UI General Discussion Archives](#)
- [jQuery UI Dev List Archives](#)
- [jQuery Plugins List Archives](#)

2. Also, an interactive, browsable version of the General Discussion mailing list can be found on [Nabble](#) (a forum-like mailing list mirror).

Chat / IRC Channel

jQuery also has a very active IRC channel, #jquery, hosted by [freenode](#).

The IRC Channel is best if you need quick help with any of the following:

- JavaScript
- jQuery syntax
- Problem solving
- Strange bugs

If your problem is more in-depth, we may ask you to post to the mailing list, or the bug tracker, so that we can help you in a more-suitable environment.

Connect info:

Server: irc.freenode.net

Room: #jquery

You can also connect at <http://webchat.freenode.net/?channels=#jquery>.

If you wish to post code snippets to the channel, you should use a paste site, like [jsfiddle.net](#) or [jsbin.com](#).

Additional info regarding jQuery's use of IRC can be found on irc.jquery.org.

StackOverflow

There is an active and well-informed support community at [StackOverflow](#). You can likely find an answer for whatever issue you're experiencing. If your question isn't addressed, you can ask a new question and often receive a quick response.

Using jQuery Core

Using jQuery Core

\$ vs \$()

Until now, we've been dealing entirely with methods that are called on a jQuery object. For example:

```
$( "h1" ).remove();
```

Most jQuery methods are called on jQuery objects as shown above; these methods are said to be part of the `$.fn` namespace, or the "jQuery prototype," and are best thought of as jQuery object methods.

However, there are several methods that do not act on a selection; these methods are said to be part of the jQuery namespace, and are best thought of as core jQuery methods.

This distinction can be incredibly confusing to new jQuery users. Here's what you need to remember:

- Methods called on jQuery selections are in the `$.fn` namespace, and automatically receive and return the selection as `this`.
- Methods in the `$` namespace are generally utility-type methods, and do not work with selections; they are not automatically passed any arguments, and their return value will vary.

There are a few cases where object methods and core methods have the same names, such as `$.each()` and `.each()`. In these cases, be extremely careful when reading the documentation that you are exploring the correct method.

In this guide, if a method can be called on a jQuery selection, we'll refer to it just by its name: `.each()`. If it is a [utility method](#) -- that is, a method that isn't called on a selection -- we'll refer to it explicitly as a method in the jQuery namespace: `$.each()`.

Using jQuery Core

\$(document).ready()

A page can't be manipulated safely until the document is "ready." jQuery detects this state of readiness for you. Code included inside `$(document).ready()` will only run once the page Document Object Model (DOM) is ready for JavaScript code to execute. Code included inside `$(window).on("load", function() { ... })` will run once the entire page (images or iframes), not just the DOM, is ready.

```
// A $( document ).ready() block.
$( document ).ready(function() {
    console.log( "ready!" );
});
```

Experienced developers sometimes use the shorthand `$()` for `$(document).ready()`. If you are writing code that people who aren't experienced with jQuery may see, it's best to use the long form.

```
// Shorthand for $( document ).ready()
$(function() {
    console.log( "ready!" );
});
```

You can also pass a named function to `$(document).ready()` instead of passing an anonymous function.

```
// Passing a named function instead of an anonymous function.

function readyFn( jQuery ) {
    // Code to run when the document is ready.
}

$( document ).ready( readyFn );
// or:
$( window ).on( "load", readyFn );
```

The example below shows `$(document).ready()` and `$(window).on("load")` in action. The code tries to load a website URL in an `<iframe>` and checks for both events:

```
<html>
<head>
    <script src="https://code.jquery.com/jquery-1.9.1.min.js"></script>

    $( document ).ready(function() {
        console.log( "document loaded" );
    });

    $( window ).on( "load", function() {
        console.log( "window loaded" );
    });
</script>
</head>
<body>
    <iframe src="http://techcrunch.com"></iframe>
</body>
</html>
```

Using jQuery Core

Avoiding Conflicts with Other Libraries

The jQuery library and virtually all of its plugins are contained within the `jQuery` namespace. As a general rule, global objects are stored inside the jQuery namespace as well, so you shouldn't get a clash between jQuery and any other library (like `prototype.js`, `MooTools`, or `YUI`).

That said, there is one caveat: *by default, jQuery uses `$` as a shortcut for `jQuery`*. Thus, if you are using another JavaScript library that uses the `$` variable, you can run into conflicts with jQuery. In order to avoid these conflicts, you need to put jQuery in no-conflict mode immediately after it is loaded onto the page and before you attempt to use jQuery in your page.

Putting jQuery Into No-Conflict Mode

When you put jQuery into no-conflict mode, you have the option of assigning a new variable name to replace the `$` alias.

```
<!-- Putting jQuery into no-conflict mode. -->
<script src="prototype.js"></script>
<script src="jquery.js"></script>

var $j = jQuery.noConflict();
// $j is now an alias to the jQuery function; creating the new alias is optional.

$j(document).ready(function() {
    $j( "div" ).hide();
});

// The $ variable now has the prototype meaning, which is a shortcut for
// document.getElementById(). mainDiv below is a DOM element, not a jQuery object.
window.onload = function() {
    var mainDiv = $( "main" );
}

</script>
```

In the code above, the `$` will revert back to its meaning in original library. You'll still be able to use the full function name `jQuery` as well as the new alias `$j` in the rest of your application. The new alias can be named anything you'd like: `jq`, `$J`, `awesomeQuery`, etc.

Finally, if you don't want to define another alternative to the full `jQuery` function name (you really like to use `$` and don't care about using the other library's `$` method), then there's still another approach you might try: simply add the `$` as an argument passed to your `jQuery(document).ready()` function. This is most frequently used in the case where you still want the benefits of really concise jQuery code, but don't want to cause conflicts with other libraries.

```
<!-- Another way to put jQuery into no-conflict mode. -->
<script src="prototype.js"></script>
<script src="jquery.js"></script>

jQuery.noConflict();

jQuery( document ).ready(function( $ ) {
    // You can use the locally-scoped $ in here as an alias to jQuery.
    $( "div" ).hide();
});

// The $ variable in the global scope has the prototype.js meaning.
window.onload = function(){
    var mainDiv = $( "main" );
}

</script>
```

This is probably the ideal solution for most of your code, considering that there'll be less code that you'll have to change in order to achieve complete compatibility.

Including jQuery Before Other Libraries

The code snippets above rely on jQuery being loaded after `prototype.js` is loaded. If you include jQuery before other libraries, you may use `jQuery` when you do some work with

jQuery, but the `$` will have the meaning defined in the other library. There is no need to relinquish the `$` alias by calling `jQuery.noConflict()`.

```
<!-- Loading jQuery before other libraries. -->
<script src="jquery.js"></script>
<script src="prototype.js"></script>

// Use full jQuery function name to reference jQuery.
jQuery( document ).ready(function() {
    jQuery( "div" ).hide();
});

// Use the $ variable as defined in prototype.js
window.onload = function() {
    var mainDiv = $( "main" );
};

</script>
```

Summary of Ways to Reference the jQuery Function

Here's a recap of ways you can reference the jQuery function when the presence of another library creates a conflict over the use of the `$` variable:

Create a New Alias

The `jQuery.noConflict()` method returns a reference to the jQuery function, so you can capture it in whatever variable you'd like:

```
<script src="prototype.js"></script>
<script src="jquery.js"></script>

// Give $ back to prototype.js; create new alias to jQuery.
var $jq = jQuery.noConflict();

</script>
```

Use an Immediately Invoked Function Expression

You can continue to use the standard `$` by wrapping your code in an immediately invoked function expression; this is also a standard pattern for jQuery plugin authoring, where the author cannot know whether another library will have taken over the `$`. See the [Plugins](#) section for more information about writing plugins.

```
<!-- Using the $ inside an immediately-invoked function expression. -->
<script src="prototype.js"></script>
<script src="jquery.js"></script>

jQuery.noConflict();

(function( $ ) {
    // Your jQuery code here, using the $
})( jQuery );

</script>
```

Note that if you use this technique, you will not be able to use `prototype.js` methods inside the immediately invoked function. `$` will be a reference to jQuery, not `prototype.js`.

Use the Argument That's Passed to the `jQuery(document).ready()` Function

```
<script src="jquery.js"></script>
<script src="prototype.js"></script>

jQuery(document).ready(function( $ ) {
    // Your jQuery code here, using $ to refer to jQuery.
});

</script>
```

Or using the more concise syntax for the DOM ready function:

```
<script src="jquery.js"></script>
<script src="prototype.js"></script>
```

```
jQuery(function($){  
    // Your jQuery code here, using the $  
});  
</script>
```

Using jQuery Core

Attributes

An element's attributes can contain useful information for your application, so it's important to be able to get and set them.

The `.attr()` method

The `.attr()` method acts as both a getter and a setter. As a setter, `.attr()` can accept either a key and a value, or an object containing one or more key/value pairs.

`.attr()` as a setter:

```
$( "a" ).attr( "href", "allMyHrefsAreTheSameNow.html" );

$( "a" ).attr({
  title: "all titles are the same too!",
  href: "somethingNew.html"
});
```

`.attr()` as a getter:

```
$( "a" ).attr( "href" ); // Returns the href for the first a element in the document
```

Using jQuery Core

Selecting Elements

The most basic concept of jQuery is to "select some elements and do something with them." jQuery supports most CSS3 selectors, as well as some non-standard selectors. For a complete selector reference, visit the [Selectors documentation on api.jquery.com](http://api.jquery.com/selectors).

Selecting Elements by ID

```
$( "#myId" ); // Note IDs must be unique per page.
```

Selecting Elements by Class Name

```
$( ".myClass" );
```

Selecting Elements by Attribute

```
$( "input[name='first_name']" );
```

Selecting Elements by Compound CSS Selector

```
$( "#contents ul.people li" );
```

Selecting Elements with a Comma-separated List of Selectors

```
$( "div.myClass, ul.people" );
```

Pseudo-Selectors

```
$( "a.external:first" );
$( "tr:odd" );

// Select all input-like elements in a form (more on this below).
$( "#myForm :input" );
$( "div:visible" );

// All except the first three divs.
$( "div:gt(2)" );

// All currently animated divs.
$( "div:animated" );
```

Note: When using the `:visible` and `:hidden` pseudo-selectors, jQuery tests the actual visibility of the element, not its CSS `visibility` or `display` properties. jQuery looks to see if the element's physical height and width on the page are both greater than zero.

However, this test doesn't work with `<tr>` elements. In the case of `<tr>` jQuery does check the CSS `display` property, and considers an element hidden if its `display` property is set to `none`.

Elements that have not been added to the DOM will always be considered hidden, even if the CSS that would affect them would render them visible. See the [Manipulating Elements](#) section to learn how to create and add elements to the DOM.

Choosing Selectors

Choosing good selectors is one way to improve JavaScript's performance. Too much specificity can be a bad thing. A selector such as `#myTable thead tr th.special` is overkill if a selector such as `#myTable th.special` will get the job done.

Does My Selection Contain Any Elements?

Once you've made a selection, you'll often want to know whether you have anything to work with. A common mistake is to use:

```
// Doesn't work!
if ( $( "div.foo" ) ) {
```

```
    ...
}
```

This won't work. When a selection is made using `$()`, an object is always returned, and objects always evaluate to `true`. Even if the selection doesn't contain any elements, the code inside the `if` statement will still run.

The best way to determine if there are any elements is to test the selection's `.length` property, which tells you how many elements were selected. If the answer is 0, the `.length` property will evaluate to `false` when used as a boolean value:

```
// Testing whether a selection contains elements.
if ( $( "div.foo" ).length ) {
    ...
}
```

Saving Selections

jQuery doesn't cache elements for you. If you've made a selection that you might need to make again, you should save the selection in a variable rather than making the selection repeatedly.

```
var divs = $( "div" );
```

Once the selection is stored in a variable, you can call jQuery methods on the variable just like you would have called them on the original selection.

A selection only fetches the elements that are on the page at the time the selection is made. If elements are added to the page later, you'll have to repeat the selection or otherwise add them to the selection stored in the variable. Stored selections don't magically update when the DOM changes.

Refining & Filtering Selections

Sometimes the selection contains more than what you're after. jQuery offers several methods for refining and filtering selections.

```
// Refining selections.
$( "div.foo" ).has( "p" );           // div.foo elements that contain <p> tags
$( "h1" ).not( ".bar" );           // h1 elements that don't have a class of bar
$( "ul li" ).filter( ".current" ); // unordered list items with class of current
$( "ul li" ).first();              // just the first unordered list item
$( "ul li" ).eq( 5 );              // the sixth
```

Selecting Form Elements

jQuery offers several pseudo-selectors that help find elements in forms. These are especially helpful because it can be difficult to distinguish between form elements based on their state or type using standard CSS selectors.

:checked

Not to be confused with `:checkbox`, `:checked` targets *checked* checkboxes, but keep in mind that this selector works also for *checked* radio buttons, and `<select>` elements (for `<select>` elements only, use the `:selected` selector):

```
$( "form :checked" );
```

The `:checked` pseudo-selector works when used with **checkboxes**, **radio buttons** and **selects**.

:disabled

Using the `:disabled` pseudo-selector targets any `<input>` elements with the `disabled` attribute:

```
$( "form :disabled" );
```

In order to get the best performance using `:disabled`, first select elements with a standard jQuery selector, then use `.filter(":disabled")`, or precede the pseudo-selector with a tag name or some other selector.

:enabled

Basically the inverse of the *:disabled* pseudo-selector, the *:enabled* pseudo-selector targets any elements that *do not* have a *disabled* attribute:

```
$( "form :enabled" );
```

In order to get the best performance using *:enabled*, first select elements with a standard jQuery selector, then use *.filter(":enabled")*, or precede the pseudo-selector with a tag name or some other selector.

:input

Using the *:input* selector selects all *<input>*, *<textarea>*, *<select>*, and *<button>* elements:

```
$( "form :input" );
```

:selected

Using the *:selected* pseudo-selector targets any selected items in *<option>* elements:

```
$( "form :selected" );
```

In order to get the best performance using *:selected*, first select elements with a standard jQuery selector, then use *.filter(":selected")*, or precede the pseudo-selector with a tag name or some other selector.

Selecting by type

jQuery provides pseudo selectors to select form-specific elements according to their type:

- [:password](#)
- [:reset](#)
- [:radio](#)
- [:text](#)
- [:submit](#)
- [:checkbox](#)
- [:button](#)
- [:image](#)
- [:file](#)

For all of these there are side notes about performance, so be sure to check out [the API docs](#) for more in-depth information.

Using jQuery Core

Working with Selections

Getters & Setters

Some jQuery methods can be used to either assign or read some value on a selection. When the method is called with a value as an argument, it's referred to as a setter because it sets (or assigns) that value. When the method is called with no argument, it gets (or reads) the value of the element. Setters affect all elements in a selection, whereas getters return the requested value only for the first element in the selection, with the exception of [.text\(\)](#), which retrieves the values of all the elements.

```
// The .html() method sets all the h1 elements' html to be "hello world":
$( "h1" ).html( "hello world" );

// The .html() method returns the html of the first h1 element:
$( "h1" ).html();
// > "hello world"
```

Setters return a jQuery object, allowing you to continue calling jQuery methods on your selection. Getters return whatever they were asked to get, so you can't continue to call jQuery methods on the value returned by the getter.

```
// Attempting to call a jQuery method after calling a getter.
// This will NOT work:
$( "h1" ).html().addClass( "test" );
```

Chaining

If you call a method on a selection and that method returns a jQuery object, you can continue to call jQuery methods on the object without pausing for a semicolon. This practice is referred to as "chaining":

```
$( "#content" ).find( "h3" ).eq( 2 ).html( "new text for the third h3!" );
```

It may help code readability to break the chain over several lines:

```
$( "#content" )
    .find( "h3" )
    .eq( 2 )
    .html( "new text for the third h3!" );
```

jQuery also provides the `.end()` method to get back to the original selection should you change the selection in the middle of a chain:

```
$( "#content" )
    .find( "h3" )
    .eq( 2 )
    .html( "new text for the third h3!" )
    .end() // Restores the selection to all h3s in #content
    .eq( 0 )
    .html( "new text for the first h3!" );
```

Chaining is extraordinarily powerful, and it's a feature that many libraries have adapted since it was made popular by jQuery. However, it must be used with care – extensive chaining can make code extremely difficult to modify or debug. There is no hard-and-fast rule to how long a chain should be – just know that it's easy to get carried away.

Using jQuery Core

Manipulating Elements

For complete documentation of jQuery manipulation methods, visit the [Manipulation documentation on api.jquery.com](http://api.jquery.com/manipulation).

Getting and Setting Information About Elements

There are many ways to change an existing element. Among the most common tasks is changing the inner HTML or attribute of an element. jQuery offers simple, cross-browser methods for these sorts of manipulations. You can also get information about elements using many of the same methods in their getter incarnations. For more information on getters and setters, see the [Working with Selections](#) section. Here are a few methods you can use to get and set information about elements:

- `.html()` – Get or set the HTML contents.
- `.text()` – Get or set the text contents; HTML will be stripped.
- `.attr()` – Get or set the value of the provided attribute.
- `.width()` – Get or set the width in pixels of the first element in the selection as an integer.
- `.height()` – Get or set the height in pixels of the first element in the selection as an integer.
- `.position()` – Get an object with position information for the first element in the selection, relative to its first positioned ancestor. *This is a getter only.*
- `.val()` – Get or set the value of form elements.

Changing things about elements is trivial, but remember that the change will affect all elements in the selection. If you just want to change one element, be sure to specify that in the selection before calling a setter method.

```
// Changing the HTML of an element.
$( "#myDiv p:first" ).html( "New <strong>first</strong> paragraph!" );
```

Moving, Copying, and Removing Elements

While there are a variety of ways to move elements around the DOM, there are generally two approaches:

- Place the selected element(s) relative to another element.
- Place an element relative to the selected element(s).

For example, jQuery provides `.insertAfter()` and `.after()`. The `.insertAfter()` method places the selected element(s) after the element provided as an argument. The `.after()` method places the element provided as an argument after the selected element. Several other methods follow this pattern: `.insertBefore()` and `.before()`, `.appendTo()` and `.append()`, and `.prependTo()` and `.prepend()`.

The method that makes the most sense will depend on what elements are selected, and whether you need to store a reference to the elements you're adding to the page. If you need to store a reference, you will always want to take the first approach – placing the selected elements relative to another element – as it returns the element(s) you're placing. In this case, `.insertAfter()`, `.insertBefore()`, `.appendTo()`, and `.prependTo()` should be the tools of choice.

```
// Moving elements using different approaches.

// Make the first list item the last list item:
var li = $( "#myList li:first" ).appendTo( "#myList" );

// Another approach to the same problem:
$( "#myList" ).append( $( "#myList li:first" ) );

// Note that there's no way to access the list item
// that we moved, as this returns the list itself.
```

Cloning Elements

Methods such as `.appendTo()` move the element, but sometimes a copy of the element is needed instead. In this case, use `.clone()` first:

```
// Making a copy of an element.

// Copy the first list item to the end of the list:
$( "#myList li:first" ).clone().appendTo( "#myList" );
```

If you need to copy related data and events, be sure to pass `true` as an argument to `.clone()`.

Removing Elements

There are two ways to remove elements from the page: `.remove()` and `.detach()`. Use `.remove()` when you want to permanently remove the selection from the page. While `.remove()` does return the removed element(s), those elements will not have their associated data and events attached to them if you return them to the page.

Use `.detach()` if you need the data and events to persist. Like `.remove()`, it returns the selection, but it also maintains the data and events associated with the selection, so you can restore the selection to the page at a later time.

The `.detach()` method is extremely valuable if you are doing heavy manipulation on an element. In that case, it's beneficial to `.detach()` the element from the page, work on it in your code, then restore it to the page when you're done. This limits expensive "DOM touches" while maintaining the element's data and events.

If you want to leave the element on the page but remove its contents, you can use `.empty()` to dispose of the element's inner HTML.

Creating New Elements

jQuery offers a trivial and elegant way to create new elements using the same `$()` method used to make selections:

```
// Creating new elements from an HTML string.
$( "<p>This is a new paragraph</p>" );
$( "<li class='new'>new list item</li>" );

// Creating a new element with an attribute object.
$( "<a/>", {
    html: "This is a <strong>new</strong> link",
    "class": "new",
    href: "foo.html"
});
```

Note that the attributes object in the second argument above, the property name `class` is quoted, although the property names `html` and `href` are not. Property names generally do not need to be quoted unless they are [reserved words](#) (as `class` is in this case).

When you create a new element, it is not immediately added to the page. There are several ways to add an element to the page once it's been created.

```
// Getting a new element on to the page.

var myNewElement = $( "<p>New element</p>" );

myNewElement.appendTo( "#content" );

myNewElement.insertAfter( "ul:last" ); // This will remove the p from #content!

$( "ul" ).last().after( myNewElement.clone() ); // Clone the p so now we have two.
```

The created element doesn't need to be stored in a variable – you can call the method to add the element to the page directly after the `$()`. However, most of the time you'll want a reference to the element you added so you won't have to select it later.

You can also create an element as you're adding it to the page, but note that in this case you don't get a reference to the newly created element:

```
// Creating and adding an element to the page at the same time.
$( "ul" ).append( "<li>list item</li>" );
```

The syntax for adding new elements to the page is easy, so it's tempting to forget that there's a huge performance cost for adding to the DOM repeatedly. If you're adding many elements to

the same container, you'll want to concatenate all the HTML into a single string, and then append that string to the container instead of appending the elements one at a time. Use an array to gather all the pieces together, then join them into a single string for appending:

```
var myItems = [];
var myList = $( "#myList" );

for ( var i = 0; i < 100; i++ ) {
    myItems.push( "<li>item " + i + "</li>" );
}

myList.append( myItems.join( "" ) );
```

Manipulating Attributes

jQuery's attribute manipulation capabilities are extensive. Basic changes are simple, but the `.attr()` method also allows for more complex manipulations. It can either set an explicit value, or set a value using the return value of a function. When the function syntax is used, the function receives two arguments: the zero-based index of the element whose attribute is being changed, and the current value of the attribute being changed.

```
// Manipulating a single attribute.
$( "#myDiv a:first" ).attr( "href", "newDestination.html" );

// Manipulating multiple attributes.
$( "#myDiv a:first" ).attr({
    href: "newDestination.html",
    rel: "nofollow"
});

// Using a function to determine an attribute's new value.
$( "#myDiv a:first" ).attr({
    rel: "nofollow",
    href: function( idx, href ) {
        return "/new/" + href;
    }
});

$( "#myDiv a:first" ).attr( "href", function( idx, href ) {
    return "/new/" + href;
});
```

Using jQuery Core

The jQuery Object

When creating new elements (or selecting existing ones), jQuery returns the elements in a collection. Many developers new to jQuery assume that this collection is an array. It has a zero-indexed sequence of DOM elements, some familiar array functions, and a `.length` property, after all. Actually, the jQuery object is more complicated than that.

DOM and DOM Elements

The Document Object Model (DOM for short) is a representation of an HTML document. It may contain any number of DOM elements. At a high level, a DOM element can be thought of as a "piece" of a web page. It may contain text and/or other DOM elements. DOM elements are described by a type, such as `<div>`, `<a>`, or `<p>`, and any number of attributes such as `src`, `href`, `class` and so on. For a more thorough description, refer to [the official DOM specification from the W3C](#).

Elements have properties like any JavaScript object. Among these properties are attributes like `.tagName` and methods like `.appendChild()`. These properties are the only way to interact with the web page via JavaScript.

The jQuery Object

It turns out that working directly with DOM elements can be awkward. The jQuery object defines [many](#) methods to smooth out the experience for developers. Some benefits of the jQuery Object include:

Compatibility – The implementation of element methods varies across browser vendors and versions. The following snippet attempts to set the inner HTML of a `<tr>` element stored in `target`:

```
var target = document.getElementById( "target" );
target.innerHTML = "<td>Hello <b>World</b>!</td>";
```

This works in many cases, but it will fail in most versions of Internet Explorer. In that case, the [recommended approach](#) is to use pure DOM methods instead. By wrapping the `target` element in a jQuery object, these edge cases are taken care of, and the expected result is achieved in all supported browsers:

```
// Setting the inner HTML with jQuery.
var target = document.getElementById( "target" );
$( target ).html( "<td>Hello <b>World</b>!</td>" );
```

Convenience – There are also a lot of common DOM manipulation use cases that are awkward to accomplish with pure DOM methods. For instance, inserting an element stored in `newElement` after the `target` element requires a rather verbose DOM method:

```
// Inserting a new element after another with the native DOM API.
var target = document.getElementById( "target" );
var newElement = document.createElement( "div" );
target.parentNode.insertBefore( newElement, target.nextSibling );
```

By wrapping the `target` element in a jQuery object, the same task becomes much simpler:

```
// Inserting a new element after another with jQuery.
var target = document.getElementById( "target" );
var newElement = document.createElement( "div" );
$( target ).after( newElement );
```

For the most part, these details are simply "gotchas" standing between you and your goals.

Getting Elements Into the jQuery Object

When the jQuery function is invoked with a CSS selector, it will return a jQuery object wrapping any element(s) that match this selector. For instance, writing:

```
// Selecting all <h1> tags.
var headings = $( "h1" );

headings is now a jQuery element containing all the <h1> tags already on the page. This can
be verified by inspecting the .length property of headings:

// Viewing the number of <h1> tags on the page.
var headings = $( "h1" );
alert( headings.length );
```

If the page has more than one <h1> tag, this number will be greater than one. If the page has no <h1> tags, the .length property will be zero. Checking the .length property is a common way to ensure that the selector successfully matched one or more elements.

If the goal is to select only the first heading element, another step is required. There are a number of ways to accomplish this, but the most straight-forward is the .eq() function.

```
// Selecting only the first <h1> element on the page (in a jQuery object)
var headings = $( "h1" );
var firstHeading = headings.eq( 0 );
```

Now firstHeading is a jQuery object containing only the first <h1> element on the page. And because firstHeading is a jQuery object, it has useful methods like .html() and .after(). jQuery also has a method named .get() which provides a related function. Instead of returning a jQuery-wrapped DOM element, it returns the DOM element itself.

```
// Selecting only the first <h1> element on the page.
var firstHeadingElem = $( "h1" ).get( 0 );
```

Alternatively, because the jQuery object is "array-like," it supports array subscripting via brackets:

```
// Selecting only the first <h1> element on the page (alternate approach).
var firstHeadingElem = $( "h1" )[ 0 ];
```

In either case, firstHeadingElem contains the native DOM element. This means it has DOM properties like .innerHTML and methods like .appendChild(), but *not* jQuery methods like .html() or .after(). The firstHeadingElem element is more difficult to work with, but there are certain instances that require it. One such instance is making comparisons.

Not All jQuery Objects are Created ===

An important detail regarding this "wrapping" behavior is that each wrapped object is unique. This is true *even if the object was created with the same selector or contain references to the exact same DOM elements*.

```
// Creating two jQuery objects for the same element.
var logo1 = $( "#logo" );
var logo2 = $( "#logo" );
```

Although logo1 and logo2 are created in the same way (and wrap the same DOM element), they are not the same object. For example:

```
// Comparing jQuery objects.
alert( $( "#logo" ) === $( "#logo" ) ); // alerts "false"
```

However, both objects contain the same DOM element. The .get() method is useful for testing if two jQuery objects have the same DOM element.

```
// Comparing DOM elements.
var logo1 = $( "#logo" );
var logo1Elem = logo1.get( 0 );
var logo2 = $( "#logo" );
```

```
var logo2Elem = logo2.get( 0 );
alert( logo1Elem === logo2Elem ); // alerts "true"
```

Many developers prefix a \$ to the name of variables that contain jQuery objects in order to help differentiate. There is nothing magic about this practice – it just helps some people keep track of what different variables contain. The previous example could be re-written to follow this convention:

```
// Comparing DOM elements (with more readable variable names).
var $logo1 = $( "#logo" );
var logo1 = $logo1.get( 0 );

var $logo2 = $( "#logo" );
var logo2 = $logo2.get( 0 );

alert( logo1 === logo2 ); // alerts "true"
```

This code functions identically to the example above, but it is a little more clear to read.

Regardless of the naming convention used, it is very important to make the distinction between jQuery object and native DOM elements. Native DOM methods and properties are not present on the jQuery object, and vice versa. Error messages like "event.target.close is not a function" and "TypeError: Object [object Object] has no method 'setAttribute'" indicate the presence of this common mistake.

jQuery Objects Are Not "Live"

Given a jQuery object with all the paragraph elements on the page:

```
// Selecting all <p> elements on the page.
var allParagraphs = $( "p" );
```

...one might expect that the contents will grow and shrink over time as <p> elements are added and removed from the document. jQuery objects do **not** behave in this manner. The set of elements contained within a jQuery object will not change unless explicitly modified. This means that the collection is not "live" – it does not automatically update as the document changes. If the document may have changed since the creation of the jQuery object, the collection should be updated by creating a new one. It can be as easy as re-running the same selector:

```
// Updating the selection.
allParagraphs = $( "p" );
```

Wrapping Up

Although DOM elements provide all the functionality one needs to create interactive web pages, they can be a hassle to work with. The jQuery object wraps these elements to smooth out this experience and make common tasks easy. When creating or selecting elements with jQuery, the result will always be wrapped in a new jQuery object. If the situation calls for the native DOM elements, they may be accessed through the `.get()` method and/or array-style subscripting.

Using jQuery Core

Traversing

Once you've made an initial selection with jQuery, you can traverse deeper into what was just selected. Traversing can be broken down into three basic parts: parents, children, and siblings. jQuery has an abundance of easy-to-use methods for all these parts. Notice that each of these methods can optionally be passed string selectors, and some can also take another jQuery object in order to filter your selection down. Pay attention and refer to the [API documentation on traversing](#) to know what variation of arguments you have available.

Parents

The methods for finding the parents from a selection include `.parent()`, `.parents()`, `.parentsUntil()`, and `.closest()`.

```
<div class="grandparent">
  <div class="parent">
    <div class="child">
      <span class="subchild"></span>
    </div>
  </div>
  <div class="surrogateParent1"></div>
  <div class="surrogateParent2"></div>
</div>

// Selecting an element's direct parent:

// returns [ div.child ]
$( "span.subchild" ).parent();

// Selecting all the parents of an element that match a given selector:

// returns [ div.parent ]
$( "span.subchild" ).parents( "div.parent" );

// returns [ div.child, div.parent, div.grandparent ]
$( "span.subchild" ).parents();

// Selecting all the parents of an element up to, but *not including* the selector:

// returns [ div.child, div.parent ]
$( "span.subchild" ).parentsUntil( "div.grandparent" );

// Selecting the closest parent, note that only one parent will be selected
// and that the initial element itself is included in the search:

// returns [ div.child ]
$( "span.subchild" ).closest( "div" );

// returns [ div.child ] as the selector is also included in the search:
$( "div.child" ).closest( "div" );
```

Children

The methods for finding child elements from a selection include `.children()` and `.find()`. The difference between these methods lies in how far into the child structure the selection is made. `.children()` only operates on direct child nodes, while `.find()` can traverse recursively into children, children of those children, and so on.

```
// Selecting an element's direct children:

// returns [ div.parent, div.surrogateParent1, div.surrogateParent2 ]
$( "div.grandparent" ).children( "div" );

// Finding all elements within a selection that match the selector:

// returns [ div.child, div.parent, div.surrogateParent1, div.surrogateParent2 ]
$( "div.grandparent" ).find( "div" );
```

Siblings

The rest of the traversal methods within jQuery all deal with finding sibling selections. There are a few basic methods as far as the direction of traversal is concerned. You can find previous elements with `.prev()`, next elements with `.next()`, and both with `.siblings()`. There are also a few other methods that build onto these basic methods: `.nextAll()`, `.nextUntil()`, `.prevAll()` and `.prevUntil()`.

```

// Selecting a next sibling of the selectors:

// returns [ div.surrogateParent1 ]
$( "div.parent" ).next();

// Selecting a prev sibling of the selectors:

// returns [] as No sibling exists before div.parent
$( "div.parent" ).prev();

// Selecting all the next siblings of the selector:

// returns [ div.surrogateParent1, div.surrogateParent2 ]
$( "div.parent" ).nextAll();

// returns [ div.surrogateParent1 ]
$( "div.parent" ).nextAll().first();

// returns [ div.surrogateParent2 ]
$( "div.parent" ).nextAll().last();

// Selecting all the previous siblings of the selector:

// returns [ div.surrogateParent1, div.parent ]
$( "div.surrogateParent2" ).prevAll();

// returns [ div.surrogateParent1 ]
$( "div.surrogateParent2" ).prevAll().first();

// returns [ div.parent ]
$( "div.surrogateParent2" ).prevAll().last();

```

Use `.siblings()` to select all siblings:

```

// Selecting an element's siblings in both directions that matches the given
selector:

// returns [ div.surrogateParent1, div.surrogateParent2 ]
$( "div.parent" ).siblings();

// returns [ div.parent, div.surrogateParent2 ]
$( "div.surrogateParent1" ).siblings();

```

See the complete documentation for these methods and more at [Traversal documentation on api.jquery.com](http://api.jquery.com).

Be cautious when traversing long distances in documents – complex traversal makes it imperative that the document's structure remain the same, which is difficult to guarantee even if you're the one creating the whole application from server to client. One- or two-step traversal is fine, but it's best to avoid traversals that go from one container to another.

Using jQuery Core

CSS, Styling, & Dimensions

jQuery includes a handy way to get and set CSS properties of elements:

```
// Getting CSS properties.

$( "h1" ).css( "fontSize" ); // Returns a string such as "19px".

$( "h1" ).css( "font-size" ); // Also works.

// Setting CSS properties.

$( "h1" ).css( "fontSize", "100px" ); // Setting an individual property.

// Setting multiple properties.
$( "h1" ).css({
    fontSize: "100px",
    color: "red"
});
```

Note the style of the argument on the second line – it is an object that contains multiple properties. This is a common way to pass multiple arguments to a function, and many jQuery setter methods accept objects to set multiple values at once.

CSS properties that normally include a hyphen need to be camelCased in JavaScript. For example, the CSS property `font-size` is expressed as `fontSize` when used as a property name in JavaScript. However, this does not apply when passing the name of a CSS property to the `.css()` method as a string – in that case, either the camelCased or hyphenated form will work.

It's not recommended to use `.css()` as a setter in production-ready code, but when passing in an object to set CSS, CSS properties will be camelCased instead of using a hyphen.

Using CSS Classes for Styling

As a getter, the `.css()` method is valuable. However, it should generally be avoided as a setter in production-ready code, because it's generally best to keep presentational information out of JavaScript code. Instead, write CSS rules for classes that describe the various visual states, and then change the class on the element.

```
// Working with classes.

var h1 = $( "h1" );

h1.addClass( "big" );
h1.removeClass( "big" );
h1.toggleClass( "big" );

if ( h1.hasClass( "big" ) ) {
    ...
}
```

Classes can also be useful for storing state information about an element, such as indicating that an element is selected.

Dimensions

jQuery offers a variety of methods for obtaining and modifying dimension and position information about an element.

The code below shows a brief overview of the dimensions functionality in jQuery. For complete details about jQuery dimension methods, visit the [dimensions documentation on api.jquery.com](http://api.jquery.com).

```
// Basic dimensions methods.

// Sets the width of all <h1> elements.
$( "h1" ).width( "50px" );

// Gets the width of the first <h1> element.
$( "h1" ).width();

// Sets the height of all <h1> elements.
$( "h1" ).height( "50px" );
```

```
// Gets the height of the first <h1> element.  
$( "h1" ).height();  
  
// Returns an object containing position information for  
// the first <h1> relative to its "offset (positioned) parent".  
$( "h1" ).position();
```

Using jQuery Core

Data Methods

There's often data about an element you want to store with the element. In plain JavaScript, you might do this by adding a property to the DOM element, but you'd have to deal with memory leaks in some browsers. jQuery offers a straightforward way to store data related to an element, and it manages the memory issues for you.

```
// Storing and retrieving data related to an element.

$( "#myDiv" ).data( "keyName", { foo: "bar" } );

$( "#myDiv" ).data( "keyName" ); // Returns { foo: "bar" }
```

Any kind of data can be stored on an element. For the purposes of this article, `.data()` will be used to store references to other elements.

For example, you may want to establish a relationship between a list item and a `<div>` that's inside of it. This relationship could be established every single time the list item is touched, but a better solution would be to establish the relationship once, then store a pointer to the `<div>` on the list item using `.data()`:

```
// Storing a relationship between elements using .data()

$( "#myList li" ).each(function() {

    var li = $( this );
    var div = li.find( "div.content" );

    li.data( "contentDiv", div );

});

// Later, we don't have to find the div again;
// we can just read it from the list item's data
var firstLi = $( "#myList li:first" );

firstLi.data( "contentDiv" ).html( "new content" );
```

In addition to passing `.data()` a single key-value pair to store data, you can also pass an object containing one or more pairs.

Using jQuery Core

Utility Methods

jQuery offers several utility methods in the `$` namespace. These methods are helpful for accomplishing routine programming tasks. For a complete reference on jQuery utility methods, visit the [utilities documentation on api.jquery.com](https://api.jquery.com/utilities-namespace/).

Below are examples of a few of the utility methods:

`$.trim()`

Removes leading and trailing whitespace:

```
// Returns "lots of extra whitespace"
$.trim( "   lots of extra whitespace   " );
```

`$.each()`

Iterates over arrays and objects:

```
$.each([ "foo", "bar", "baz" ], function( idx, val ) {
    console.log( "element " + idx + " is " + val );
});

$.each({ foo: "bar", baz: "bim" }, function( k, v ) {
    console.log( k + " : " + v );
});
```

The method `.each()` can be called on a selection to iterate over the elements contained in the selection. `.each()`, not `$.each()`, should be used for iterating over elements in a selection.

`$.inArray()`

Returns a value's index in an array, or -1 if the value is not in the array:

```
var myArray = [ 1, 2, 3, 5 ];

if ( $.inArray( 4, myArray ) !== -1 ) {
    console.log( "found it!" );
}
```

`$.extend()`

Changes the properties of the first object using the properties of subsequent objects:

```
var firstObject = { foo: "bar", a: "b" };
var secondObject = { foo: "baz" };

var newObject = $.extend( firstObject, secondObject );

console.log( firstObject.foo ); // "baz"
console.log( newObject.foo ); // "baz"
```

If you don't want to change any of the objects you pass to `$.extend()`, pass an empty object as the first argument:

```
var firstObject = { foo: "bar", a: "b" };
var secondObject = { foo: "baz" };

var newObject = $.extend( {}, firstObject, secondObject );

console.log( firstObject.foo ); // "bar"
console.log( newObject.foo ); // "baz"
```

`$.proxy()`

Returns a function that will always run in the provided scope — that is, sets the meaning of `this` inside the passed function to the second argument.

```
var myFunction = function() {
    console.log( this );
};

var myObject = {
    foo: "bar"
};
```

```
myFunction(); // window

var myProxyFunction = $.proxy( myFunction, myObject );

myProxyFunction(); // myObject
```

If you have an object with methods, you can pass the object and the name of a method to return a function that will always run in the scope of the object.

```
var myObject = {
  myFn: function() {
    console.log( this );
  }
};

$( "#foo" ).click( myObject.myFn ); // HTMLElement #foo
$( "#foo" ).click( $.proxy( myObject, "myFn" ) ); // myObject
```

Testing Type

Sometimes the `typeof` operator [can be confusing or inconsistent](#), so instead of using `typeof`, jQuery offers utility methods to help determine the type of a value.

First of all, you have methods to test if a specific value is of a specific type.

```
$.isArray([]); // true
$.isFunction(function() {}); // true
$.isNumeric(3.14); // true
```

Additionally, there is `$.type()` which checks for the internal class used to create a value. You can see the method as a better alternative for the `typeof` operator.

```
$.type( true ); // "boolean"
$.type( 3 ); // "number"
$.type( "test" ); // "string"
$.type( function() {} ); // "function"

$.type( new Boolean() ); // "boolean"
$.type( new Number(3) ); // "number"
$.type( new String('test') ); // "string"
$.type( new Function() ); // "function"

$.type( [] ); // "array"
$.type( null ); // "null"
$.type( /test/ ); // "regexp"
$.type( new Date() ); // "date"
```

As always, you can check the [API docs](#) for a more in-depth explanation.

Using jQuery Core

Iterating over jQuery and non-jQuery Objects

jQuery provides an object iterator utility called `$.each()` as well as a jQuery collection iterator: `.each()`. These are not interchangeable. In addition, there are a couple of helpful methods called `$.map()` and `.map()` that can shortcut one of our common iteration use cases.

`$.each()`

[`\$.each\(\)`](#) is a generic iterator function for looping over object, arrays, and array-like objects. Plain objects are iterated via their named properties while arrays and array-like objects are iterated via their indices.

`$.each()` is essentially a drop-in replacement of a traditional `for` or `for-in` loop. Given:

```
var sum = 0;

var arr = [ 1, 2, 3, 4, 5 ];
```

Then this:

```
for ( var i = 0, l = arr.length; i < l; i++ ) {
    sum += arr[ i ];
}

console.log( sum ); // 15
```

Can be replaced with this:

```
$.each( arr, function( index, value ){
    sum += value;
});

console.log( sum ); // 15
```

Notice that we don't have to access `arr[index]` as the value is conveniently passed to the callback in `$.each()`.

In addition, given:

```
var sum = 0;
var obj = {
    foo: 1,
    bar: 2
}
```

Then this:

```
for (var item in obj) {
    sum += obj[ item ];
}

console.log( sum ); // 3
```

Can be replaced with this:

```
$.each( obj, function( key, value ) {
    sum += value;
});

console.log( sum ); // 3
```

Again, we don't have to directly access `obj[key]` as the value is passed directly to the callback.

Note that `$.each()` is for plain objects, arrays, array-like objects *that are not jQuery collections*.

This would be considered incorrect:

```
// Incorrect:
$.each( $( "p" ), function() {
```



```

        // Do something
    });

```

For jQuery collections, use `.each()`.

.each()

[.each\(\)](#) is used directly on a jQuery collection. It iterates over each matched element in the collection and performs a callback on that object. The index of the current element within the collection is passed as an argument to the callback. The value (the DOM element in this case) is also passed, but the callback is fired within the context of the current matched element so the `this` keyword points to the current element as expected in other jQuery callbacks.

For example, given the following markup:

```

<ul>
    <li><a href="#">Link 1</a></li>
    <li><a href="#">Link 2</a></li>
    <li><a href="#">Link 3</a></li>
</ul>

```

`.each()` may be used like so:

```

$( "li" ).each( function( index, element ){
    console.log( $( this ).text() );
});

// Logs the following:
// Link 1
// Link 2
// Link 3

```

The Second Argument

The question is often raised, "If `this` is the element, why is there a second DOM element argument passed to the callback?"

Whether intentional or inadvertent, the execution context may change. When consistently using the keyword `this`, it's easy to end up confusing ourselves or other developers reading the code. Even if the execution context remains the same, it may be more readable to use the second parameter as a named parameter. For example:

```

$( "li" ).each( function( index, listItem ) {
    this === listItem; // true

    // For example only. You probably shouldn't call $.ajax() in a loop.
    $.ajax({
        success: function( data ) {
            // The context has changed.
            // The "this" keyword no longer refers to listItem.
            this !== listItem; // true
        }
    });
});

```

Sometimes `.each()` Isn't Necessary

Many jQuery methods implicitly iterate over the entire collection, applying their behavior to each matched element. For example, this is unnecessary:

```

$( "li" ).each( function( index, el ) {
    $( el ).addClass( "newClass" );
});

```

And this is fine:

```

$( "li" ).addClass( "newClass" );

```

Each `` in the document will have the class "newClass" added.

On the other hand, some methods do not iterate over the collection. `.each()` is required when we need to get information from the element before setting a new value.

This will not work:

```
// Doesn't work:
$( "input" ).val( $( this ).val() + "%" );

// .val() does not change the execution context, so this === window
```

Rather, this is how it should be written:

```
$( "input" ).each( function( i, el ) {
    var elem = $( el );
    elem.val( elem.val() + "%" );
});
```

The following is a list of methods that require `.each()`:

- [.attr\(\)](#) (getter)
- [.css\(\)](#) (getter)
- [.data\(\)](#) (getter)
- [.height\(\)](#) (getter)
- [.html\(\)](#) (getter)
- [.innerHeight\(\)](#)
- [.innerWidth\(\)](#)
- [.offset\(\)](#) (getter)
- [.outerHeight\(\)](#)
- [.outerWidth\(\)](#)
- [.position\(\)](#)
- [.prop\(\)](#) (getter)
- [.scrollLeft\(\)](#) (getter)
- [.scrollTop\(\)](#) (getter)
- [.val\(\)](#) (getter)
- [.width\(\)](#) (getter)

Note that in most cases, the "getter" signature returns the result from the first element in a jQuery collection while the setter acts over the entire collection of matched elements. The exception to this is `.text()` where the getter signature will return a concatenated string of text from all matched elements.

In addition to a setter value, the attribute, property, CSS setters, and DOM insertion "setter" methods (i.e. `.text()` and `.html()`) accept anonymous callback functions that are applied to each element in the matching set. The arguments passed to the callback are the index of the matched element within the set and the result of the 'getter' signature of the method.

For example, these are equivalent:

```
$( "input" ).each( function( i, el ) {
    var elem = $( el );
    elem.val( elem.val() + "%" );
});

$( "input" ).val(function( index, value ) {
    return value + "%";
});
```

One other thing to keep in mind with this implicit iteration is that traversal methods such as `.children()` or `.parent()` will act on each matched element in a collection, returning a combined collection of all children or parent nodes.

[.map\(\)](#)

There is a common iteration use case that can be better handled by using the `.map()` method. Anytime we want to create an array or concatenated string based on all matched elements in our jQuery selector, we're better served using `.map()`.

For example instead of doing this:

```
var newArr = [];

$( "li" ).each( function() {
    newArr.push( this.id );
});
```

We can do this:

```
$( "li" ).map( function(index, element) {
    return this.id;
}).get();
```

Notice the `.get()` chained at the end. `.map()` actually returns a jQuery-wrapped collection, even if we return strings out of the callback. We need to use the argument-less version of `.get()` in order to return a basic JavaScript array that we can work with. To concatenate into a string, we can chain the plain JS `.join()` array method after `.get()`.

[\\$.map](#)

Like `$.each()` and `.each()`, there is a `$.map()` as well as `.map()`. The difference is also very similar to both `.each()` methods. `$.map()` works on plain JavaScript arrays while `.map()` works on jQuery element collections. Because it's working on a plain array, `$.map()` returns a plain array and `.get()` does not need to be called – in fact, it will throw an error as it's not a native JavaScript method.

A word of warning: `$.map()` switches the order of callback arguments. This was done in order to match the native JavaScript `.map()` method made available in ECMAScript 5.

For example:

```
<li id="a"></li>
<li id="b"></li>
<li id="c"></li>
```

```
var arr = [{
    id: "a",
    tagName: "li"
}, {
    id: "b",
    tagName: "li"
}, {
    id: "c",
    tagName: "li"
}];

// Returns [ "a", "b", "c" ]
$( "li" ).map( function( index, element ) {
    return element.id;
}).get();

// Also returns [ "a", "b", "c" ]
// Note that the value comes first with $.map
$.map( arr, function( value, index ) {
    return value.id;
});

</script>
```

Using jQuery Core

Using jQuery's .index() Function

.index() is a method on jQuery objects that's generally used to search for a given element within the jQuery object that it's called on. This method has four different signatures with different semantics that can be confusing. This article covers details about how to understand the way .index() works with each signature.

.index() with No Arguments

```
<ul>
    <div></div>
    <li id="foo1">foo</li>
    <li id="bar1">bar</li>
    <li id="baz1">baz</li>
</ul>

<div></div>

var foo = $( "#foo1" );

console.log( "Index: " + foo.index() ); // 1

var listItem = $( "li" );

// This implicitly calls .first()
console.log( "Index: " + listItem.index() ); // 1
console.log( "Index: " + listItem.first().index() ); // 1

var div = $( "div" );

// This implicitly calls .first()
console.log( "Index: " + div.index() ); // 0
console.log( "Index: " + div.first().index() ); // 0
```

In the first example, .index() gives the zero-based index of #foo1 within its parent. Since #foo1 is the second child of its parent, index() returns 1.

Note: Before jQuery 1.9, .index() only worked reliably on a single element, which is why we've used .first() on each of our examples. In jQuery 1.9+ this can be ignored, as the API was updated to define that it operates on the first element only.

.index() with a String Argument

```
<ul>
    <div class="test"></div>
    <li id="foo1">foo</li>
    <li id="bar1" class="test">bar</li>
    <li id="baz1">baz</li>
    <div class="test"></div>
</ul>
<div id="last"></div>

var foo = $( "li" );

// This implicitly calls .first()
console.log( "Index: " + foo.index( "li" ) ); // 0
console.log( "Index: " + foo.first().index( "li" ) ); // 0

var baz = $( "#baz1" );
console.log( "Index: " + baz.index( "li" ) ); // 2

var listItem = $( "#bar1" );
console.log( "Index: " + listItem.index( ".test" ) ); // 1

var div = $( "#last" );
console.log( "Index: " + div.index( "div" ) ); // 2
```

When .index() is called with a string argument, there are two things to consider. First, jQuery will implicitly call .first() on the original jQuery object. It will find the index of the first element, not the last element in this case. This is inconsistent, so be careful here.

The second point to consider is that jQuery is querying the entire DOM using the passed in string selector and checking the index within that newly queried jQuery object. For example, when using .index("div") in the last example above, jQuery is selecting all of the <div> elements in the document, then searching for the index that contains the first element in the jQuery object .index() is called on.

.index() with a jQuery Object Argument

```

<ul>
    <div class="test"></div>
    <li id="foo1">foo</li>
    <li id="bar1" class="test">bar</li>
    <li id="baz1">baz</li>
    <div class="test"></div>
</ul>
<div id="last"></div>

var foo = $( "li" );
var baz = $( "#baz1" );

console.log( "Index: " + foo.index( baz ) ); // 2

var tests = $( ".test" );
var bar = $( "#bar1" );

// Implicitly calls .first() on the argument.
console.log( "Index: " + tests.index( bar ) ); // 1

console.log( "Index: " + tests.index( bar.first() ) ); // 1

```

In this case, the first element of the jQuery object that is passed into `.index()` is being checked against all of the elements in the original jQuery object. The original jQuery object, on the left side of `.index()`, is array-like and is searched from index 0 through `length - 1` for the first element of the argument jQuery object.

.index() with a DOM Element Argument

In this case, the DOM element that's passed into `.index()` is being checked against all of the elements in the original jQuery object. Once all other cases are understood, this should be the simplest case. It is very similar to the previous case, except since the DOM element is passed directly, it is not taken from a jQuery object container.

Using jQuery Core - Frequently Asked Questions

Frequently Asked Questions

Using jQuery Core - Frequently Asked Questions

How do I select an item using class or ID?

This code selects an element with an ID of "myDivId". Since IDs are unique, this expression always selects either zero or one elements depending upon whether or not an element with the specified ID exists.

```
$( "#myDivId" );
```

This code selects an element with a class of "myCssClass". Since any number of elements can have the same class, this expression will select any number of elements.

```
$( ".myCssClass" );
```

A jQuery object containing the selected element can be assigned to a JavaScript variable like normal:

```
var myDivElement = $( "#myDivId" );
```

Usually, elements in a jQuery object are acted on by other jQuery functions:

```
var myValue = $( "#myDivId" ).val(); // Get the value of a form input.
```

```
$( "#myDivId" ).val( "hello world" ); // Set the value of a form input.
```

Related Articles

- [Selecting Elements](#)
- [Working with Selections](#)

Using jQuery Core - Frequently Asked Questions

How do I select elements when I already have a DOM element?

If you have a variable containing a DOM element, and want to select elements related to that DOM element, simply wrap it in a jQuery object.

```
var myDomElement = document.getElementById( "foo" ); // A plain DOM element.  
$( myDomElement ).find( "a" ); // Finds all anchors inside the DOM element.
```

Many people try to concatenate a DOM element or jQuery object with a CSS selector, like so:

```
$( myDomElement + ".bar" ); // This is equivalent to $( "[object HTMLElement].bar" );
```

Unfortunately, you cannot concatenate strings to objects.

Related Articles

- [The jQuery Object](#)

Using jQuery Core - Frequently Asked Questions

How do I test whether an element has a particular class?

[.hasClass\(\)](#) (added in version 1.2) handles this common use case:

```
$( "div" ).click(function() {  
    if ( $( this ).hasClass( "protected" ) ) {  
        $( this )  
            .animate({ left: -10 })  
            .animate({ left: 10 })  
            .animate({ left: -10 })  
            .animate({ left: 10 })  
            .animate({ left: 0 });  
    }  
});
```

You can also use the [.is\(\)](#) method along with an appropriate selector for more advanced matching:

```
if ( $( "#myDiv" ).is( ".pretty.awesome" ) ) {  
    $( "#myDiv" ).show();  
}
```

Note that this method allows you to test for other things as well. For example, you can test whether an element is hidden (by using the custom [:hidden](#) selector):

```
if ( $( "#myDiv" ).is( ":hidden" ) ) {  
    $( "#myDiv" ).show();  
}
```

Using jQuery Core - Frequently Asked Questions

How do I test whether an element exists?

Use the [.length](#) property of the jQuery collection returned by your selector:

```
if ( $( "#myDiv" ).length ) {  
    $( "#myDiv" ).show();  
}
```

Note that it isn't always necessary to test whether an element exists. The following code will show the element if it exists, and do nothing (with no errors) if it does not:

```
$( "#myDiv" ).show();
```

Using jQuery Core - Frequently Asked Questions

How do I determine the state of a toggled element?

You can determine whether an element is collapsed or not by using the `:visible` and `:hidden` selectors.

```
var isVisible = $( "#myDiv" ).is( ":visible" );  
var isHidden = $( "#myDiv" ).is( ":hidden" );
```

If you're simply acting on an element based on its visibility, just include `:visible` or `:hidden` in the selector expression. For example:

```
$( "#myDiv:visible" ).animate({  
    left: "+=200px"  
}, "slow" );
```

Using jQuery Core - Frequently Asked Questions

How do I select an element by an ID that has characters used in CSS notation?

Because jQuery uses CSS syntax for selecting elements, some characters are interpreted as CSS notation. In order to tell jQuery to treat these characters literally rather than as CSS notation, they must be escaped by placing two backslashes in front of them.

See the [Selector documentation](#) for further details.

```
// Does not work:
$( "#some:id" )

// Works!
$( "#some\\:id" )

// Does not work:
$( "#some.id" )

// Works!
$( "#some\\.id" )
```

The following function takes care of escaping these characters and places a "#" at the beginning of the ID string:

```
function jq( myid ) {
    return "#" + myid.replace( /(:|\.|\[|\]|,|=|@)/g, "\\$1" );
}
```

The function can be used like so:

```
$( jq( "some.id" ) )
```

Using jQuery Core - Frequently Asked Questions

How do I disable/enable a form element?

You can enable or disable a form element using the `.prop()` method:

```
// Disable #x
$( "#x" ).prop( "disabled", true );

// Enable #x
$( "#x" ).prop( "disabled", false );
```

Using jQuery Core - Frequently Asked Questions

How do I check/uncheck a checkbox input or radio button?

You can check or uncheck a checkbox element or a radio button using the `.prop()` method:

```
// Check #x
$( "#x" ).prop( "checked", true );

// Uncheck #x
$( "#x" ).prop( "checked", false );
```

Using jQuery Core - Frequently Asked Questions

How do I get the text value of a selected option?

Select elements typically have two values that you want to access. First there's the value to be sent to the server, which is easy:

```
$( "#myselect" ).val();  
// => 1
```

The second is the text value of the select. For example, using the following select box:

```
<select id="myselect">  
  <option value="1">Mr</option>  
  <option value="2">Mrs</option>  
  <option value="3">Ms</option>  
  <option value="4">Dr</option>  
  <option value="5">Prof</option>  
</select>
```

If you wanted to get the string "Mr" if the first option was selected (instead of just "1") you would do that in the following way:

```
$( "#myselect option:selected" ).text();  
// => "Mr"
```

Using jQuery Core - Frequently Asked Questions

How do I replace text from the 3rd element of a list of 10 items?

Either the `:eq()` selector or the `.eq()` method will allow you to select the proper item. However, to replace the text, you must get the value before you set it:

```
// This doesn't work; text() returns a string, not the jQuery object:
$( this ).find( "li a" ).eq( 2 ).text().replace( "foo", "bar" );

// This works:
var thirdLink = $( this ).find( "li a" ).eq( 2 );

var linkText = thirdLink.text().replace( "foo", "bar" );

thirdLink.text( linkText );
```

The first example just discards the modified text. The second example saves the modified text and then replaces the old text with the new modified text. Remember, `.text()` gets; `.text("foo")` sets.

Using jQuery Core - Frequently Asked Questions

How do I pull a native DOM element from a jQuery object?

A jQuery object is an array-like wrapper around one or more DOM elements. To get a reference to the actual DOM elements (instead of the jQuery object), you have two options. The first (and fastest) method is to use array notation:

```
$( "#foo" )[ 0 ]; // Equivalent to document.getElementById( "foo" )
```

The second method is to use the [.get\(\)](#) function:

```
$( "#foo" ).get( 0 ); // Identical to above, only slower.
```

You can also call [.get\(\)](#) without any arguments to retrieve a true array of DOM elements.

Events

jQuery provides simple methods for attaching event handlers to selections. When an event occurs, the provided function is executed. Inside the function, `this` refers to the DOM element that initiated the event.

For details on jQuery events, visit the [Events documentation on api.jquery.com](https://api.jquery.com/events/).

The event handling function can receive an event object. This object can be used to determine the nature of the event, and to prevent the event's default behavior.

For details on the event object, visit the [Event object documentation on api.jquery.com](https://api.jquery.com/event-object/).

Events

jQuery Event Basics

jQuery Event Basics

Setting Up Event Responses on DOM Elements

jQuery makes it straightforward to set up event-driven responses on page elements. These events are often triggered by the end user's interaction with the page, such as when text is entered into a form element or the mouse pointer is moved. In some cases, such as the page load and unload events, the browser itself will trigger the event.

jQuery offers convenience methods for most native browser events. These methods — including `.click()`, `.focus()`, `.blur()`, `.change()`, etc. — are shorthand for jQuery's `.on()` method. The `on` method is useful for binding the same handler function to multiple events, when you want to provide data to the event handler, when you are working with custom events, or when you want to pass an object of multiple events and handlers.

```
// Event setup using a convenience method
$( "p" ).click(function() {
    console.log( "You clicked a paragraph!" );
});

// Equivalent event setup using the `.on()` method
$( "p" ).on( "click", function() {
    console.log( "click" );
});
```

Extending Events to New Page Elements

It is important to note that `.on()` can only create event listeners on elements that exist *at the time you set up the listeners*. Similar elements created after the event listeners are established will not automatically pick up event behaviors you've set up previously. For example:

```
$( document ).ready(function(){

    // Sets up click behavior on all button elements with the alert class
    // that exist in the DOM when the instruction was executed
    $( "button.alert" ).on( "click", function() {
        console.log( "A button with the alert class was clicked!" );
    });

    // Now create a new button element with the alert class. This button
    // was created after the click listeners were applied above, so it
    // will not have the same click behavior as its peers
    $( "<button class='alert'>Alert!</button>" ).appendTo( document.body );
});
```

Consult the article on event delegation to see how to use `.on()` so that event behaviors will be extended to new elements without having to rebind them.

Inside the Event Handler Function

Every event handling function receives an event object, which contains many properties and methods. The event object is most commonly used to prevent the default action of the event via the `.preventDefault()` method. However, the event object contains a number of other useful properties and methods, including:

pageX, pageY

The mouse position at the time the event occurred, relative to the top left corner of the page display area (not the entire browser window).

type

The type of the event (e.g., "click").

which

The button or key that was pressed.

data

Any data that was passed in when the event was bound. For example:

```
// Event setup using the `.on()` method with data
$( "input" ).on(
    "change",
    { foo: "bar" }, // Associate data with event binding
    function( eventObject ) {
        console.log( "An input value has changed! ", eventObject.data.foo);
    }
);
```

target

The DOM element that initiated the event.

namespace

The namespace specified when the event was triggered.

timeStamp

The difference in milliseconds between the time the event occurred in the browser and January 1, 1970.

preventDefault()

Prevent the default action of the event (e.g. following a link).

stopPropagation()

Stop the event from bubbling up to other elements.

In addition to the event object, the event handling function also has access to the DOM element that the handler was bound to via the keyword `this`. To turn the DOM element into a jQuery object that we can use jQuery methods on, we simply do `$(this)`, often following this idiom:

```
var element = $( this );
```

A fuller example would be:

```
// Preventing a link from being followed
$( "a" ).click(function( eventObject ) {
    var elem = $( this );
    if ( elem.attr( "href" ).match( /evil/ ) ) {
        eventObject.preventDefault();
        elem.addClass( "evil" );
    }
});
```

Setting Up Multiple Event Responses

Quite often elements in your application will be bound to multiple events. If multiple events are to share the same handling function, you can provide the event types as a space-separated list to `.on()`:

```
// Multiple events, same handler
$( "input" ).on(
    "click change", // Bind handlers for multiple events
    function() {
        console.log( "An input was clicked or changed!" );
    }
);
```

When each event has its own handler, you can pass an object into `.on()` with one or more key/value pairs, with the key being the event name and the value being the function to handle the event.

```
// Binding multiple events with different handlers
$( "p" ).on({
    "click": function() { console.log( "clicked!" ); },
```

```

        "mouseover": function() { console.log( "hovered!" ); }
    });

```

Namespacing Events

For complex applications and for plugins you share with others, it can be useful to namespace your events so you don't unintentionally disconnect events that you didn't or couldn't know about.

```

// Namespacing events
$( "p" ).on( "click.myNamespace", function() { /* ... */ } );
$( "p" ).off( "click.myNamespace" );
$( "p" ).off( ".myNamespace" ); // Unbind all events in the namespace

```

Tearing Down Event Listeners

To remove an event listener, you use the `.off()` method and pass in the event type to off. If you attached a named function to the event, then you can isolate the event tear down to just that named function by passing it as the second argument.

```

// Tearing down all click handlers on a selection
$( "p" ).off( "click" );

// Tearing down a particular click handler, using a reference to the function
var foo = function() { console.log( "foo" ); };
var bar = function() { console.log( "bar" ); };

$( "p" ).on( "click", foo ).on( "click", bar );
$( "p" ).off( "click", bar ); // foo is still bound to the click event

```

Setting Up Events to Run Only Once

Sometimes you need a particular handler to run only once — after that, you may want no handler to run, or you may want a different handler to run. jQuery provides the `.one()` method for this purpose.

```

// Switching handlers using the `.one()` method
$( "p" ).one( "click", firstClick );

function firstClick() {
    console.log( "You just clicked this for the first time!" );

    // Now set up the new handler for subsequent clicks;
    // omit this step if no further click responses are needed
    $( this ).click( function() { console.log( "You have clicked this before!" ); } );
};

```

Note that in the code snippet above, the `firstClick` function will be executed for the first click on *each* paragraph element rather than the function being removed from *all* paragraphs when *any* paragraph is clicked for the first time.

`.one()` can also be used to bind multiple events:

```

// Using .one() to bind several events
$( "input[id]" ).one( "focus mouseover keydown", firstEvent);

function firstEvent( eventObject ) {
    console.log( "A " + eventObject.type + " event occurred for the first time on the input with id " + this.id );
}

```

In this case, the `firstEvent` function will be executed once *for each event*. For the snippet above, this means that once an input element gains focus, the handler function will still execute for the first keydown event on that element.

Events

Event Helpers

jQuery offers a few event-related helper functions that save you a few keystrokes. Here is an example of one, the `.hover()` function.

.hover()

The [`.hover\(\)`](#) method lets you pass one or two functions to be run when the `mouseenter` and `mouseleave` events occur on an element. If you pass one function, it will be run for both events; if you pass two functions, the first will run for `mouseenter`, and the second will run for `mouseleave`.

Note: Prior to jQuery 1.4, the `.hover()` method required two functions.

```
// The hover helper function
$( "#menu li" ).hover(function() {
    $( this ).toggleClass( "hover" );
});
```

You can find more helper functions on the [API site for Events](#).

Events

Introducing Events

Introduction

Web pages are all about interaction. Users perform a countless number of actions such as moving their mice over the page, clicking on elements, and typing in textboxes — all of these are examples of events. In addition to these user events, there are a slew of others that occur, like when the page is loaded, when video begins playing or is paused, etc. Whenever something interesting occurs on the page, an event is fired, meaning that the browser basically announces that something has happened. It's this announcement that allows developers to "listen" for events and react to them appropriately.

What's a DOM event?

As mentioned, there are a myriad of event types, but perhaps the ones that are easiest to understand are user events, like when someone clicks on an element or types into a form. These types of events occur on an element, meaning that when a user clicks on a button for example, the button has had an event occur on it. While user interactions aren't the only types of DOM events, they're certainly the easiest to understand when starting out. Mozilla Developer Network has a good reference of [available DOM events](#).

Ways to listen for events

There are many ways to listen for events. Actions are constantly occurring on a webpage, but the developer is only notified about them if they're *listening* for them. Listening for an event basically means you're waiting for the browser to tell you that a specific event has occurred and then you'll specify how the page should react.

To specify to the browser what to do when an event occurs, you provide a function, also known as an *event handler*. This function is executed whenever the event occurs (or until the event is unbound).

For instance, to alert a message whenever a user clicks on a button, you might write something like this:

```
<button onclick="alert('Hello')">Say hello</button>
```

The event we want to listen to is specified by the button's `onclick` attribute, and the event handler is the `alert` function which alerts "Hello" to the user. While this works, it's an abysmal way to achieve this functionality for a couple of reasons:

1. First, we're coupling our view code (HTML) with our interaction code (JS). That means that whenever we need to update functionality, we'd have to edit our HTML which is just a bad practice and a maintenance nightmare.
2. Second, it's not scalable. If you had to attach this functionality onto numerous buttons, you'd not only bloat the page with a bunch of repetitious code, but you would again destroy maintainability.

Utilizing inline event handlers like this can be considered *obtrusive JavaScript*, but its opposite, *unobtrusive JavaScript* is a much more common way of discussing the topic. The notion of *unobtrusive JavaScript* is that your HTML and JS are kept separate and are therefore more maintainable. Separation of concerns is important because it keeps like pieces of code together (i.e. HTML, JS, CSS) and unlike pieces of code apart, facilitating changes, enhancements, etc. Furthermore, unobtrusive JavaScript stresses the importance of adding the least amount of cruft to a page as possible. If a user's browser doesn't support JavaScript, then it shouldn't be intertwined into the markup of the page. Also, to prevent naming collisions, JS code should utilize a single namespace for different pieces of functionality or libraries. jQuery is a good example of this, in that the `jquery` object/constructor (and also the `$` alias to `jquery`) only utilizes a single global variable, and all of jQuery's functionality is packaged into that one object.

To accomplish the desired task unobtrusively, let's change our HTML a little bit by removing the `onclick` attribute and replacing it with an `id`, which we'll utilize to "hook onto" the button

from within a script file.

```
<button id="helloBtn">Say hello</button>
```

If we wanted to be informed when a user clicks on that button unobtrusively, we might do something like the following in a separate script file:

```
// Event binding using addEventListener
var helloBtn = document.getElementById( "helloBtn" );

helloBtn.addEventListener( "click", function( event ) {
    alert( "Hello." );
}, false );
```

Here we're saving a reference to the button element by calling `getElementById` and assigning its return value to a variable. We then call `addEventListener` and provide an event handler function that will be called whenever that event occurs. While there's nothing wrong with this code as it will work fine in modern browsers, it won't fare well in versions of IE prior to IE9. This is because Microsoft chose to implement a different method, `attachEvent`, as opposed to the W3C standard `addEventListener`, and didn't get around to changing it until IE9 was released. For this reason, it's beneficial to utilize jQuery because it abstracts away browser inconsistencies, allowing developers to use a single API for these types of tasks, as seen below.

```
// Event binding using a convenience method
$( "#helloBtn" ).click(function( event ) {
    alert( "Hello." );
});
```

The `$("#helloBtn")` code selects the button element using the `$` (a.k.a. jQuery) function and returns a jQuery object. The jQuery object has a bunch of methods (functions) available to it, one of them named `click`, which resides in the jQuery object's prototype. We call the `click` method on the jQuery object and pass along an anonymous function event handler that's going to be executed when a user clicks the button, alerting "Hello." to the user.

There are a number of ways that events can be listened for using jQuery:

```
// The many ways to bind events with jQuery
// Attach an event handler directly to the button using jQuery's
// shorthand `click` method.
$( "#helloBtn" ).click(function( event ) {
    alert( "Hello." );
});

// Attach an event handler directly to the button using jQuery's
// `bind` method, passing it an event string of `click`
$( "#helloBtn" ).bind( "click", function( event ) {
    alert( "Hello." );
});

// As of jQuery 1.7, attach an event handler directly to the button
// using jQuery's `on` method.
$( "#helloBtn" ).on( "click", function( event ) {
    alert( "Hello." );
});

// As of jQuery 1.7, attach an event handler to the `body` element that
// is listening for clicks, and will respond whenever *any* button is
// clicked on the page.
$( "body" ).on({
    click: function( event ) {
        alert( "Hello." );
    }
}, "button" );

// An alternative to the previous example, using slightly different syntax.
$( "body" ).on( "click", "button", function( event ) {
    alert( "Hello." );
});
```

As of jQuery 1.7, all events are bound via the `on` method, whether you call it directly or whether you use an alias/shortcut method such as `bind` or `click`, which are mapped to the `on` method internally. With this in mind, it's beneficial to use the `on` method because the others are all just syntactic sugar, and utilizing the `on` method is going to result in faster and more consistent code.

Let's look at the `on` examples from above and discuss their differences. In the first example, a string of `click` is passed as the first argument to the `on` method, and an anonymous function is passed as the second. This looks a lot like the `bind` method before it. Here, we're attaching an

event handler directly to `#helloBtn`. If there were any other buttons on the page, they wouldn't alert "Hello" when clicked because the event is only attached to `#helloBtn`.

In the second `on` example, we're passing an object (denoted by the curly braces `{}`), which has a property of `click` whose value is an anonymous function. The second argument to the `on` method is a jQuery selector string of `button`. While examples 1–3 are functionally equivalent, example 4 is different in that the `body` element is listening for click events that occur on *any* button element, not just `#helloBtn`. The final example above is exactly the same as the one preceding it, but instead of passing an object, we pass an event string, a selector string, and the callback. Both of these are examples of event delegation, a process by which an element higher in the DOM tree listens for events occurring on its children.

Event delegation

Event delegation works because of the notion of *event bubbling*. For most events, whenever something occurs on a page (like an element is clicked), the event travels from the element it occurred on, up to its parent, then up to the parent's parent, and so on, until it reaches the root element, a.k.a. the `window`. So in our table example, whenever a `td` is clicked, its parent `tr` would also be notified of the click, the parent `table` would be notified, the `body` would be notified, and ultimately the `window` would be notified as well. While event bubbling and delegation work well, the delegating element (in our example, the `table`) should always be as close to the delegates as possible so the event doesn't have to travel way up the DOM tree before its handler function is called.

The two main pros of event delegation over binding directly to an element (or set of elements) are performance and the aforementioned event bubbling. Imagine having a large table of 1,000 cells and binding to an event for each cell. That's 1,000 separate event handlers that the browser has to attach, even if they're all mapped to the same function. Instead of binding to each individual cell though, we could instead use delegation to listen for events that occur on the parent table and react accordingly. One event would be bound instead of 1,000, resulting in way better performance and memory management.

The event bubbling that occurs affords us the ability to add cells via Ajax for example, without having to bind events directly to those cells since the parent table is listening for clicks and is therefore notified of clicks on its children. If we weren't using delegation, we'd have to constantly bind events for every cell that's added which is not only a performance issue, but could also become a maintenance nightmare.

The event object

In all of the previous examples, we've been using anonymous functions and specifying an event argument within that function. Let's change it up a little bit.

```
// Binding a named function
function sayHello( event ) {
    alert( "Hello." );
}

$( "#helloBtn" ).on( "click", sayHello );
```

In this slightly different example, we're defining a function called `sayHello` and then passing that function into the `on` method instead of an anonymous function. So many online examples show anonymous functions used as event handlers, but it's important to realize that you can also pass defined functions as event handlers as well. This is important if different elements or different events should perform the same functionality. This helps to keep your code [DRY](#).

But what about that `event` argument in the `sayHello` function — what is it and why does it matter? In all DOM event callbacks, jQuery passes an *event object* argument which contains information about the event, such as precisely when and where it occurred, what type of event it was, which element the event occurred on, and a plethora of other information. Of course you don't have to call it `event`; you could call it `e` or whatever you want to, but `event` is a pretty common convention.

If the element has default functionality for a specific event (like a link opens a new page, a button in a form submits the form, etc.), that default functionality can be canceled. This is often useful for Ajax requests. When a user clicks on a button to submit a form via Ajax, we'd want to cancel the button/form's default action (to submit it to the form's `action` attribute), and we would instead do an Ajax request to accomplish the same task for a more seamless

experience. To do this, we would utilize the event object and call its `.preventDefault()` method. We can also prevent the event from bubbling up the DOM tree using `.stopPropagation()` so that parent elements aren't notified of its occurrence (in the case that event delegation is being used).

```
// Preventing a default action from occurring and stopping the event bubbling
$( "form" ).on( "submit", function( event ) {

    // Prevent the form's default submission.
    event.preventDefault();

    // Prevent event from bubbling up DOM tree, prohibiting delegation
    event.stopPropagation();

    // Make an AJAX request to submit the form data

});
```

When utilizing both `.preventDefault()` and `.stopPropagation()` simultaneously, you can instead return `false` to achieve both in a more concise manner, but it's advisable to only return `false` when both are actually necessary and not just for the sake of terseness. A final note on `.stopPropagation()` is that when using it in delegated events, the soonest that event bubbling can be stopped is when the event reaches the element that is delegating it.

It's also important to note that the event object contains a property called `originalEvent`, which is the event object that the browser itself created. jQuery wraps this native event object with some useful methods and properties, but in some instances, you'll need to access the original event via `event.originalEvent` for instance. This is especially useful for touch events on mobile devices and tablets.

Finally, to inspect the event itself and see all of the data it contains, you should log the event in the browser's console using `console.log`. This will allow you to see all of an event's properties (including the `originalEvent`) which can be really helpful for debugging.

```
// Logging an event's information
$( "form" ).on( "submit", function( event ) {

    // Prevent the form's default submission.
    event.preventDefault();

    // Log the event object for inspectin'
    console.log( event );

    // Make an AJAX request to submit the form data

});
```

Events

Handling Events

jQuery provides a method `.on()` to respond to any event on the selected elements. This is called an *event binding*. Although `.on()` isn't the only method provided for event binding, it is a best practice to use this for jQuery 1.7+. To learn more, [read more about the evolution of event binding in jQuery](#).

The `.on()` method provides several useful features:

- [Bind any event triggered on the selected elements to an event handler](#)
- [Bind multiple events to one event handler](#)
- [Bind multiple events and multiple handlers to the selected elements](#)
- [Use details about the event in the event handler](#)
- [Pass data to the event handler for custom events](#)
- [Bind events to elements that will be rendered in the future](#)

Examples

Simple event binding

```
// When any <p> tag is clicked, we expect to see '<p> was clicked' in the console.
$( "p" ).on( "click", function() {
    console.log( "<p> was clicked" );
});
```

Many events, but only one event handler

Suppose you want to trigger the same event whenever the mouse hovers over or leaves the selected elements. The best practice for this is to use "mouseenter mouseleave". Note the difference between this and the next example.

```
// When a user focuses on or changes any input element,
// we expect a console message bind to multiple events
$( "div" ).on( "mouseenter mouseleave", function() {
    console.log( "mouse hovered over or left a div" );
});
```

Many events and handlers

Suppose that instead you want different event handlers for when the mouse enters and leaves an element. This is more common than the previous example. For example, if you want to show and hide a tooltip on hover, you would use this.

`.on()` accepts an object containing multiple events and handlers.

```
$( "div" ).on({
    mouseenter: function() {
        console.log( "hovered over a div" );
    },
    mouseleave: function() {
        console.log( "mouse left a div" );
    },
    click: function() {
        console.log( "clicked on a div" );
    }
});
```

The event object

Handling events can be tricky. It's often helpful to use the extra information contained in the event object passed to the event handler for more control. To become familiar with the event object, use this code to inspect it in your browser console after you click on a <div> in the page. For a breakdown of the event object, see [Inside the Event Handling Function](#).

```
$( "div" ).on( "click", function( event ) {
    console.log( "event object:" );
    console.dir( event );
});
```

Passing data to the event handler

You can pass your own data to the event object.

```
$( "p" ).on( "click", {
    foo: "bar"
}, function( event ) {
    console.log( "event data: " + event.data.foo + " (should be 'bar')" );
});
```

Binding events to elements that don't exist yet

This is called *event delegation*. Here's an example just for completeness, but see the page on [Event Delegation](#) for a full explanation.

```
$( "ul" ).on( "click", "li", function() {
    console.log( "Something in a <ul> was clicked, and we detected that it was
an <li> element." );
});
```

Connecting Events to Run Only Once

Sometimes you need a particular handler to run only once — after that, you may want no handler to run, or you may want a different handler to run. jQuery provides the `.one()` method for this purpose.

```
// Switching handlers using the `.one()` method
$( "p" ).one( "click", function() {
    console.log( "You just clicked this for the first time!" );
    $( this ).click(function() {
        console.log( "You have clicked this before!" );
    });
});
```

The `.one()` method is especially useful if you need to do some complicated setup the first time an element is clicked, but not subsequent times.

`.one()` accepts the same arguments as `.on()` which means it supports multiple events to one or multiple handlers, passing custom data and event delegation.

Disconnecting Events

Although all the fun of jQuery occurs in the `.on()` method, its counterpart is just as important if you want to be a responsible developer. `.off()` cleans up that event binding when you don't need it anymore. Complex user interfaces with lots of event bindings can bog down browser performance, so using the `.off()` method diligently is a best practice to ensure that you only have the event bindings that you need, when you need them.

```
// Unbinding all click handlers on a selection
$( "p" ).off( "click" );

// Unbinding a particular click handler, using a reference to the function
var foo = function() {
    console.log( "foo" );
};

var bar = function() {
    console.log( "bar" );
};

$( "p" ).on( "click", foo ).on( "click", bar );

// foo will stay bound to the click event
$( "p" ).off( "click", bar );
```

Namespacing Events

For complex applications and for plugins you share with others, it can be useful to namespace your events so you don't unintentionally disconnect events that you didn't or couldn't know about. For details, see [Event Namespacing](#).

Events

Inside the Event Handling Function

Every event handling function receives an event object, which contains many properties and methods. The event object is most commonly used to prevent the default action of the event via the `.preventDefault()` method. However, the event object contains a number of other useful properties and methods, including:

pageX, pageY

The mouse position at the time the event occurred, relative to the top left of the page.

type

The type of the event (e.g. "click").

which

The button or key that was pressed.

data

Any data that was passed in when the event was bound.

target

The DOM element that initiated the event.

preventDefault()

Prevent the default action of the event (e.g. following a link).

stopPropagation()

Stop the event from bubbling up to other elements.

In addition to the event object, the event handling function also has access to the DOM element that the handler was bound to via the keyword `this`. To turn the DOM element into a jQuery object that we can use jQuery methods on, we simply do `$(this)`, often following this idiom:

```
var elem = $( this );

// Preventing a link from being followed
$( "a" ).click(function( event ) {
    var elem = $( this );
    if ( elem.attr( "href" ).match( "evil" ) ) {
        event.preventDefault();
        elem.addClass( "evil" );
    }
});
```

Events

Understanding Event Delegation

Introduction

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

Example

For the remainder of the lesson, we will reference the following HTML structure:

```
<html>
<body>
<div id="container">
  <ul id="list">
    <li><a href="http://domain1.com">Item #1</a></li>
    <li><a href="/local/path/1">Item #2</a></li>
    <li><a href="/local/path/2">Item #3</a></li>
    <li><a href="http://domain4.com">Item #4</a></li>
  </ul>
</div>
</body>
</html>
```

When an anchor in our `#list` group is clicked, we want to log its text to the console. Normally we could directly bind to the click event of each anchor using the `.on()` method:

```
// Attach a directly bound event handler
$( "#list a" ).on( "click", function( event ) {
  event.preventDefault();
  console.log( $( this ).text() );
});
```

While this works perfectly fine, there are drawbacks. Consider what happens when we add a new anchor after having already bound the above listener:

```
// Add a new element on to our existing list
$( "#list" ).append( "<li><a href='http://newdomain.com'>Item #5</a></li>" );
```

If we were to click our newly added item, nothing would happen. This is because of the directly bound event handler that we attached previously. Direct events are only attached to elements at the time the `.on()` method is called. In this case, since our new anchor did not exist when `.on()` was called, it does not get the event handler.

Event Propagation

Understanding how events propagate is an important factor in being able to leverage Event Delegation. Any time one of our anchor tags is clicked, a *click* event is fired for that anchor, and then bubbles up the DOM tree, triggering each of its parent click event handlers:

- `<a>`
- ``
- `<ul #list>`
- `<div #container>`
- `<body>`
- `<html>`
- *document* root

This means that anytime you click one of our bound anchor tags, you are effectively clicking the entire document body! This is called *event bubbling* or *event propagation*.

Since we know how events bubble, we can create a *delegated* event:

```
// Attach a delegated event handler
$( "#list" ).on( "click", "a", function( event ) {
  event.preventDefault();
  console.log( $( this ).text() );
});
```

Notice how we have moved the `a` part from the selector to the second parameter position of the `.on()` method. This second, selector parameter tells the handler to listen for the specified event, and when it hears it, check to see if the triggering element for that event matches the second parameter. In this case, the triggering event is our anchor tag, which matches that parameter. Since it matches, our anonymous function will execute. We have now attached a single *click* event listener to our `` that will listen for clicks on its descendant anchors, instead of attaching an unknown number of directly bound events to the existing anchor tags only.

Using the Triggering Element

What if we wanted to open the link in a new window if that link is an external one (as denoted here by beginning with "http")?

```
// Attach a delegated event handler
$( "#list" ).on( "click", "a", function( event ) {
    var elem = $( this );
    if ( elem.is( "[href^='http']" ) ) {
        elem.attr( "target", "_blank" );
    }
});
```

This simply passes the `.is()` method a selector to see if the `href` attribute of the element starts with "http". We have also removed the `event.preventDefault();` statement as we want the default action to happen (which is to follow the `href`).

We can actually simplify our code by allowing the selector parameter of `.on()` do our logic for us:

```
// Attach a delegated event handler with a more refined selector
$( "#list" ).on( "click", "a[href^='http']", function( event ) {
    $( this ).attr( "target", "_blank" );
});
```

Summary

Event delegation refers to the process of using event propagation (bubbling) to handle events at a higher level in the DOM than the element on which the event originated. It allows us to attach a single event listener for elements that exist now or in the future.

Events

Triggering Event Handlers

jQuery provides a way to trigger the event handlers bound to an element without any user interaction via the `.trigger()` method.

What handlers can be `.trigger()`'d?

jQuery's event handling system is a layer on top of native browser events. When an event handler is added using `.on("click", function() {...})`, it can be triggered using jQuery's `.trigger("click")` because jQuery stores a reference to that handler when it is originally added. Additionally, it will trigger the JavaScript inside the `onclick` attribute. The `.trigger()` function cannot be used to mimic native browser events, such as clicking on a file input box or an anchor tag. This is because, there is no event handler attached using jQuery's event system that corresponds to these events.

```
<a href="http://learn.jquery.com">Learn jQuery</a>
// This will not change the current page
$( "a" ).trigger( "click" );
```

How can I mimic a native browser event, if not `.trigger()`?

In order to trigger a native browser event, you have to use [document.createEventObject](#) for < IE9 and [document.createEvent](#) for all other browsers. Using these two APIs, you can programmatically create an event that behaves exactly as if someone has actually clicked on a file input box. The default action will happen, and the browse file dialog will display.

The jQuery UI Team created [jquery.simulate.js](#) in order to simplify triggering a native browser event for use in their automated testing. Its usage is modeled after jQuery's `trigger`.

```
// Triggering a native browser event using the simulate plugin
$( "a" ).simulate( "click" );
```

This will not only trigger the jQuery event handlers, but also follow the link and change the current page.

`.trigger()` VS `.triggerHandler()`

There are four differences between `.trigger()` and `.triggerHandler()`

1. `.triggerHandler()` only triggers the event on the first element of a jQuery object.
2. `.triggerHandler()` cannot be chained. It returns the value that is returned by the last handler, not a jQuery object.
3. `.triggerHandler()` will not cause the default behavior of the event (such as a form submission).
4. Events triggered by `.triggerHandler()`, will not bubble up the DOM hierarchy. Only the handlers on the single element will fire.

For more information see the [triggerHandler documentation](#)

Don't use `.trigger()` simply to execute specific functions

While this method has its uses, it should not be used simply to call a function that was bound as a click handler. Instead, you should store the function you want to call in a variable, and pass the variable name when you do your binding. Then, you can call the function itself whenever you want, without the need for `.trigger()`.

```
// Triggering an event handler the right way
var foo = function( event ) {
    if ( event ) {
        console.log( event );
    } else {
        console.log( "this didn't come from an event!" );
    }
};
$( "p" ).on( "click", foo );
```



```
foo(); // instead of $( "p" ).trigger( "click" )
```

A more complex architecture can be built on top of trigger using the [publish-subscribe pattern](#) using [jQuery plugins](#). With this technique, `.trigger()` can be used to notify other sections of code that an application specific event has happened.

Events

History of jQuery Events

Throughout the evolution of jQuery the means of event binding has changed for various reasons ranging from performance to semantics. As of jQuery v1.7 the `.on()` method is the accepted means of both directly binding events and creating delegated events. This article aims to explore the history of *event delegation* from jQuery v1.0 to the present and how each version leverages it.

Given the following HTML, for our example we want to log the text of the each `` to console whenever it is clicked.

```
<div id="container">
  <ul id="list">
    <li>Item #1</li>
    <li>Item #2</li>
    <li>Item #3</li>
    <li>...</li>
    <li>Item #100</li>
  </ul>
</div>
```

[.bind\(\)](#) (Deprecated)

Introduced in jQuery v1.0

It is possible to use `.bind()` and attach a handler to every element.

```
$( "#list li" ).bind( "click", function( event ) {
    var elem = $( event.target );
    console.log( elem.text() );
});
```

As discussed in the [event delegation](#) article, this is not optimal.

liveQuery

liveQuery was a popular jQuery plugin that allowed for the creation of events which would be triggered for elements that existed now or in the future. This plugin did not use event delegation and used expensive CPU processing to poll the DOM for changes every 20ms and fire events accordingly.

[.bind\(\)](#) delegation (Deprecated)

Introduced in jQuery v1.0

Generally we don't associate `.bind()` with *event delegation*, however prior to jQuery v1.3 it was the only means of delegation available to us.

```
$( "#list" ).bind( "click", function( event ) {
    var elem = $( event.target );
    if ( elem.is( "li" ) ) {
        console.log( elem.text() );
    }
});
```

We are able to take advantage of *event bubbling* here by attaching a *click* event to the parent `` element. Whenever the `` is clicked, the event bubbles up to its parent, the ``, which executes our event handler. Our event handler checks to see if the **event.target** (the element that caused the event to fire) matches our selector.

[.live\(\)](#) (Deprecated)

Introduced in jQuery v1.3

All `.live()` event handlers are bound to the *document* root by default.

```
$( "#list li" ).live( "click", function( event ) {
    var elem = $( this );
    console.log( elem.text() );
});
```

When we use `.live()` our event is bound to `$(document)`. When the `` is clicked, bubbling occurs and our `click` event is fired for each of the following elements:

- ``
- `<div>`
- `<body>`
- `<html>`
- `document` root

The last element to receive the `click` event is `document`, this is where our `.live()` event is bound. `.live()` will then check to see if our selector `#list li` is the element that triggered the event, if so our event handler is executed.

[.live\(\)](#) w/ context (Deprecated)

Introduced in jQuery v1.4

Passing the `context` as a second optional argument to the `$()` function has been supported since v1.0. However support for using this `context` with the `$.live()` method was not added until v1.4.

If we were take our previous `.live()` example and provide it the default `context`, it would look like:

```
$( "#list li", document ).live( "click", function( event ) {
    var elem = $( this );
    console.log( elem.text() );
});
```

Since we can override the `context` when using the `.live()` method, we can specify a `context` that is closer to the element in the DOM hierarchy

```
$( "li", "#list" ).live( "click", function( event ) {
    var elem = $( this );
    console.log( elem.text() );
});
```

In this instance when an `` is clicked the event still bubbles all the way up the `document tree` as it did before. However, our event handler is now bound to the parent `` tag, so we do not have to wait for the event to bubble all the way up to the `document` root.

[.delegate\(\)](#) (Deprecated)

First introduced in jQuery v1.4.2

The `.delegate()` method provides a clear difference between the `context` of where to attach delegated event handler, and the `selector` to match when the event bubbles up to the delegated element.

```
$( "#list" ).delegate( "li", "click", function( event ) {
    var elem = $( this );
    console.log( elem.text() );
});
```

[.on\(\)](#)

First introduced in jQuery v1.7

The `.on()` method gives us a semantic approach for creating directly bound events as well as delegated events. It eliminates the need to use the deprecated `.bind()`, `.live()`, and `.delegate()` methods, providing a single API for creating events.

```
$( "#list" ).on( "click", "li", function( event ) {
    var elem = $( this );
    console.log( elem.text() );
});
```

Summary

All of these ways of *event delegation* were innovative and considered a best practice at the time of their release. Depending on what version of jQuery you have implemented use the appropriate means of *event delegation*.

Events

Introducing Custom Events

Custom Events

We're all familiar with the basic events — click, mouseover, focus, blur, submit, etc. — that we can latch on to as a user interacts with the browser. Custom events open up a whole new world of event-driven programming.

It can be difficult at first to understand why you'd want to use custom events, when the built-in events seem to suit your needs just fine. It turns out that custom events offer a whole new way of thinking about event-driven JavaScript. Instead of focusing on the element that triggers an action, custom events put the spotlight on the element being acted upon. This brings a bevy of benefits, including:

- Behaviors of the target element can easily be triggered by different elements using the same code.
- Behaviors can be triggered across multiple, similar, target elements at once.
- Behaviors are more clearly associated with the target element in code, making code easier to read and maintain.

Why should you care? An example is probably the best way to explain. Suppose you have a lightbulb in a room in a house. The lightbulb is currently turned on, and it's controlled by two three-way switches and a clapper:

```
<div class="room" id="kitchen">
  <div class="lightbulb on"></div>
  <div class="switch"></div>
  <div class="switch"></div>
  <div class="clapper"></div>
</div>
```

Triggering the clapper or either of the switches will change the state of the lightbulb. The switches and the clapper don't care what state the lightbulb is in; they just want to change the state.

Without custom events, you might write some code like this:

```
$( ".switch, .clapper" ).click(function() {
    var light = $( this ).closest( ".room" ).find( ".lightbulb" );
    if ( light.is( ".on" ) ) {
        light.removeClass( "on" ).addClass( "off" );
    } else {
        light.removeClass( "off" ).addClass( "on" );
    }
});
```

With custom events, your code might look more like this:

```
$( ".lightbulb" ).on( "light:toggle", function( event ) {
    var light = $( this );
    if ( light.is( ".on" ) ) {
        light.removeClass( "on" ).addClass( "off" );
    } else {
        light.removeClass( "off" ).addClass( "on" );
    }
});

$( ".switch, .clapper" ).click(function() {
    var room = $( this ).closest( ".room" );
    room.find( ".lightbulb" ).trigger( "light:toggle" );
});
```

This last bit of code is not that exciting, but something important has happened: we've moved the behavior of the lightbulb away from the switches and the clapper and to the lightbulb itself.

Let's make our example a little more interesting. We'll add another room to our house, along with a master switch, as shown here:

```
<div class="room" id="kitchen">
  <div class="lightbulb on"></div>
  <div class="switch"></div>
  <div class="switch"></div>
  <div class="clapper"></div>
```

```

</div>
<div class="room" id="bedroom">
  <div class="lightbulb on"></div>
  <div class="switch"></div>
  <div class="switch"></div>
  <div class="clapper"></div>
</div>
<div id="master_switch"></div>

```

If there are any lights on in the house, we want the master switch to turn all the lights off; otherwise, we want it to turn all lights on. To accomplish this, we'll add two more custom events to the lightbulbs: `light:on` and `light:off`. We'll make use of them in the `light:toggle` custom event, and use some logic to decide which one the master switch should trigger:

```

$( ".lightbulb" ).on( "light:toggle", function( event ) {
    var light = $( this );
    if ( light.is( ".on" ) ) {
        light.trigger( "light:off" );
    } else {
        light.trigger( "light:on" );
    }
}).on( "light:on", function( event ) {
    $( this ).removeClass( "off" ).addClass( "on" );
}).on( "light:off", function( event ) {
    $( this ).removeClass( "on" ).addClass( "off" );
});

$( ".switch, .clapper" ).click(function() {
    var room = $( this ).closest( ".room" );
    room.find( ".lightbulb" ).trigger( "light:toggle" );
});

$( "#master_switch" ).click(function() {
    var lightbulbs = $( ".lightbulb" );

    // Check if any lightbulbs are on
    if ( lightbulbs.is( ".on" ) ) {
        lightbulbs.trigger( "light:off" );
    } else {
        lightbulbs.trigger( "light:on" );
    }
});

```

Note how the behavior of the master switch is attached to the master switch; the behavior of a lightbulb belongs to the lightbulbs.

Naming Custom Events

You can use any name for a custom event, however you should beware of creating new events with names that might be used by future DOM events. For this reason, in this article we have chosen to use `light:` for all of our event names, as events with colons are unlikely to be used by a future DOM spec.

Recap: `.on()` and `.trigger()`

In the world of custom events, there are two important jQuery methods: `.on()` and `.trigger()`. In the [Events](#) chapter, we saw how to use these methods for working with user events; for this chapter, it's important to remember two things:

- `.on()` method takes an event type and an event handling function as arguments. Optionally, it can also receive event-related data as its second argument, pushing the event handling function to the third argument. Any data that is passed will be available to the event handling function in the `data` property of the event object. The event handling function always receives the event object as its first argument.
- `.trigger()` method takes an event type as its argument. Optionally, it can also take an array of values. These values will be passed to the event handling function as arguments after the event object.

Here is an example of the usage of `.on()` and `.trigger()` that uses custom data in both cases:

```

$( document ).on( "myCustomEvent", {
    foo: "bar"
}, function( event, arg1, arg2 ) {
    console.log( event.data.foo ); // "bar"
    console.log( arg1 );          // "bim"
    console.log( arg2 );          // "baz"
});

```

```
$( document ).trigger( "myCustomEvent", [ "bim", "baz" ] );
```

Conclusion

Custom events offer a new way of thinking about your code: they put the emphasis on the target of a behavior, not on the element that triggers it. If you take the time at the outset to spell out the pieces of your application, as well as the behaviors those pieces need to exhibit, custom events can provide a powerful way for you to "talk" to those pieces, either one at a time or en masse. Once the behaviors of a piece have been described, it becomes trivial to trigger those behaviors from anywhere, allowing for rapid creation of and experimentation with interface options. Finally, custom events can enhance code readability and maintainability, by making clear the relationship between an element and its behaviors.

Events

jQuery Event Extensions

jQuery offers several ways to extend its event system to provide custom functionality when events are attached to elements. Internally in jQuery, these extensions are primarily used to ensure that standard events such as `submit` and `change` behave consistently across browsers. However, they can also be used to define new events with custom behavior.

This document covers the extensions available starting with jQuery 1.7; a sparsely documented subset of this functionality has been available since jQuery 1.3 but the differences in functionality are extensive. For an overview of special events in earlier versions, see [Ben Alman's jQuery Special Events](#) article.

****Note:**** jQuery event extensions are an advanced feature; they require a deeper knowledge of both browser behavior and jQuery internals than most of the API. Most users of jQuery will not need to use event extensions, and those who do should use them with care. For example, on a large project with third-party plugins, changing the behavior of standard events such as `click` or `mouseover` can cause serious compatibility issues.

Events overview and general advice

When writing an event extension, it is essential to understand the flow of events through jQuery's internal event system. For a description of the event system from the API level, including a discussion of event delegation, see the [`.on\(\)`](#) method.

To simplify event management, jQuery only attaches a single event handler per element per event type (using `addEventListener` on W3C-compliant browsers or `attachEvent` on older IE) and then dispatches to event handlers that are attached via jQuery's APIs. For example, if three `click` event handlers are attached to an element, jQuery attaches its own handler when the first handler is attached and adds the user's event handler to a list to be executed when the event occurs. For subsequent event handlers, jQuery only adds them to its own internal list since it has already called the browser to attach its solitary handler. Conversely, jQuery removes its own event handler from the browser when the final event of a particular type is removed from the element.

An event can be a *native* event defined by the W3C and fired by the browser in response to something such as a user clicking a mouse button or pressing a key. It can also be a *custom* event, triggered only by code via jQuery's `.trigger()` or `.triggerHandler()` methods. Code can also trigger native browser events, which is convenient for simulating user actions.

In general, jQuery does not have intrinsic knowledge of whether an event name may be fired by a browser. So by default, jQuery always attaches an event to the browser when an API call is made to add an event handler for that event. If that event type is never generated by the browser, the only way the handler will be called is if JavaScript code triggers the event. Although there is generally no harm in attaching an unused event name to the browser, the default behavior can be overridden using the special event `setup` hook as described below.

Whenever elements are removed from a document via jQuery, the event system tries to ensure that events and related data are removed from the elements to prevent memory leaks. (Older versions of Internet Explorer are notorious for leaking memory in these situations if not managed carefully.) If an event extension attaches events or creates new objects, it should detach those objects or clear the data when the event is removed by defining `remove` and `teardown` hooks.

jQuery event extension developers should avoid using event names that have special meaning in a DOM setting. Event names such as `click`, `change`, or `load` have specific semantics defined by the W3C, so using them as custom events can cause unexpected behavior. Generally, jQuery event extensions should *only* be used for W3C-defined event names when the extension is normalizing behavior across browsers. A common convention to avoid collisions for custom events is to embed a colon or dash in the event type name, since no W3C events use those characters.

Although jQuery's event system is oriented towards delivering DOM events to DOM elements, jQuery methods can be used to attach and trigger events on plain objects. For example, it can be used as a simple publish/subscribe mechanism. Developers of event

extensions should attempt to avoid unwanted behavior if their extensions are used in a mixed scenario with DOM and plain objects. The canonical way that jQuery detects a DOM element is to check for `elem.nodeType === 1` on the object.

jQuery.event.props: Array

jQuery defines an [Event object](#) that represents a cross-browser subset of the information available when an event occurs. The `jQuery.event.props` property is an array of string names for properties that are always copied when jQuery processes a *native* browser event. (Events fired in code by `.trigger()` do not use this list, since the code can construct a `jQuery.Event` object with the needed values and trigger using that object.)

To add a property name to this list, use `jQuery.event.props.push("newPropertyName")`. However, be aware that every event processed by jQuery will now attempt to copy this property name from the native browser event to jQuery's constructed event. If the property does not exist for that event type, it will get an undefined value. Adding many properties to this list can significantly reduce event delivery performance, so for infrequently-needed properties it is more efficient to use the value directly from `event.originalEvent` instead. If properties must be copied, you are strongly advised to use `jQuery.event.fixHooks` as of version 1.7.

jQuery.event.fixHooks: Object

The `fixHooks` interface provides a per-event-type way to extend or normalize the event object that jQuery creates when it processes a *native* browser event. A `fixHooks` entry is an object that has two properties, each being optional:

props: Array: Strings representing properties that should be copied from the browser's event object to the jQuery event object. If omitted, no additional properties are copied beyond the standard ones that jQuery copies and normalizes (e.g. `event.target` and `event.relatedTarget`).

filter: Function(event, originalEvent): jQuery calls this function after it constructs the `jQuery.Event` object, copies standard properties from `jQuery.event.props`, and copies the `fixHooks`-specific props (if any) specified above. The function can create new properties on the event object or modify existing ones. The second argument is the browser's native event object, which is also available in `event.originalEvent`.

Note that for all events, the browser's native event object is available in `event.originalEvent`; if the jQuery event handler examines the properties there instead of jQuery's normalized event object, there is no need to create a `fixHooks` entry to copy or modify the properties.

For example, to set a hook for the "drop" event that copies the `dataTransfer` property, assign an object to `jQuery.event.fixHooks.drop`:

```
jQuery.event.fixHooks.drop = {
    props: [ "dataTransfer" ]
};
```

Since `fixHooks` is an advanced feature and rarely used externally, jQuery does not include code or interfaces to deal with conflict resolution. If there is a chance that some other code may be assigning `fixHooks` to the same events, the code should check for an existing hook and take appropriate measures. A simple solution might look like this:

```
if ( jQuery.event.fixHooks.drop ) {
    throw new Error( "Someone else took the jQuery.event.fixHooks.drop hook!" );
}

jQuery.event.fixHooks.drop = {
    props: [ "dataTransfer" ]
};
```

When there are known cases of different plugins wanting to attach to the drop hook, this solution might be more appropriate:

```
var existingHook = jQuery.event.fixHooks.drop;

if ( !existingHook ) {
    jQuery.event.fixHooks.drop = {
        props: [ "dataTransfer" ]
    };
} else {
```



```

        if ( existingHook.props ) {
            existingHook.props.push( "dataTransfer" );
        } else {
            existingHook.props = [ "dataTransfer" ];
        }
    }
}

```

Special event hooks

The jQuery special event hooks are a set of per-event-name functions and properties that allow code to control the behavior of event processing within jQuery. The mechanism is similar to `fixHooks` in that the special event information is stored in `jQuery.event.special.NAME`, where `NAME` is the name of the special event. Event names are case sensitive.

As with `fixHooks`, the special event hooks design assumes it will be very rare that two unrelated pieces of code want to process the same event name. Special event authors who need to modify events with existing hooks will need to take precautions to avoid introducing unwanted side-effects by clobbering those hooks.

noBubble: Boolean

Indicates whether this event type should be bubbled when the `.trigger()` method is called; by default it is `false`, meaning that a triggered event will bubble to the element's parents up to the document (if attached to a document) and then to the window. Note that defining `noBubble` on an event will effectively prevent that event from being used for delegated events with `.trigger()`.

bindType: String, delegateType: String

When defined, these string properties specify that a special event should be handled like another event type until the event is delivered. The `bindType` is used if the event is attached directly, and the `delegateType` is used for delegated events. These types are generally DOM event types, and *should not* be a special event themselves.

The behavior of these properties is easiest to see with an example. Assume a special event defined as follows:

```

jQuery.event.special.pushy = {
    bindType: "click",
    delegateType: "click"
};

```

When these properties are defined, the following behavior occurs in the jQuery event system:

- Event handlers for the "pushy" event are actually attached to "click" — both directly bound and delegated events.
- Special event hooks for "click" are called if they exist, *except* the `handle` hook for "pushy" is called when an event is delivered if one exists.
- Event handlers for "pushy" must be removed using the "pushy" event name, and are unaffected if "click" events are removed from the same elements.

So given the special event above, this code shows that a pushy isn't removed by removing clicks. That might be an effective way to defend against an ill-behaved plugin that didn't namespace its removal of click events, for example:

```

var elem = $( "p" );

elem.on( "click", function( event ) {
    $( "body" ).append( "I am a " + event.type + "!" );
});

elem.on( "pushy", function( event ) {
    $( "body" ).append( "I am pushy but still a " + event.type + "!" );
});

elem.trigger( "click" ); // Triggers both handlers

elem.off( "click" );

elem.trigger( "click" ); // Still triggers "pushy"

elem.off( "pushy" );

```

These two properties are often used in conjunction with a `handle` hook function; the hook might, for example, change the event name from "click" to "pushy" before calling event handlers. See below for an example.

The `handleObj` object

Many of the special event hook functions below are passed a `handleObj` object that provides more information about the event, how it was attached, and its current state. This object and its contents should be treated as read-only data, and only the properties below are documented for use by special event handlers. For the discussion below, assume an event is attached with this code:

```
$( ".dialog" ).on( "click.myPlugin", "button", {
  mydata: 42
}, myHandler );
```

type: String: The type of event, such as "click". When special event mapping is used via `bindType` or `delegateType`, this will be the mapped type.

origType: String: The original type name (in this case, "click") regardless of whether it was mapped via `bindType` or `delegateType`. So when a "pushy" event is mapped to "click" its `origType` would be "pushy". See the examples in those special event properties above for more detail.

namespace: String: Namespace(s), if any, provided when the event was attached, such as "myPlugin". When multiple namespaces are given, they are separated by periods and sorted in ascending alphabetical order. If no namespaces are provided, this property is an empty string.

selector: String: For delegated events, this is the selector used to filter descendant elements and determine if the handler should be called. In the example it is "button". For directly bound events, this property is `null`.

data: Object: The data, if any, passed to jQuery during event binding, e.g. { `myData`: 42 }. If the data argument was omitted or `undefined`, this property is `undefined` as well.

handler: function(event: `jQuery.Event`): Event handler function passed to jQuery during event binding; in the example it is a reference to `myHandler`. If `false` was passed during event binding, the handler refers to a single shared function that simply returns `false`.

setup: function(data: Object, namespaces, eventHandle: function)

The setup hook is called the first time an event of a particular type is attached to an element; this provides the hook an opportunity to do processing that will apply to all events of this type on this element. The `this` keyword will be a reference to the element where the event is being attached and `eventHandle` is jQuery's event handler function. In most cases the `namespaces` argument should not be used, since it only represents the namespaces of the *first* event being attached; subsequent events may not have this same namespaces.

This hook can perform whatever processing it desires, including attaching its own event handlers to the element or to other elements and recording setup information on the element using the `jQuery.data()` method. If the setup hook wants jQuery to add a browser event (via `addEventListener` or `attachEvent`, depending on browser) it should return `false`. In all other cases, jQuery will not add the browser event, but will continue all its other bookkeeping for the event. This would be appropriate, for example, if the event was never fired by the browser but invoked by `.trigger()`. To attach the jQuery event handler in the setup hook, use the `eventHandle` argument.

teardown: function()

The teardown hook is called when the final event of a particular type is removed from an element. The `this` keyword will be a reference to the element where the event is being cleaned up. This hook should return `false` if it wants jQuery to remove the event from the browser's event system (via `removeEventListener` or `detachEvent`). In most cases, the setup and teardown hooks should return the same value.

If the setup hook attached event handlers or added data to an element through a mechanism such as `jQuery.data()`, the teardown hook should reverse the process and remove them.

jQuery will generally remove the data and events when an element is totally removed from the document, but failing to remove data or events on teardown will cause a memory leak if the element stays in the document.

add: function(handleObj)

Each time an event handler is added to an element through an API such as `.on()`, jQuery calls this hook. The `this` keyword will be the element to which the event handler is being added, and the `handleObj` argument is as described in the section above. The return value of this hook is ignored.

remove: function(handleObj)

When an event handler is removed from an element using an API such as `.off()`, this hook is called. The `this` keyword will be the element where the handler is being removed, and the `handleObj` argument is as described in the section above. The return value of this hook is ignored.

trigger: function(event: jQuery.Event, data: Object)

Called when the `.trigger()` or `.triggerHandler()` methods are used to trigger an event for the special type from code, as opposed to events that originate from within the browser. The `this` keyword will be the element being triggered, and the event argument will be a `jQuery.Event` object constructed from the caller's input. At minimum, the event type, data, namespace, and target properties are set on the event. The data argument represents additional data passed by `.trigger()` if present.

The trigger hook is called early in the process of triggering an event, just after the `jQuery.Event` object is constructed and before any handlers have been called. It can process the triggered event in any way, for example by calling `event.stopPropagation()` or `event.preventDefault()` before returning. If the hook returns `false`, jQuery does not perform any further event triggering actions and returns immediately. Otherwise, it performs the normal trigger processing, calling any event handlers for the element and bubbling the event (unless propagation is stopped in advance or `noBubble` was specified for the special event) to call event handlers attached to parent elements.

_default: function(event: jQuery.Event, data: Object)

When the `.trigger()` method finishes running all the event handlers for an event, it also looks for and runs any method on the target object by the same name unless of the handlers called `event.preventDefault()`. So, `.trigger("submit")` will execute the `submit()` method on the element if one exists. When a `_default` hook is specified, the hook is called just prior to checking for and executing the element's default method. If this hook returns the value `false` the element's default method will be called; otherwise it is not.

handle: function(event: jQuery.Event, data: Object)

jQuery calls a handle hook when the event has occurred and jQuery would normally call the user's event handler specified by `.on()` or another event binding method. If the hook exists, jQuery calls it *instead of* that event handler, passing it the event and any data passed from `.trigger()` if it was not a native event. The `this` keyword is the DOM element being handled, and `event.handleObj` property has the detailed event information.

Based in the information it has, the handle hook should decide whether to call the original handler function which is in `event.handleObj.handler`. It can modify information in the event object before calling the original handler, but *must restore* that data before returning or subsequent unrelated event handlers may act unpredictably. In most cases, the handle hook should return the result of the original handler, but that is at the discretion of the hook. The handle hook is unique in that it is the only special event function hook that is called under its original special event name when the type is mapped using `bindType` and `delegateType`. For that reason, it is almost always an error to have anything other than a handle hook present if the special event defines a `bindType` and `delegateType`, since those other hooks will never be called.

Example: Multiclick event

This `multiclick` special event maps itself into a standard click event, but uses a handle hook so that it can monitor the event and only deliver it when the user clicks on the element a multiple of the number of times specified during event binding.

The hook stores the current click count in the data object, so multiclick handlers on different elements don't interfere with each other. It changes the event type to the original "multiclick" type before calling the handler and restores it to the mapped "click" type before returning:

```
jQuery.event.special.multiclick = {
    delegateType: "click",
    bindType: "click",
    handle: function( event ) {
        var handleObj = event.handleObj;
        var targetData = jQuery.data( event.target );
        var ret = null;

        // If a multiple of the click count, run the handler
        targetData.clicks = ( targetData.clicks || 0 ) + 1;

        if ( targetData.clicks % event.data.clicks === 0 ) {
            event.type = handleObj.origType;
            ret = handleObj.handler.apply( this, arguments );
            event.type = handleObj.type;
            return ret;
        }
    }
};

// Sample usage
$( "p" ).on( "multiclick", {
    clicks: 3
}, function( event ) {
    alert( "clicked 3 times" );
});
```

Effects

jQuery makes it trivial to add simple effects to your page. Effects can use the built-in settings, or provide a customized duration. You can also create custom animations of arbitrary CSS properties.

For complete details on jQuery effects, visit the [Effects documentation on api.jquery.com](https://api.jquery.com/effects/).

Effects

Introduction to Effects

Showing and Hiding Content

jQuery can show or hide content instantaneously with `.show()` or `.hide()`:

```
// Instantaneously hide all paragraphs
$( "p" ).hide();

// Instantaneously show all divs that have the hidden style class
$( "div.hidden" ).show();
```

When jQuery hides an element, it sets its CSS `display` property to `none`. This means the content will have zero width and height; it does not mean that the content will simply become transparent and leave an empty area on the page.

jQuery can also show or hide content by means of animation effects. You can tell `.show()` and `.hide()` to use animation in a couple of ways. One is to pass in an argument of `'slow'`, `'normal'`, or `'fast'`:

```
// Slowly hide all paragraphs
$( "p" ).hide( "slow" );

// Quickly show all divs that have the hidden style class
$( "div.hidden" ).show( "fast" );
```

If you prefer more direct control over the duration of the animation effect, you can pass the desired duration in milliseconds to `.show()` and `.hide()`:

```
// Hide all paragraphs over half a second
$( "p" ).hide( 500 );

// Show all divs that have the hidden style class over 1.25 seconds
$( "div.hidden" ).show( 1250 );
```

Most developers pass in a number of milliseconds to have more precise control over the duration.

Fade and Slide Animations

You may have noticed that `.show()` and `.hide()` use a combination of slide and fade effects when showing and hiding content in an animated way. If you would rather show or hide content with one effect or the other, there are additional methods that can help. `.slideDown()` and `.slideUp()` show and hide content, respectively, using only a slide effect. Slide animations are accomplished by rapidly making changes to an element's CSS `height` property.

```
// Hide all paragraphs using a slide up animation over 0.8 seconds
$( "p" ).slideUp( 800 );

// Show all hidden divs using a slide down animation over 0.6 seconds
$( "div.hidden" ).slideDown( 600 );
```

Similarly `.fadeIn()` and `.fadeOut()` show and hide content, respectively, by means of a fade animation. Fade animations involve rapidly making changes to an element's CSS `opacity` property.

```
// Hide all paragraphs using a fade out animation over 1.5 seconds
$( "p" ).fadeOut( 1500 );

// Show all hidden divs using a fade in animation over 0.75 seconds
$( "div.hidden" ).fadeIn( 750 );
```

Changing Display Based on Current Visibility State

jQuery can also let you change a content's visibility based on its current visibility state. `.toggle()` will show content that is currently hidden and hide content that is currently visible. You can pass the same arguments to `.toggle()` as you pass to any of the effects methods above.

```
// Instantaneously toggle the display of all paragraphs
$( "p" ).toggle();
```

```
// Slowly toggle the display of all images
$( "img" ).toggle( "slow" );

// Toggle the display of all divs over 1.8 seconds
$( "div" ).toggle( 1800 );
```

`.toggle()` will use a combination of slide and fade effects, just as `.show()` and `.hide()` do. You can toggle the display of content with just a slide or a fade using `.slideToggle()` and `.fadeToggle()`.

```
// Toggle the display of all ordered lists over 1 second using slide up/down
animations
$( "ol" ).slideToggle( 1000 );

// Toggle the display of all blockquotes over 0.4 seconds using fade in/out
animations
$( "blockquote" ).fadeToggle( 400 );
```

Doing Something After an Animation Completes

A common mistake when implementing jQuery effects is assuming that the execution of the next method in your chain will wait until the animation runs to completion.

```
// Fade in all hidden paragraphs; then add a style class to them (not quite right)
$( "p.hidden" ).fadeIn( 750 ).addClass( "lookAtMe" );
```

It is important to realize that `.fadeIn()` above only *kicks off* the animation. Once started, the animation is implemented by rapidly changing CSS properties in a JavaScript `setInterval()` loop. When you call `.fadeIn()`, it starts the animation loop and then returns the jQuery object, passing it along to `.addClass()` which will then add the `lookAtMe` style class while the animation loop is just getting started.

To defer an action until after an animation has run to completion, you need to use an animation callback function. You can specify your animation callback as the second argument passed to any of the animation methods discussed above. For the code snippet above, we can implement a callback as follows:

```
// Fade in all hidden paragraphs; then add a style class to them (correct with
animation callback)
$( "p.hidden" ).fadeIn( 750, function() {
    // this = DOM element which has just finished being animated
    $( this ).addClass( "lookAtMe" );
});
```

Note that you can use the keyword `this` to refer to the DOM element being animated. Also note that the callback will be called for each element in the jQuery object. This means that if your selector returns no elements, your animation callback will never run! You can solve this problem by testing whether your selection returned any elements; if not, you can just run the callback immediately.

```
// Run a callback even if there were no elements to animate
var someElement = $( "#nonexistent" );

var cb = function() {
    console.log( "done!" );
};

if ( someElement.length ) {
    someElement.fadeIn( 300, cb );
} else {
    cb();
}
```

Managing Animation Effects

jQuery provides some additional features for controlling your animations:

`.stop()`

`.stop()` will immediately terminate all animations running on the elements in your selection. You might give end-users control over page animations by rigging a button they can click to stop the animations.

```
// Create a button to stop all animations on the page:
$( "<button type='button'></button>" )
```

```

        .text( "Stop All Animations" )
        .on( "click", function() {
            $( "body *" ).filter( ":animated" ).stop();
        })
        .appendTo( document.body );

```

.delay()

`.delay()` is used to introduce a delay between successive animations. For example:

```

// Hide all level 1 headings over half a second; then wait for 1.5 seconds
// and reveal all level 1 headings over 0.3 seconds
$( "h1" ).hide( 500 ).delay( 1500 ).show( 300 );

```

jQuery.fx

The `jQuery.fx` object has a number of properties that control how effects are implemented. `jQuery.fx.speeds` maps the `slow`, `normal`, and `fast` duration arguments mentioned above to a specific number of milliseconds. The default value of `jQuery.fx.speeds` is:

```

{
    slow: 600,
    fast: 200,
    _default: 400 // Default speed, used for "normal"
}

```

You can modify any of these settings and even introduce some of your own:

```

jQuery.fx.speeds.fast = 300;
jQuery.fx.speeds.blazing = 100;
jQuery.fx.speeds.excruciating = 60000;

```

`jQuery.fx.interval` controls the number of frames per second that is displayed in an animation. The default value is 13 milliseconds between successive frames. You can set this to a lower value for faster browsers to make the animations run smoother. However this will mean more frames per second and thus a higher computational load for the browser, so you should be sure to test the performance implications of doing so thoroughly.

Finally, `jQuery.fx.off` can be set to `true` to disable all animations. Elements will immediately be set to the target final state instead. This can be especially useful when dealing with older browsers; you also may want to provide the option to disable all animations to your users.

```

$( "<button type='button'></button>" )
    .text( "Disable Animations" )
    .on( "click", function() {
        jQuery.fx.off = true;
    })
    .appendTo( document.body );

```


Effects

Custom Effects with .animate()

jQuery makes it possible to animate arbitrary CSS properties via the `.animate()` method. The `.animate()` method lets you animate to a set value, or to a value relative to the current value.

```
// Custom effects with .animate()
$( "div.funtimes" ).animate(
    {
        left: "+=50",
        opacity: 0.25
    },

    // Duration
    300,

    // Callback to invoke when the animation is finished
    function() {
        console.log( "done!" );
    }
);
```

Note: Color-related properties cannot be animated with `.animate()` using jQuery out of the box. Color animations can easily be accomplished by including the [color plugin](#). We'll discuss using plugins later in the book.

Easing

Definition: Easing describes the manner in which an effect occurs — whether the rate of change is steady, or varies over the duration of the animation. jQuery includes only two methods of easing: swing and linear. If you want more natural transitions in your animations, various easing plugins are available.

As of jQuery 1.4, it is possible to do per-property easing when using the `.animate()` method.

```
// Per-property easing
$( "div.funtimes" ).animate({
    left: [ "+=50", "swing" ],
    opacity: [ 0.25, "linear" ]
}, 300 );
```

For more details on easing options, see [Animation documentation on api.jquery.com](#).

Effects

Queue & Dequeue Explained

Queues are the foundation for all animations in jQuery, they allow a series functions to be executed asynchronously on an element. Methods such as `.slideUp()`, `.slideDown()`, `.fadeIn()`, and `.fadeOut()` all use `.animate()`, which leverages *queues* to build up the series of steps that will transition one or more CSS values throughout the duration of the animation.

We can pass a callback function to the `.animate()` method, which will execute once the animation has completed.

```
$( ".box" )
  .animate( {
    height: 20
  }, "slow", function() {
    $( "#title" ).html( "We're in the callback, baby!" );
  } );
```

Queues As Callbacks

Instead of passing a callback as an argument, we can add another function to the *queue* that will act as our callback. This will execute after all of the steps in the animation have completed.

```
$( ".box" )
  .animate( {
    height: 20
  }, "slow")
  .queue( function() {
    $( "#title" ).html( "We're in the animation, baby!" );

    // This tells jQuery to continue to the next item in the queue
    $( this ).dequeue();
  } );
```

In this example, the queued function will execute right after the animation.

jQuery does not have any insight into how the queue items function, so we need to call `.dequeue()`, which tells jQuery when to move to the next item in the queue.

Another way of *dequeuing* is by calling the function that is passed to your callback. That function will automatically call `.dequeue()` for you.

```
.queue( function( next ) {
  console.log( "I fired!" );
  next();
} );
```

Custom Queues

Up to this point all of our animation and queue examples have been using the default queue name which is `fx`. Elements can have multiple queues attached to them, and we can give each of these queues a different name. We can specify a custom queue name as the first argument to the `.queue()` method.

```
$( ".box" )
  .queue( "steps", function( next ) {
    console.log( "Step 1" );
    next();
  } )
  .queue( "steps", function( next ) {
    console.log( "Step 2" );
    next();
  } )
  .dequeue( "steps" );
```

Notice that we have to call the `.dequeue()` method passing it the name of our custom queue to start the execution. Every queue except for the default, `fx`, has to be manually kicked off by calling `.dequeue()` and passing it the name of the queue.

Clearing The Queue

Since queues are just a set of ordered operations, our application may have some logic in place that needs to prevent the remaining queue entries from executing. We can do this by calling the `.clearQueue()` method, which will empty the queue.

```
$( ".box" )
    .queue( "steps", function( next ) {
        console.log( "Will never log because we clear the queue" );
        next();
    } )
    .clearQueue( "steps" )
    .dequeue( "steps" );
```

In this example, nothing will happen as we removed everything from the `steps` queue.

Another way of clearing the queue is to call `.stop(true)`. That will stop the currently running animations and will clear the queue.

Replacing The Queue

When you pass an array of functions as the second argument to `.queue()`, that array will replace the queue.

```
$( ".box" )
    .queue( "steps", function( next ) {
        console.log( "I will never fire as we totally replace the queue" );
        next();
    } )
    .queue( "steps", [
        function( next ) {
            console.log( "I fired!" );
            next();
        }
    ] )
    .dequeue( "steps" );
```

You can also call `.queue()` without passing it functions, which will return the queue of that element as an array.

```
$( ".box" ).queue( "steps", function( next ) {
    console.log( "I fired!" );
    next();
} );

console.log( $( ".box" ).queue( "steps" ) );

$( ".box" ).dequeue( "steps" );
```

Ajax

Traditionally webpages required reloading to update their content. For web-based email this meant that users had to manually reload their inbox to check and see if they had new mail. This had huge drawbacks: it was slow and it required user input. When the user reloaded their inbox, the server had to reconstruct the entire web page and resend all of the HTML, CSS, JavaScript, as well as the user's email. This was hugely inefficient. Ideally, the server should only have to send the user's new messages, not the entire page. By 2003, all the major browsers solved this issue by adopting the XMLHttpRequest (XHR) object, allowing browsers to communicate with the server without requiring a page reload.

The XMLHttpRequest object is part of a technology called Ajax (Asynchronous JavaScript and XML). Using Ajax, data could then be passed between the browser and the server, using the XMLHttpRequest API, without having to reload the web page. With the widespread adoption of the XMLHttpRequest object it quickly became possible to build web applications like Google Maps, and Gmail that used XMLHttpRequest to get new map tiles, or new email without having to reload the entire page.

Ajax requests are triggered by JavaScript code; your code sends a request to a URL, and when it receives a response, a callback function can be triggered to handle the response. Because the request is asynchronous, the rest of your code continues to execute while the request is being processed, so it's imperative that a callback be used to handle the response.

Unfortunately, different browsers implement the Ajax API differently. Typically this meant that developers would have to account for all the different browsers to ensure that Ajax would work universally. Fortunately, jQuery provides Ajax support that abstracts away painful browser differences. It offers both a full-featured `$.ajax()` method, and simple convenience methods such as `$.get()`, `$.getScript()`, `$.getJSON()`, `$.post()`, and `$.load()`.

Most jQuery applications don't in fact use XML, despite the name "Ajax"; instead, they transport data as plain HTML or JSON (JavaScript Object Notation).

In general, Ajax does not work across domains. For instance, a webpage loaded from `example1.com` is unable to make an Ajax request to `example2.com` as it would violate the same origin policy. As a work around, JSONP (JSON with Padding) uses `` tags to load files containing arbitrary JavaScript content and JSON, from another domain. More recently browsers have implemented a technology called Cross-Origin Resource Sharing (CORS), that allows Ajax requests to different domains.

Ajax

Key Concepts

Proper use of Ajax-related jQuery methods requires understanding some key concepts first.

GET vs. POST

The two most common "methods" for sending a request to a server are GET and POST. It's important to understand the proper application of each.

The GET method should be used for non-destructive operations — that is, operations where you are only "getting" data from the server, not changing data on the server. For example, a query to a search service might be a GET request. GET requests may be cached by the browser, which can lead to unpredictable behavior if you are not expecting it. GET requests generally send all of their data in a query string.

The POST method should be used for destructive operations — that is, operations where you are changing data on the server. For example, a user saving a blog post should be a POST request. POST requests are generally not cached by the browser; a query string can be part of the URL, but the data tends to be sent separately as post data.

Data Types

jQuery generally requires some instruction as to the type of data you expect to get back from an Ajax request; in some cases the data type is specified by the method name, and in other cases it is provided as part of a configuration object. There are several options:

text

For transporting simple strings.

html

For transporting blocks of HTML to be placed on the page.

script

For adding a new script to the page.

json

For transporting JSON-formatted data, which can include strings, arrays, and objects.

Note: As of jQuery 1.4, if the JSON data sent by your server isn't properly formatted, the request may fail silently. See <http://json.org> for details on properly formatting JSON, but as a general rule, use built-in language methods for generating JSON on the server to avoid syntax issues.

jsonp

For transporting JSON data from another domain.

xml

For transporting data in a custom XML schema.

Consider using the JSON format in most cases, as it provides the most flexibility. It is especially useful for sending both HTML and data at the same time.

A is for Asynchronous

The asynchronicity of Ajax catches many new jQuery users off guard. Because Ajax calls are asynchronous by default, the response is not immediately available. Responses can only be

handled using a callback. So, for example, the following code will not work:

```
var response;

$.get( "foo.php", function( r ) {
    response = r;
});

console.log( response ); // undefined
```

Instead, we need to pass a callback function to our request; this callback will run when the request succeeds, at which point we can access the data that it returned, if any.

```
$.get( "foo.php", function( response ) {
    console.log( response ); // server response
});
```

Same-Origin Policy and JSONP

In general, Ajax requests are limited to the same protocol (http or https), the same port, and the same domain as the page making the request. This limitation does not apply to scripts that are loaded via jQuery's Ajax methods.

Note: Versions of Internet Explorer less than 10 do not support cross-domain AJAX requests.

The other exception is requests targeted at a JSONP service on another domain. In the case of JSONP, the provider of the service has agreed to respond to your request with a script that can be loaded into the page using a `<script>` tag, thus avoiding the same-origin limitation; that script will include the data you requested, wrapped in a callback function you provide.

Ajax and Firebug

Firebug (or the Webkit Inspector in Chrome or Safari) is an invaluable tool for working with Ajax requests. You can see Ajax requests as they happen in the Console tab of Firebug (and in the Resources > XHR panel of Webkit Inspector), and you can click on a request to expand it and see details such as the request headers, response headers, response content, and more. If something isn't going as expected with an Ajax request, this is the first place to look to track down what's wrong.

Ajax

jQuery's Ajax-Related Methods

While jQuery does offer many Ajax-related convenience methods, the core `$.ajax()` method is at the heart of all of them, and understanding it is imperative. We'll review it first, and then touch briefly on the convenience methods.

It's often considered good practice to use the `$.ajax()` method over the jQuery provided [convenience methods](#). As you'll see, it offers features that the convenience methods do not, and its syntax allows for the ease of readability.

`$.ajax()`

jQuery's core `$.ajax()` method is a powerful and straightforward way of creating Ajax requests. It takes a configuration object that contains all the instructions jQuery requires to complete the request. The `$.ajax()` method is particularly valuable because it offers the ability to specify both success and failure callbacks. Also, its ability to take a configuration object that can be defined separately makes it easier to write reusable code. For complete documentation of the configuration options, visit <http://api.jquery.com/jquery.ajax/>.

```
// Using the core $.ajax() method
$.ajax({

    // The URL for the request
    url: "post.php",

    // The data to send (will be converted to a query string)
    data: {
        id: 123
    },

    // Whether this is a POST or GET request
    type: "GET",

    // The type of data we expect back
    dataType : "json",
})
// Code to run if the request succeeds (is done);
// The response is passed to the function
.done(function( json ) {
    $( "<h1>" ).text( json.title ).appendTo( "body" );
    $( "<div class='content'>" ).html( json.html ).appendTo( "body" );
})
// Code to run if the request fails; the raw request and
// status codes are passed to the function
.fail(function( xhr, status, errorThrown ) {
    alert( "Sorry, there was a problem!" );
    console.log( "Error: " + errorThrown );
    console.log( "Status: " + status );
    console.dir( xhr );
})
// Code to run regardless of success or failure;
.always(function( xhr, status ) {
    alert( "The request is complete!" );
});
```

Note: Regarding the `dataType` setting, if the server sends back data that is in a different format than you specify, your code may fail, and the reason will not always be clear, because the HTTP response code will not show an error. When working with Ajax requests, make sure your server is sending back the data type you're asking for, and verify that the `Content-type` header is accurate for the data type. For example, for JSON data, the `Content-type` header should be `application/json`.

`$.ajax()` Options

There are many, many options for the `$.ajax()` method, which is part of its power. For a complete list of options, visit <http://api.jquery.com/jquery.ajax/>; here are several that you will use frequently:

async

Set to `false` if the request should be sent synchronously. Defaults to `true`. Note that if you set this option to `false`, your request will block execution of other code until the response is

received.

cache

Whether to use a cached response if available. Defaults to `true` for all `dataTypes` except "script" and "jsonp". When set to `false`, the URL will simply have a cachebusting parameter appended to it.

done

A callback function to run if the request succeeds. The function receives the response data (converted to a JavaScript object if the `dataType` was JSON), as well as the text status of the request and the raw request object.

fail

A callback function to run if the request results in an error. The function receives the raw request object and the text status of the request.

always

A callback function to run when the request is complete, regardless of success or failure. The function receives the raw request object and the text status of the request.

context

The scope in which the callback function(s) should run (i.e. what `this` will mean inside the callback function(s)). By default, `this` inside the callback function(s) refers to the object originally passed to `$.ajax()`.

data

The data to be sent to the server. This can either be an object or a query string, such as `foo=bar&baz=bim`.

dataType

The type of data you expect back from the server. By default, jQuery will look at the MIME type of the response if no `dataType` is specified.

jsonp

The callback name to send in a query string when making a JSONP request. Defaults to "callback".

timeout

The time in milliseconds to wait before considering the request a failure.

traditional

Set to `true` to use the param serialization style in use prior to jQuery 1.4. For details, see <http://api.jquery.com/jquery.param/>.

type

The type of the request, "POST" or "GET". Defaults to "GET". Other request types, such as "PUT" and "DELETE" can be used, but they may not be supported by all browsers.

url

The URL for the request.

The `url` option is the only required property of the `$.ajax()` configuration object; all other properties are optional. This can also be passed as the first argument to `$.ajax()`, and the options object as the second argument.

Convenience Methods

If you don't need the extensive configurability of `$.ajax()`, and you don't care about handling errors, the Ajax convenience functions provided by jQuery can be useful, terse ways to accomplish Ajax requests. These methods are just "wrappers" around the core `$.ajax()` method, and simply pre-set some of the options on the `$.ajax()` method.

The convenience methods provided by jQuery are:

`$.get`

Perform a GET request to the provided URL.

`$.post`

Perform a POST request to the provided URL.

`$.getScript`

Add a script to the page.

`$.getJSON`

Perform a GET request, and expect JSON to be returned.

In each case, the methods take the following arguments, in order:

`url`

The URL for the request. Required.

`data`

The data to be sent to the server. Optional. This can either be an object or a query string, such as `foo=bar&baz=bim`.

Note: This option is not valid for `$.getScript`.

`success callback`

A callback function to run if the request succeeds. Optional. The function receives the response data (converted to a JavaScript object if the data type was JSON), as well as the text status of the request and the raw request object.

`data type`

The type of data you expect back from the server. Optional.

Note: This option is only applicable for methods that don't already specify the data type in their name.

```
// Using jQuery's Ajax convenience methods

// Get plain text or HTML
$.get( "/users.php", {
    userId: 1234
}, function( resp ) {
    console.log( resp ); // server response
});

// Add a script to the page, then run a function defined in it
$.getScript( "/static/js/myScript.js", function() {
    functionFromMyScript();
});
```

```
// Get JSON-formatted data from the server
$.getJSON( "/details.php", function( resp ) {

    // Log each key in the response data
    $.each( resp, function( key, value ) {
        console.log( key + " : " + value );
    });

});
```

\$.fn.load

The `.load()` method is unique among jQuery's Ajax methods in that it is called on a selection. The `.load()` method fetches HTML from a URL, and uses the returned HTML to populate the selected element(s). In addition to providing a URL to the method, you can optionally provide a selector; jQuery will fetch only the matching content from the returned HTML.

```
// Using .load() to populate an element
$( "#newContent" ).load( "/foo.html" );

// Using .load() to populate an element based on a selector
$( "#newContent" ).load( "/foo.html #myDiv h1:first", function( html ) {
    alert( "Content updated!" );
});
```

Ajax

Ajax and Forms

jQuery's ajax capabilities can be especially useful when dealing with forms. There are several advantages, which can range from serialization, to simple client-side validation (e.g. "Sorry, that username is taken"), to [prefilters](#) (explained below), and even more!

Serialization

Serializing form inputs in jQuery is extremely easy. Two methods come supported natively: `.serialize()` and `.serializeArray()`. While the names are fairly self-explanatory, there are many advantages to using them.

The `.serialize()` method serializes a form's data into a query string. For the element's value to be serialized, it **must** have a `name` attribute. Please note that values from inputs with a type of checkbox or radio are included only if they are checked.

```
// Turning form data into a query string
$( "#myForm" ).serialize();

// Creates a query string like this:
// field_1=something&field2=somethingElse
```

While plain old serialization is great, sometimes your application would work better if you sent over an array of objects, instead of just the query string. For that, jQuery has the `.serializeArray()` method. It's very similar to the `.serialize()` method listed above, except it produces an array of objects, instead of a string.

```
// Creating an array of objects containing form data
$( "#myForm" ).serializeArray();

// Creates a structure like this:
// [
//   {
//     name : "field_1",
//     value : "something"
//   },
//   {
//     name : "field_2",
//     value : "somethingElse"
//   }
// ]
```

Client-side validation

Client-side validation is, much like many other things, extremely easy using jQuery. While there are several cases developers can test for, some of the most common ones are: presence of a required input, valid usernames/emails/phone numbers/etc..., or checking an "I agree..." box.

Please note that it is advisable that you also perform server-side validation for your inputs. However, it typically makes for a better user experience to be able to validate some things without submitting the form.

With that being said, let's jump on in to some examples! First, we'll see how easy it is to check if a required field doesn't have anything in it. If it doesn't, then we'll `return false`, and prevent the form from processing.

```
// Using validation to check for the presence of an input
$( "#form" ).submit(function( event ) {

    // If .required's value's length is zero
    if ( $( ".required" ).val().length === 0 ) {

        // Usually show some kind of error message here

        // Prevent the form from submitting
        event.preventDefault();
    } else {

        // Run $.ajax() here
    }
});
```

Let's see how easy it is to check for invalid characters in a phone number:

```

// Validate a phone number field
$( "#form" ).submit(function( event ) {
    var inputtedPhoneNumber = $( "#phone" ).val();

    // Match only numbers
    var phoneNumberRegex = /^\d*$/;

    // If the phone number doesn't match the regex
    if ( !phoneNumberRegex.test( inputtedPhoneNumber ) ) {

        // Usually show some kind of error message here

        // Prevent the form from submitting
        event.preventDefault();
    } else {

        // Run $.ajax() here
    }
});

```

Prefiltering

A prefilter is a way to modify the ajax options before each request is sent (hence, the name prefilter).

For example, say we would like to modify all cross-domain requests through a proxy. To do so with a prefilter is quite simple:

```

// Using a proxy with a prefilter
$.ajaxPrefilter(function( options, originalOptions, jqXHR ) {
    if ( options.crossDomain ) {
        options.url = "http://mydomain.net/proxy/" + encodeURIComponent(
options.url );
        options.crossDomain = false;
    }
});

```

You can pass in an optional argument before the callback function that specifies which `dataTypes` you'd like the prefilter to be applied to. For example, if we want our prefilter to only apply to `JSON` and `script` requests, we'd do:

```

// Using the optional dataType argument
$.ajaxPrefilter( "json script", function( options, originalOptions, jqXHR ) {

    // Do all of the prefiltering here, but only for
    // requests that indicate a dataType of "JSON" or "script"
});

```

Ajax

Working with JSONP

The advent of JSONP — essentially a consensual cross-site scripting hack — has opened the door to powerful mashups of content. Many prominent sites provide JSONP services, allowing you access to their content via a predefined API. A particularly great source of JSONP-formatted data is the [Yahoo! Query Language](#), which we'll use in the following example to fetch news about cats.

```
// Using YQL and JSONP
$.ajax({
  url: "http://query.yahooapis.com/v1/public/yql",

  // The name of the callback parameter, as specified by the YQL service
  jsonp: "callback",

  // Tell jQuery we're expecting JSONP
  dataType: "jsonp",

  // Tell YQL what we want and that we want JSON
  data: {
    q: "select title,abstract,url from search.news where query=\"cat\"",
    format: "json"
  },

  // Work with the response
  success: function( response ) {
    console.log( response ); // server response
  }
});
```

jQuery handles all the complex aspects of JSONP behind-the-scenes — all we have to do is tell jQuery the name of the JSONP callback parameter specified by YQL ("callback" in this case), and otherwise the whole process looks and feels like a normal Ajax request.

Ajax

Ajax Events

Often, you'll want to perform an operation whenever an Ajax request starts or stops, such as showing or hiding a loading indicator. Rather than defining this behavior inside every Ajax request, you can bind Ajax events to elements just like you'd bind other events. For a complete list of Ajax events, visit [Ajax Events documentation on docs.jquery.com](https://docs.jquery.com/Ajax/Events).

```
// Setting up a loading indicator using Ajax Events
$( "#loading_indicator" )
    .ajaxStart(function() {
        $( this ).show();
    })
    .ajaxStop(function() {
        $( this ).hide();
    });
```

Plugins

A jQuery plugin is simply a new method that we use to extend jQuery's prototype object. By extending the prototype object you enable all jQuery objects to inherit any methods that you add. As established, whenever you call `jQuery()` you're creating a new jQuery object, with all of jQuery's methods inherited.

The idea of a plugin is to do something with a collection of elements. You could consider each method that comes with the jQuery core a plugin, like `.fadeOut()` or `.addClass()`.

You can make your own plugins and use them privately in your code or you can release them into the wild. There are thousands of jQuery plugins available online. The barrier to creating a plugin of your own is so low that you'll want to do it straight away!

Plugins

Finding & Evaluating Plugins

One of the most celebrated aspects of jQuery is its extensive plugin ecosystem. From table sorting to form validation to autocompletion – if there's a need for it, chances are good that someone has written a plugin for it.

The quality of jQuery plugins varies widely. Many plugins are extensively tested and well-maintained, but others are hastily created and then ignored. More than a few fail to follow best practices. Some plugins, mainly [jQuery UI](#), are maintained by the jQuery team. The quality of these plugins is as good as jQuery itself.

The easiest way to find plugins is to search Google or the [jQuery Plugins Registry](#). Once you've identified some options, you may want to consult the [jQuery forums](#) or the #jquery IRC channel to get input from others.

When looking for a plugin to fill a need, do your homework. Ensure that the plugin is well-documented, and look for the author to provide lots of examples of its use. Be wary of plugins that do far more than you need; they can end up adding substantial overhead to your page. For more tips on spotting a sub-par plugin, read [Signs of a poorly written jQuery plugin](#) by Remy Sharp.

Once you choose a plugin, you'll need to add it to your page. Download the plugin, unzip it if necessary, place it within your application's directory structure, then include the plugin in your page using a script tag (after you include jQuery).

Plugins

How to Create a Basic Plugin

Sometimes you want to make a piece of functionality available throughout your code. For example, perhaps you want a single method you can call on a jQuery selection that performs a series of operations on the selection. In this case, you may want to write a plugin.

How jQuery Works 101: jQuery Object Methods

Before we write our own plugins, we must first understand a little about how jQuery works. Take a look at this code:

```
$( "a" ).css( "color", "red" );
```

This is some pretty basic jQuery code, but do you know what's happening behind the scenes? Whenever you use the `$` function to select elements, it returns a jQuery object. This object contains all of the methods you've been using (`.css()`, `.click()`, etc.) and all of the elements that fit your selector. The jQuery object gets these methods from the `$.fn` object. This object contains all of the jQuery object methods, and if we want to write our own methods, it will need to contain those as well.

Basic Plugin Authoring

Let's say we want to create a plugin that makes text within a set of retrieved elements green. All we have to do is add a function called `greenify` to `$.fn` and it will be available just like any other jQuery object method.

```
$.fn.greenify = function() {
    this.css( "color", "green" );
};

$( "a" ).greenify(); // Makes all the links green.
```

Notice that to use `.css()`, another method, we use `this`, not `$(this)`. This is because our `greenify` function is a part of the same object as `.css()`.

Chaining

This works, but there are a couple of things we need to do for our plugin to survive in the real world. One of jQuery's features is chaining, when you link five or six actions onto one selector. This is accomplished by having all jQuery object methods return the original jQuery object again (there are a few exceptions: `.width()` called without parameters returns the width of the selected element, and is not chainable). Making our plugin method chainable takes one line of code:

```
$.fn.greenify = function() {
    this.css( "color", "green" );
    return this;
}

$( "a" ).greenify().addClass( "greenified" );
```

Protecting the \$ Alias and Adding Scope

The `$` variable is very popular among JavaScript libraries, and if you're using another library with jQuery, you will have to make jQuery not use the `$` with `jQuery.noConflict()`. However, this will break our plugin since it is written with the assumption that `$` is an alias to the jQuery function. To work well with other plugins, *and* still use the jQuery `$` alias, we need to put all of our code inside of an [Immediately Invoked Function Expression](#), and then pass the function jQuery, and name the parameter `$`:

```
(function ( $ ) {

    $.fn.greenify = function() {
        this.css( "color", "green" );
        return this;
    };

})( jQuery );
```

In addition, the primary purpose of an Immediately Invoked Function is to allow us to have our own private variables. Pretend we want a different color green, and we want to store it in a variable.

```
(function ( $ ) {
    var shade = "#556b2f";

    $.fn.greenify = function() {
        this.css( "color", shade );
        return this;
    };
})( jQuery );
```

Minimizing Plugin Footprint

It's good practice when writing plugins to only take up one slot within `$.fn`. This reduces both the chance that your plugin will be overridden, and the chance that your plugin will override other plugins. In other words, this is bad:

```
(function( $ ) {
    $.fn.openPopup = function() {
        // Open popup code.
    };

    $.fn.closePopup = function() {
        // Close popup code.
    };
})( jQuery );
```

It would be much better to have one slot, and use parameters to control what action that one slot performs.

```
(function( $ ) {
    $.fn.popup = function( action ) {
        if ( action === "open" ) {
            // Open popup code.
        }

        if ( action === "close" ) {
            // Close popup code.
        }
    };
})( jQuery );
```

Using the `each()` Method

Your typical jQuery object will contain references to any number of DOM elements, and that's why jQuery objects are often referred to as collections. If you want to do any manipulating with specific elements (e.g. getting a data attribute, calculating specific positions) then you need to use `.each()` to loop through the elements.

```
$.fn.myNewPlugin = function() {
    return this.each(function() {
        // Do something to each element here.
    });
};
```

Notice that we return the results of `.each()` instead of returning `this`. Since `.each()` is already chainable, it returns `this`, which we then return. This is a better way to maintain chainability than what we've been doing so far.

Accepting Options

As your plugins get more and more complex, it's a good idea to make your plugin customizable by accepting options. The easiest way to do this, especially if there are lots of options, is with an object literal. Let's change our greenify plugin to accept some options.

```
(function ( $ ) {

    $.fn.greenify = function( options ) {

        // This is the easiest way to have default options.
        var settings = $.extend({
            // These are the defaults.
            color: "#556b2f",
            backgroundColor: "white"
        }, options );

        // Greenify the collection based on the settings variable.
        return this.css({
            color: settings.color,
            backgroundColor: settings.backgroundColor
        });

    };

})( jQuery );
```

Example usage:

```
$( "div" ).greenify({
    color: "orange"
});
```

The default value for `color` of `#556b2f` gets overridden by `$.extend()` to be orange.

Putting It Together

Here's an example of a small plugin using some of the techniques we've discussed:

```
(function( $ ) {

    $.fn.showLinkLocation = function() {

        this.filter( "a" ).each(function() {
            var link = $( this );
            link.append( " (" + link.attr( "href" ) + ")" );
        });

        return this;

    };

})( jQuery );

// Usage example:
$( "a" ).showLinkLocation();
```

This handy plugin goes through all anchors in the collection and appends the `href` attribute in parentheses.

```
<!-- Before plugin is called: -->
<a href="page.html">Foo</a>

<!-- After plugin is called: -->
<a href="page.html">Foo (page.html)</a>
```

Our plugin can be optimized though:

```
(function( $ ) {

    $.fn.showLinkLocation = function() {

        this.filter( "a" ).append(function() {
            return " (" + this.href + ")";
        });

        return this;

    };

})( jQuery );
```

We're using the `.append()` method's capability to accept a callback, and the return value of that callback will determine what is appended to each element in the collection. Notice also that we're not using the `.attr()` method to retrieve the `href` attribute, because the native DOM API gives us easy access with the aptly named `href` property.

Plugins

Advanced Plugin Concepts

Provide Public Access to Default Plugin Settings

An improvement we can, and should, make to the code above is to expose the default plugin settings. This is important because it makes it very easy for plugin users to override/customize the plugin with minimal code. And this is where we begin to take advantage of the function object.

```
// Plugin definition.
$.fn.highlight = function( options ) {

    // Extend our default options with those provided.
    // Note that the first argument to extend is an empty
    // object — this is to keep from overriding our "defaults" object.
    var opts = $.extend( {}, $.fn.highlight.defaults, options );

    // Our plugin implementation code goes here.

};

// Plugin defaults — added as a property on our plugin function.
$.fn.highlight.defaults = {
    foreground: "red",
    background: "yellow"
};
```

Now users can include a line like this in their scripts:

```
// This needs only be called once and does not
// have to be called from within a "ready" block
$.fn.highlight.defaults.foreground = "blue";
```

And now we can call the plugin method like this and it will use a blue foreground color:

```
$( "#myDiv" ).highlight();
```

As you can see, we've allowed the user to write a single line of code to alter the default foreground color of the plugin. And users can still selectively override this new default value when they want:

```
// Override plugin default foreground color.
$.fn.highlight.defaults.foreground = "blue";

// ...

// Invoke plugin using new defaults.
$( ".highlightDiv" ).highlight();

// ...

// Override default by passing options to plugin method.
$( "#green" ).highlight({
    foreground: "green"
});
```

Provide Public Access to Secondary Functions as Applicable

This item goes hand-in-hand with the previous item and is an interesting way to extend your plugin (and to let others extend your plugin). For example, the implementation of our plugin may define a function called "format" which formats the highlight text. Our plugin may now look like this, with the default implementation of the format method defined below the highlight function:

```
// Plugin definition.
$.fn.highlight = function( options ) {

    // Iterate and reformat each matched element.
    return this.each(function() {

        var elem = $( this );

        // ...

        var markup = elem.html();

        // Call our format function.
        markup = $.fn.highlight.format( markup );
```

```

        elem.html( markup );
    });

    };

    // Define our format function.
    $.fn.hilight.format = function( txt ) {
        return "<strong>" + txt + "</strong>";
    };

```

We could have just as easily supported another property on the options object that allowed a callback function to be provided to override the default formatting. That's another excellent way to support customization of your plugin. The technique shown here takes this a step further by actually exposing the format function so that it can be redefined. With this technique it would be possible for others to ship their own custom overrides of your plugin – in other words, it means others can write plugins for your plugin.

Considering the trivial example plugin we're building in this article, you may be wondering when this would ever be useful. One real-world example is the [Cycle Plugin](#). The Cycle Plugin is a slideshow plugin which supports a number of built-in transition effects – scroll, slide, fade, etc. But realistically, there is no way to define every single type of effect that one might wish to apply to a slide transition. And that's where this type of extensibility is useful. The Cycle Plugin exposes a "transitions" object to which users can add their own custom transition definitions. It's defined in the plugin like this:

```

$.fn.cycle.transitions = {
    // ...
};

```

This technique makes it possible for others to define and ship transition definitions that plug-in to the Cycle Plugin.

Keep Private Functions Private

The technique of exposing part of your plugin to be overridden can be very powerful. But you need to think carefully about what parts of your implementation to expose. Once it's exposed, you need to keep in mind that any changes to the calling arguments or semantics may break backward compatibility. As a general rule, if you're not sure whether to expose a particular function, then you probably shouldn't.

So how then do we define more functions without cluttering the namespace and without exposing the implementation? This is a job for closures. To demonstrate, we'll add another function to our plugin called "debug". The debug function will log the number of selected elements to the console. To create a closure, we wrap the entire plugin definition in a function (as detailed in the [jQuery Authoring Guidelines](#)).

```

// Create closure.
(function( $ ) {

    // Plugin definition.
    $.fn.hilight = function( options ) {
        debug( this );
        // ...
    };

    // Private function for debugging.
    function debug( obj ) {
        if ( window.console && window.console.log ) {
            window.console.log( "hilight selection count: " + obj.length );
        }
    };

    // ...

// End of closure.
})( jQuery );

```

Our "debug" method cannot be accessed from outside of the closure and thus is private to our implementation.

Bob and Sue

Let's say Bob has created a wicked new gallery plugin (called "superGallery") which takes a list of images and makes them navigable. Bob's thrown in some animation to make it more interesting. He's tried to make the plugin as customizable as possible, and has ended up with something like this:

```
jQuery.fn.superGallery = function( options ) {

    // Bob's default settings:
    var defaults = {
        textColor: "#000",
        backgroundColor: "#fff",
        fontSize: "1em",
        delay: "quite long",
        getTextFromTitle: true,
        getTextFromRel: false,
        getTextFromAlt: false,
        animateWidth: true,
        animateOpacity: true,
        animateHeight: true,
        animationDuration: 500,
        clickImgToGoToNext: true,
        clickImgToGoToLast: false,
        nextButtonText: "next",
        previousButtonText: "previous",
        nextButtonTextColor: "red",
        previousButtonTextColor: "red"
    };

    var settings = $.extend( {}, defaults, options );

    return this.each(function() {
        // Plugin code would go here...
    });

};
```

The first thing that probably comes to your mind (OK, maybe not the first) is the prospect of how huge this plugin must be to accommodate such a level of customization. The plugin, if it weren't fictional, would probably be a lot larger than necessary. There are only so many kilobytes people will be willing to spend!

Now, our friend Bob thinks this is all fine; in fact, he's quite impressed with the plugin and its level of customization. He believes that all the options make for a more versatile solution, one which can be used in many different situations.

Sue, another friend of ours, has decided to use this new plugin. She has set up all of the options required and now has a working solution sitting in front of her. It's only five minutes later, after playing with the plugin, that she realizes the gallery would look much nicer if each image's width were animated at a slower speed. She hastily searches through Bob's documentation but finds no *animateWidthDuration* option!

Do You See The Problem?

It's not really about how many options your plugin has; but what options it has!

Bob has gone a little over the top. The level of customization he's offering, while it may seem high, is actually quite low, especially considering all the possible things one might want to control when using this plugin. Bob has made the mistake of offering a lot of ridiculously specific options, rendering his plugin much more difficult to customize!

A Better Model

So it's pretty obvious: Bob needs a new customization model, one which does not relinquish control or abstract away the necessary details.

The reason Bob is so drawn to this high-level simplicity is that the jQuery framework very much lends itself to this mindset. Offering a *previousButtonTextColor* option is nice and simple, but let's face it, the vast majority of plugin users are going to want way more control!

Here are a few tips which should help you create a better set of customizable options for your plugins:

Don't Create Plugin-specific Syntax

Developers who use your plugin shouldn't have to learn a new language or terminology just to get the job done.

Bob thought he was offering maximum customization with his *delay* option (look above). He made it so that with his plugin you can specify four different delays, "quite short," "very short," "quite long," or "very long":

```
var delayDuration = 0;
switch ( settings.delay ) {
    case "very short":
        delayDuration = 100;
        break;

    case "quite short":
        delayDuration = 200;
        break;

    case "quite long":
        delayDuration = 300;
        break;

    case "very long":
        delayDuration = 400;
        break;

    default:
        delayDuration = 200;
}
}
```

Not only does this limit the level of control people have, but it takes up quite a bit of space. Twelve lines of code just to define the delay time is a bit much, don't you think? A better way to construct this option would be to let plugin users specify the amount of time (in milliseconds) as a number, so that no processing of the option needs to take place.

The key here is not to diminish the level of control through your abstraction. Your abstraction, whatever it is, can be as simplistic as you want, but make sure that people who use your plugin will still have that much-sought-after low-level control! (By low-level I mean non-abstracted.)

Give Full Control of Elements

If your plugin creates elements to be used within the DOM, then it's a good idea to offer plugin users some way to access those elements. Sometimes this means giving certain elements IDs or classes. But note that your plugin shouldn't rely on these hooks internally:

A bad implementation:

```
// Plugin code
$( "<div class='gallery-wrapper' />" ).appendTo( "body" );

$( ".gallery-wrapper" ).append( "..." );
```

To allow users to access and even manipulate that information, you can store it in a variable containing the settings of your plugin. A better implementation of the previous code is shown below:

```
// Retain an internal reference:
var wrapper = $( "<div />" )
    .attr( settings.wrapperAttrs )
    .appendTo( settings.container );

// Easy to reference later...
wrapper.append( "..." );
```

Notice that we've created a reference to the injected wrapper and we're also calling the `.attr()` method to add any specified attributes to the element. So, in our settings it might be handled like this:

```
var defaults = {
    wrapperAttrs : {
        class: "gallery-wrapper"
    },
    // ... rest of settings ...
};

// We can use the extend method to merge options/settings as usual:
```

```
// But with the added first parameter of TRUE to signify a DEEP COPY:
var settings = $.extend( true, {}, defaults, options );
```

The `$.extend()` method will now recurse through all nested objects to give us a merged version of both the defaults and the passed options, giving the passed options precedence.

The plugin user now has the power to specify any attribute of that wrapper element so if they require that there be a hook for any CSS styles then they can quite easily add a class or change the name of the ID without having to go digging around in plugin source.

The same model can be used to let the user define CSS styles:

```
var defaults = {
    wrapperCSS: {},
    // ... rest of settings ...
};

// Later on in the plugin where we define the wrapper:
var wrapper = $( "<div />" )
    .attr( settings.wrapperAttrs )
    .css( settings.wrapperCSS ) // ** Set CSS!
    .appendTo( settings.container );
```

Your plugin may have an associated stylesheet where developers can add CSS styles. Even in this situation it's a good idea to offer some convenient way of setting styles in JavaScript, without having to use a selector to get at the elements.

Provide Callback Capabilities

What is a callback? – A callback is essentially a function to be called later, normally triggered by an event. It's passed as an argument, usually to the initiating call of a component, in this case, a jQuery plugin.

If your plugin is driven by events then it might be a good idea to provide a callback capability for each event. Plus, you can create your own custom events and then provide callbacks for those. In this gallery plugin it might make sense to add an "onImageShow" callback.

```
var defaults = {

    // We define an empty anonymous function so that
    // we don't need to check its existence before calling it.
    onImageShow : function() {},

    // ... rest of settings ...

};

// Later on in the plugin:
nextButton.on( "click", showNextImage );

function showNextImage() {

    // Returns reference to the next image node
    var image = getNextImage();

    // Stuff to show the image here...

    // Here's the callback:
    settings.onImageShow.call( image );

}
```

Instead of initiating the callback via traditional means (adding parenthesis) we're calling it in the context of `image` which will be a reference to the image node. This means that you have access to the actual image node through the `this` keyword within the callback:

```
$( "ul.imgs li" ).superGallery({
    onImageShow: function() {
        $( this ).after( "<span>" + $( this ).attr( "longdesc" ) + "</span>"
    );
    },
    // ... other options ...
});
```

Similarly you could add an "onImageHide" callback and numerous other ones. The point of callbacks is to give plugin users an easy way to add additional functionality without digging around in the source.

Remember, It's a Compromise

Your plugin is not going to be able to work in every situation. And equally, it's not going to be very useful if you offer no or very few methods of control. So, remember, it's always going to be a compromise. Three things you must always take into account are:

- *Flexibility*: How many situations will your plugin be able to deal with?
- *Size*: Does the size of your plugin correspond to its level of functionality? I.e. Would you use a very basic tooltip plugin if it was 20k in size? – Probably not!
- *Performance*: Does your plugin heavily process the options in any way? Does this affect speed? Is the overhead caused worth it for the end user?

Plugins

Writing Stateful Plugins with the jQuery UI Widget Factory

Writing Stateful Plugins with the jQuery UI Widget Factory

While most existing jQuery plugins are stateless – that is, we call them on an element and that is the extent of our interaction with the plugin – there's a large set of functionality that doesn't fit into the basic plugin pattern.

In order to fill this gap, jQuery UI has implemented a more advanced plugin system. The new system manages state, allows multiple functions to be exposed via a single plugin, and provides various extension points. This system is called the Widget Factory and is exposed as `jQuery.widget` as part of jQuery UI 1.8; however, it can be used independently of jQuery UI.

To demonstrate the capabilities of the Widget Factory, we'll build a simple progress bar plugin.

To start, we'll create a progress bar that just lets us set the progress once. As we can see below, this is done by calling `jQuery.widget` with two parameters: the name of the plugin to create and an object literal containing functions to support our plugin.

When our plugin gets called, it will create a new plugin instance and all functions will be executed within the context of that instance. This is different from a standard jQuery plugin in two important ways. First, the context is an object, not a DOM element. Second, the context is always a single object, never a collection.

A simple, stateful plugin using the jQuery UI Widget Factory:

```
$.widget( "nmk.progressbar", {
    _create: function() {
        var progress = this.options.value + "%";
        this.element.addClass( "progressbar" ).text( progress );
    }
});
```

The name of the plugin must contain a namespace; in this case we've used the `nmk` namespace. There is a limitation that namespaces be exactly one level deep – that is, we can't use a namespace like `nmk.foo`. We can also see that the Widget Factory has provided two properties for us. `this.element` is a jQuery object containing exactly one element. If our plugin is called on a jQuery object containing multiple elements, a separate plugin instance will be created for each element, and each instance will have its own `this.element`. The second property, `this.options`, is a hash containing key/value pairs for all of our plugin's options. These options can be passed to our plugin as shown here.

Note: In our example we use the `nmk` namespace. The `ui` namespace is reserved for official jQuery UI plugins. When building your own plugins, you should create your own namespace. This makes it clear where the plugin came from and whether it is part of a larger collection.

Passing options to a widget:

```
$( "<div />" ).appendTo( "body" ).progressbar({ value: 20 });
```

When we call `jQuery.widget` it extends jQuery by adding a method to `jQuery.fn` (the same way we'd create a standard plugin). The name of the function it adds is based on the name you pass to `jQuery.widget`, without the namespace; in our case it will create `jQuery.fn.progressbar`. The options passed to our plugin get set in `this.options` inside of our plugin instance.

As shown below, we can specify default values for any of our options. When designing your API, you should figure out the most common use case for your plugin so that you can set appropriate default values and make all options truly optional.

Setting default options for a widget:

```
$.widget( "nmk.progressbar", {
    // Default options.
    options: {
        value: 0
    },
    _create: function() {
        var progress = this.options.value + "%";
        this.element.addClass( "progressbar" ).text( progress );
    }
});
```

Adding Methods to a Widget

Now that we can initialize our progress bar, we'll add the ability to perform actions by calling methods on our plugin instance. To define a plugin method, we just include the function in the object literal that we pass to `jQuery.widget`. We can also define "private" methods by prepending an underscore to the function name.

Creating widget methods:

```
$.widget( "nmk.progressbar", {
    options: {
        value: 0
    },
    _create: function() {
        var progress = this.options.value + "%";
        this.element.addClass( "progressbar" ).text( progress );
    },
    // Create a public method.
    value: function( value ) {

        // No value passed, act as a getter.
        if ( value === undefined ) {

            return this.options.value;

        // Value passed, act as a setter.
        } else {

            this.options.value = this._constrain( value );
            var progress = this.options.value + "%";
            this.element.text( progress );

        }

    },
    // Create a private method.
    _constrain: function( value ) {

        if ( value > 100 ) {
            value = 100;
        }

        if ( value < 0 ) {
            value = 0;
        }

        return value;
    }
});
```

To call a method on a plugin instance, you pass the name of the method to the jQuery plugin. If you are calling a method that accepts parameters, you simply pass those parameters after the method name.

Calling methods on a plugin instance:

```
var bar = $( "<div />" ).appendTo( "body" ).progressbar( { value: 20 } );

// Get the current value.
alert( bar.progressbar( "value" ) );

// Update the value.
bar.progressbar( "value", 50 );

// Get the current value again.
alert( bar.progressbar( "value" ) );
```

Note: Executing methods by passing the method name to the same jQuery function that was used to initialize the plugin may seem odd. This is done to prevent pollution of the jQuery namespace while maintaining the ability to chain method calls.

Working with Widget Options

One of the methods that is automatically available to our plugin is the option method. The option method allows you to get and set options after initialization. This method works exactly like jQuery's `.css()` and `.attr()` methods: you can pass just a name to use it as a getter, a name and value to use it as a single setter, or a hash of name/value pairs to set multiple values. When used as a getter, the plugin will return the current value of the option that corresponds to the name that was passed in. When used as a setter, the plugin's `_setOption` method will be called for each option that is being set. We can specify a `_setOption` method in our plugin to react to option changes.

Responding when an option is set:

```
$.widget( "nmk.progressbar", {
    options: {
        value: 0
    },
    _create: function() {
        this.element.addClass( "progressbar" );
        this._update();
    },
    _setOption: function( key, value ) {
        this.options[ key ] = value;
        this._update();
    },
    _update: function() {
        var progress = this.options.value + "%";
        this.element.text( progress );
    }
});
```

Adding Callbacks

One of the easiest ways to make your plugin extensible is to add callbacks so users can react when the state of your plugin changes. We can see below how to add a callback to our progress bar to signify when the progress has reached 100%. The `_trigger` method takes three parameters: the name of the callback, a native event object that initiated the callback, and a hash of data relevant to the event. The callback name is the only required parameter, but the others can be very useful for users who want to implement custom functionality on top of your plugin. For example, if we were building a draggable plugin, we could pass the native `mousemove` event when triggering a drag callback; this would allow users to react to the drag based on the x/y coordinates provided by the event object.

Providing callbacks for user extension:

```
$.widget( "nmk.progressbar", {
    options: {
        value: 0
    },
    _create: function() {
        this.element.addClass( "progressbar" );
        this._update();
    },
    _setOption: function( key, value ) {
        this.options[ key ] = value;
        this._update();
    },
    _update: function() {
        var progress = this.options.value + "%";
        this.element.text( progress );
        if ( this.options.value == 100 ) {
            this._trigger( "complete", null, { value: 100 } );
        }
    }
});
```

Callback functions are essentially just additional options, so you can get and set them just like any other option. Whenever a callback is executed, a corresponding event is triggered as well. The event type is determined by concatenating the plugin name and the callback name. The callback and event both receive the same two parameters: an event object and a hash of data relevant to the event, as we'll see below.

If your plugin has functionality that you want to allow the user to prevent, the best way to support this is by creating cancelable callbacks. Users can cancel a callback, or its associated event, the same way they cancel any native event: by calling `event.preventDefault()` or using `return false`. If the user cancels the callback, the `_trigger` method will return false so you can implement the appropriate functionality within your plugin.

Binding to widget events:

```
var bar = $( "<div />" ).appendTo( "body" ).progressbar({
    complete: function( event, data ) {
        alert( "Callbacks are great!" );
    }
}).bind( "progressbarcomplete", function( event, data ) {
    alert( "Events bubble and support many handlers for extreme flexibility." );
    alert( "The progress bar value is " + data.value );
});
bar.progressbar( "option", "value", 100 );
```

The Widget Factory: Under the Hood

When you call `jquery.widget`, it creates a constructor function for your plugin and sets the object literal that you pass in as the prototype for your plugin instances. All of the functionality that automatically gets added to your plugin comes from a base widget prototype, which is defined as `jquery.widget.prototype`. When a plugin instance is created, it is stored on the original DOM element using `jquery.data`, with the plugin's full name (the plugin's namespace, plus a hyphen, plus the plugin's name) as the key. For example the jQuery UI dialog widget uses a key of `"ui-dialog"`.

Because the plugin instance is directly linked to the DOM element, you can access the plugin instance directly instead of going through the exposed plugin method if you want. This will allow you to call methods directly on the plugin instance instead of passing method names as strings and will also give you direct access to the plugin's properties.

```
var bar = $( "<div />" )
    .appendTo( "body" )
    .progressbar()
    .data( "nmk-progressbar" );

// Call a method directly on the plugin instance.
bar.option( "value", 50 );

// Access properties on the plugin instance.
alert( bar.options.value );
```

One of the biggest benefits of having a constructor and prototype for a plugin is the ease of extending the plugin. By adding or modifying methods on the plugin's prototype, we can modify the behavior of all instances of our plugin. For example, if we wanted to add a method to our progress bar to reset the progress to 0% we could add this method to the prototype and it would instantly be available to be called on any plugin instance.

```
$.nmk.progressbar.prototype.reset = function() {
    this._setOption( "value", 0 );
};
```

Cleaning Up

In some cases, it will make sense to allow users to apply and then later unapply your plugin. You can accomplish this via the `_destroy` method. Within the `_destroy` method, you should undo anything your plugin may have done during initialization or later use. The `_destroy` method is automatically called if the element that your plugin instance is tied to is removed from the DOM, so this can be used for garbage collection as well. The base `destroy` method calls `_destroy` on the plugin instance.

Adding a destroy method to a widget:

```
$.widget( "nmk.progressbar", {
    options: {
        value: 0
    },
    _create: function() {
        this.element.addClass( "progressbar" );
        this._update();
    },
    _setOption: function( key, value ) {
        this.options[ key ] = value;
        this._update();
    },
    _update: function() {
        var progress = this.options.value + "%";
        this.element.text( progress );
        if ( this.options.value === 100 ) {
            this._trigger( "complete", null, { value: 100 } );
        }
    },
    _destroy: function() {
        this.element
            .removeClass( "progressbar" )
            .text( "" );
    }
});
```

Conclusion

The Widget Factory is only one way of creating stateful plugins. There are a few different models that can be used and each have their own advantages and disadvantages. The Widget Factory solves lots of common problems for you and can greatly improve productivity, it also greatly improves code reuse, making it a great fit for jQuery UI as well as many other stateful plugins.

Performance

Performance

Append Outside of Loops

Touching the DOM comes at a cost. If you're appending a lot of elements to the DOM, you will want to append them all at once, rather than one at a time. This is a common problem when appending elements within a loop.

```
$.each( myArray, function( i, item ) {
    var newListItem = "<li>" + item + "</li>";
    $( "#ballers" ).append( newListItem );
});
```

One common technique is to leverage a document fragment. During each iteration of the loop, you append the element to the fragment rather than the DOM element. After the loop, just append the fragment to the DOM element.

```
var frag = document.createDocumentFragment();
$.each( myArray, function( i, item ) {
    var newListItem = document.createElement( "li" );
    var itemText = document.createTextNode( item );
    newListItem.appendChild( itemText );
    frag.appendChild( newListItem );
});
$( "#ballers" )[ 0 ].appendChild( frag );
```

Another simple technique is to build up a string during each iteration of the loop. After the loop, just set the HTML of the DOM element to that string.

```
var myHtml = "";
$.each( myArray, function( i, item ) {
    myHtml += "<li>" + item + "</li>";
});
$( "#ballers" ).html( myHtml );
```

There are of course other techniques you could certainly test out. A great way to test the performance of these is through a site called [jsperf](http://jsperf.com). This site allows you to benchmark each technique and visually see how it performs across all the browsers.

Performance

Cache Length During Loops

In a for loop, don't access the length property of an array every time; cache it beforehand.

```
var myLength = myArray.length;
for ( var i = 0; i < myLength; i++ ) {
    // do stuff
}
```

Performance

Detach Elements to Work with Them

The DOM is slow; you want to avoid manipulating it as much as possible. jQuery introduced `detach()` in version 1.4 to help address this issue, allowing you to remove an element from the DOM while you work with it.

```
var table = $( "#myTable" );
var parent = table.parent();

table.detach();

// ... add lots and lots of rows to table

parent.append( table );
```

Performance

Don't Act on Absent Elements

jQuery won't tell you if you're trying to run a whole lot of code on an empty selection – it will proceed as though nothing's wrong. It's up to you to verify that your selection contains some elements.

```
// Bad: This runs three functions before it
// realizes there's nothing in the selection
$( "#nosuchthing" ).slideUp();

// Better:
var elem = $( "#nosuchthing" );

if ( elem.length ) {

    elem.slideUp();

}

// Best: Add a doOnce plugin.
jQuery.fn.doOnce = function( func ) {

    this.length && func.apply( this );

    return this;

}

$( "li.cartitems" ).doOnce(function() {

    // make it ajax! \o/

});
```

This guidance is especially applicable for jQuery UI widgets, which have a lot of overhead even when the selection doesn't contain elements.

Performance

Optimize Selectors

Selector optimization is less important than it used to be, as more browsers implement `document.querySelector()` and the burden of selection shifts from jQuery to the browser. However, there are still some tips to keep in mind when selector performance becomes a bottleneck.

jQuery Extensions

When possible, avoid selectors that include [jQuery extensions](#). These extensions cannot take advantage of the performance boost provided by the native `querySelectorAll()` DOM method and, therefore, require the use of the Sizzle selector engine provided by jQuery.

```
// Slower (the zero-based :even selector is a jQuery extension)
$( "#my-table tr:even" );

// Better, though not exactly equivalent
$( "#my-table tr:nth-child(odd)" );
```

Keep in mind that many jQuery extensions, including `:even` in the above example, do not have exact equivalents in the CSS specification. In some situations the convenience of these extensions could outweigh their performance cost.

Avoid Excessive Specificity

```
$( ".data table.attendees td.gonzalez" );

// Better: Drop the middle if possible.
$( ".data td.gonzalez" );
```

A "flatter" DOM also helps improve selector performance, as the selector engine has fewer layers to traverse when looking for an element.

ID-Based Selectors

Beginning your selector with an ID is a safe bet.

```
// Fast:
$( "#container div.robotarm" );

// Super-fast:
$( "#container" ).find( "div.robotarm" );
```

With the first approach, jQuery queries the DOM using `document.querySelectorAll()`. With the second, jQuery uses `document.getElementById()`, which is faster, although the speed improvement may be diminished by the subsequent call to `.find()`.

Tips for Older Browsers

When support for older browsers, such as Internet Explorer 8 and below, is necessary, consider the following tips:

Specificity

Be specific on the right-hand side of your selector, and less specific on the left.

```
// Unoptimized:
$( "div.data .gonzalez" );

// Optimized:
$( ".data td.gonzalez" );
```

Use `tag.class` if possible on your right-most selector, and just `tag` or just `.class` on the left.

Avoid the Universal Selector

Selections that specify or imply that a match could be found anywhere can be very slow.

```
$( ".buttons > *" ); // Extremely expensive.  
$( ".buttons" ).children(); // Much better.  
  
$( ":radio" ); // Implied universal selection.  
$( "*:radio" ); // Same thing, explicit now.  
$( "input:radio" ); // Much better.
```

Performance

Use Stylesheets for Changing CSS on Many Elements

If you're changing the CSS of more than 20 elements using `.css()`, consider adding a style tag to the page instead for a nearly 60% increase in speed.

```
// Fine for up to 20 elements, slow after that:
$( "a.swedberg" ).css( "color", "#0769ad" );

// Much faster:
$( "<style type='text/css'>a.swedberg { color: #0769ad }</style>" )
  .appendTo( "head" );
```

Performance

Don't Treat jQuery as a Black Box

Use the source as your documentation. Bookmark [the source code](#) and refer to it often.

Code Organization

Understanding the basic mechanics is one thing, but the essence of building applications is understanding how to organize code so that it is navigable and well-encapsulated instead of a whole slew of global functions.

Code Organization

Code Organization Concepts

When you move beyond adding simple enhancements to your website with jQuery and start developing full-blown client-side applications, you need to consider how to organize your code. In this chapter, we'll take a look at various code organization patterns you can use in your jQuery application and explore the [RequireJS](#) dependency management and build system.

Key Concepts

Before we jump into code organization patterns, it's important to understand some concepts that are common to all good code organization patterns.

- Your code should be divided into units of functionality — modules, services, etc. Avoid the temptation to have all of your code in one huge `$(document).ready()` block. This concept, loosely, is known as encapsulation.
- Don't repeat yourself. Identify similarities among pieces of functionality, and use inheritance techniques to avoid repetitive code.
- Despite jQuery's DOM-centric nature, JavaScript applications are not all about the DOM. Remember that not all pieces of functionality need to — or should — have a DOM representation.
- Units of functionality should be [loosely coupled](#), that is, a unit of functionality should be able to exist on its own, and communication between units should be handled via a messaging system such as custom events or pub/sub. Stay away from direct communication between units of functionality whenever possible.

The concept of loose coupling can be especially troublesome to developers making their first foray into complex applications, so be mindful of this as you're getting started.

Encapsulation

The first step to code organization is separating pieces of your application into distinct pieces; sometimes, even just this effort is sufficient to lend

The Object Literal

An object literal is perhaps the simplest way to encapsulate related code. It doesn't offer any privacy for properties or methods, but it's useful for eliminating anonymous functions from your code, centralizing configuration options, and easing the path to reuse and refactoring.

```
// An object literal
var myFeature = {
  myProperty: "hello",

  myMethod: function() {
    console.log( myFeature.myProperty );
  },

  init: function( settings ) {
    myFeature.settings = settings;
  },

  readSettings: function() {
    console.log( myFeature.settings );
  }
};

myFeature.myProperty === "hello"; // true

myFeature.myMethod(); // "hello"

myFeature.init({
  foo: "bar"
});

myFeature.readSettings(); // { foo: "bar" }
```

The object literal above is simply an object assigned to a variable. The object has one property and several methods. All of the properties and methods are public, so any part of your

application can see the properties and call methods on the object. While there is an `init` method, there's nothing requiring that it be called before the object is functional.

How would we apply this pattern to jQuery code? Let's say that we had this code written in the traditional jQuery style:

```
// Clicking on a list item loads some content using the
// list item's ID, and hides content in sibling list items
$( document ).ready(function() {
    $( "#myFeature li" ).append( "<div>" ).click(function() {
        var item = $( this );
        var div = item.find( "div" );
        div.load( "foo.php?item=" + item.attr( "id" ), function() {
            div.show();
            item.siblings().find( "div" ).hide();
        });
    });
});
```

If this were the extent of our application, leaving it as-is would be fine. On the other hand, if this was a piece of a larger application, we'd do well to keep this functionality separate from unrelated functionality. We might also want to move the URL out of the code and into a configuration area. Finally, we might want to break up the chain to make it easier to modify pieces of the functionality later.

```
// Using an object literal for a jQuery feature
var myFeature = {
    init: function( settings ) {
        myFeature.config = {
            items: $( "#myFeature li" ),
            container: $( "<div class='container'></div>" ),
            urlBase: "/foo.php?item="
        };

        // Allow overriding the default config
        $.extend( myFeature.config, settings );

        myFeature.setup();
    },

    setup: function() {
        myFeature.config.items
            .each( myFeature.createContainer )
            .click( myFeature.showItem );
    },

    createContainer: function() {
        var item = $( this );
        var container = myFeature.config.container
            .clone()
            .appendTo( item );
        item.data( "container", container );
    },

    buildUrl: function() {
        return myFeature.config.urlBase + myFeature.currentItem.attr( "id"
    );
},

    showItem: function() {
        myFeature.currentItem = $( this );
        myFeature.getContent( myFeature.showContent );
    },

    getContent: function( callback ) {
        var url = myFeature.buildUrl();
        myFeature.currentItem.data( "container" ).load( url, callback );
    },

    showContent: function() {
        myFeature.currentItem.data( "container" ).show();
        myFeature.hideContent();
    },

    hideContent: function() {
        myFeature.currentItem.siblings().each(function() {
            $( this ).data( "container" ).hide();
        });
    }
};

$( document ).ready( myFeature.init );
```

The first thing you'll notice is that this approach is obviously far longer than the original — again, if this were the extent of our application, using an object literal would likely be overkill. Assuming it's not the extent of our application, though, we've gained several things:

- We've broken our feature up into tiny methods. In the future, if we want to change how content is shown, it's clear where to change it. In the original code, this step is much harder to locate.
- We've eliminated the use of anonymous functions.
- We've moved configuration options out of the body of the code and put them in a central location.
- We've eliminated the constraints of the chain, making the code easier to refactor, remix, and rearrange.

For non-trivial features, object literals are a clear improvement over a long stretch of code stuffed in a `$(document).ready()` block, as they get us thinking about the pieces of our functionality. However, they aren't a whole lot more advanced than simply having a bunch of function declarations inside of that `$(document).ready()` block.

The Module Pattern

The module pattern overcomes some of the limitations of the object literal, offering privacy for variables and functions while exposing a public API if desired.

```
// The module pattern
var feature = (function() {

    // Private variables and functions
    var privateThing = "secret";
    var publicThing = "not secret";

    var changePrivateThing = function() {
        privateThing = "super secret";
    };

    var sayPrivateThing = function() {
        console.log( privateThing );
        changePrivateThing();
    };

    // Public API
    return {
        publicThing: publicThing,
        sayPrivateThing: sayPrivateThing
    };

})();

feature.publicThing; // "not secret"

// Logs "secret" and changes the value of privateThing
feature.sayPrivateThing();
```

In the example above, we self-execute an anonymous function that returns an object. Inside of the function, we define some variables. Because the variables are defined inside of the function, we don't have access to them outside of the function unless we put them in the return object. This means that no code outside of the function has access to the `privateThing` variable or to the `changePrivateThing` function. However, `sayPrivateThing` does have access to `privateThing` and `changePrivateThing`, because both were defined in the same scope as `sayPrivateThing`.

This pattern is powerful because, as you can gather from the variable names, it can give you private variables and functions while exposing a limited API consisting of the returned object's properties and methods.

Below is a revised version of the previous example, showing how we could create the same feature using the module pattern while only exposing one public method of the module, `showItemByIndex()`.

```
// Using the module pattern for a jQuery feature
$( document ).ready(function() {
    var feature = (function() {
        var items = $( "#myFeature li" );
        var container = $( "<div class='container'></div>" );
        var currentItem = null;
        var urlBase = "/foo.php?item=";

        var createContainer = function() {
            var item = $( this );
            var _container = container.clone().appendTo( item );
            item.data( "container", _container );
        };

        var buildUrl = function() {
            return urlBase + currentItem.attr( "id" );
        };

        return {
            showItemByIndex: function( index ) {
                // ... (code for showItemByIndex)
            }
        };
    })();
});
```

```

    };

    var showItem = function() {
        currentItem = $( this );
        getContent( showContent );
    };

    var showItemByIndex = function( idx ) {
        $.proxy( showItem, items.get( idx ) );
    };

    var getContent = function( callback ) {
        currentItem.data( "container" ).load( buildUrl(), callback
    );

    };

    var showContent = function() {
        currentItem.data( "container" ).show();
        hideContent();
    };

    var hideContent = function() {
        currentItem.siblings().each(function() {
            $( this ).data( "container" ).hide();
        });
    };

    items.each( createContainer ).click( showItem );

    return {
        showItemByIndex: showItemByIndex
    };
}());

feature.showItemByIndex( 0 );
});

```

Code Organization**Beware Anonymous Functions**

Anonymous functions bound everywhere are a pain. They're difficult to debug, maintain, test, or reuse. Instead, use an object literal to organize and name your handlers and callbacks.

```
// BAD
$( document ).ready(function() {

    $( "#magic" ).click(function( event ) {
        $( "#yayeffects" ).slideUp(function() {
            // ...
        });
    });

    $( "#happiness" ).load( url + " #unicorns", function() {
        // ...
    });

});

// BETTER
var PI = {

    onReady: function() {
        $( "#magic" ).click( PI.candyMtn );
        $( "#happiness" ).load( PI.url + " #unicorns", PI.unicornCb );
    },

    candyMtn: function( event ) {
        $( "#yayeffects" ).slideUp( PI.slideCb );
    },

    slideCb: function() { ... },

    unicornCb: function() { ... }

};

$( document ).ready( PI.onReady );
```

Code Organization

Keep Things DRY

Don't repeat yourself; if you're repeating yourself, you're doing it wrong.

```
// BAD
if ( eventfade.data( "currently" ) !== "showing" ) {
    eventfade.stop();
}

if ( eventhover.data( "currently" ) !== "showing" ) {
    eventhover.stop();
}

if ( spans.data( "currently" ) !== "showing" ) {
    spans.stop();
}

// GOOD!!
var elems = [ eventfade, eventhover, spans ];

$.each( elems, function( i, elem ) {
    if ( elem.data( "currently" ) !== "showing" ) {
        elem.stop();
    }
});
```

Code Organization

Feature & Browser Detection

Can I Use This Browser Feature?

There are a couple of common ways to check whether or not a particular feature is supported by a user's browser:

- Browser Detection
- Specific Feature Detection

In general, we recommend specific feature detection. Let's look at why.

Browser Detection

Browser detection is a method where the browser's User Agent (UA) string is checked for a particular pattern unique to a browser family or version. For example, this is Chrome 39's UA string on Mac OS X Yosemite:

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.99 Safari/537.36
```

Browser UA detection may check this string for something like "Chrome" or "Chrome/39" or any other part the developer feels identifies the browser they intend to target.

While this seems to be an easy solution, there are several problems:

Other browsers other than your target may have the same issue.

If we target a specific browser for different functionality, we implicitly exclude any browser we did not account for. This is also not future-proof. If the browser we target receives a bug fix or change, we may not be able to discern between a "working" and "non-working" UA string. We may also need to update our test for each new release. This isn't a maintainable solution.

User Agents are unreliable.

User Agents are set by the client browser. In the early days of the web, browsers would mimic each others' UA strings in order to bypass exactly this type of detection. It is still possible that a browser with very different capabilities may mimic just the portion of the UA string you're targeting.

The UA string is also user-configurable. While the user may change this string, the browser's feature support remains the same.

In general, we do not recommend UA string-based feature detection.

Specific Feature Detection

Specific feature detection checks if a specific feature is available, instead of developing against a specific browser. This way, developers can write their code for two cases: the browser **does** support said feature, or the browser **does not** support said feature.

Developing against specific features, instead of developing against a specific browser, not only clears up the peripheral logic of your application, but also makes your job easier as a developer.

We recommend specific feature detection over UA string detection.

Now how would you go about doing that?

How to go about feature detection

There are several ways to go about feature detection:

- Straight JavaScript
- A Helper Library

Straight JavaScript

Let's take a look at how to check whether or not a `<canvas>` element exists in a specific browser, without using a helper library. We do this by specifically querying whether the method or property exists:

```
// We want to show a graph in browsers that support canvas,
// but a data table in browsers that don't.
var elem = document.createElement( "canvas" );

if ( elem.getContext && elem.getContext( "2d" ) ) {
    showGraph();
} else {
    showTable();
}
```

This is a very simple way to provide conditional experiences, depending on the features present in the user's browser. We can extract this into a helper function for reuse, but still have to write a test for every feature we're concerned about. This can be time-consuming and error-prone. Instead you might be interested in using a helper library.

A Helper Library

Thankfully, there are some great helper libraries (like [Modernizr](#)) that provide a simple, high-level API for determining if a browser has a specific feature available or not.

For example, utilizing Modernizr, we are able to do the same canvas detection test with this code:

```
if ( Modernizr.canvas ) {
    showGraphWithCanvas();
} else {
    showTable();
}
```

For more in-depth information on Modernizr, feel free to check out their [documentation](#).

Performance Considerations

So, while the Modernizr syntax is great, it can end up being quite cumbersome to have several conditionals. Secondly, we're sending the code for both conditions to every browser, regardless if we'll need it or not.

If you're using Modernizr, we highly encourage you to use the [build configurator](#), a tool that allows you to create custom builds of the library. You can exclude checks you don't need, which will save bytes and reduce the time it takes the page to load. Running every check that Modernizr can do, even when you don't need them, can slow down the page load.

Other Resources

Feature Detection Tools

- [modernizr](#) — Conditionally check to see if a specific feature is available in a browser.
- [html5please](#) — Use the new and shiny responsibly.
- [html5please API](#) — An API you can query to see how good (or bad) support for a specific feature is.
- [caniuse](#) — Browser compatibility tables for HTML5, CSS3, SVG, etc.

Helpful Articles

- [Browser Feature Detection](#)
- [Using Modernizr to detect HTML5 and CSS3 browser support](#)
- [Polyfilling the HTML5 gaps](#) by Addy Osmani
- [Feature, Browser, and Form Factor Detection: It's Good for the Environment](#) by Michael Mahemoff

Code Organization - Deferreds

Deferreds

At a high-level, deferreds can be thought of as a way to represent asynchronous operations which can take a long time to complete. They're the asynchronous alternative to blocking functions and the general idea is that rather than your application blocking while it awaits some request to complete before returning a result, a deferred object can instead be returned immediately. You can then attach callbacks to the deferred object: they will be called once the request has actually completed.

Promises

In its most basic form, a "promise" is a model that provides a solution for the concept of deferred (or future) results in software engineering. The main idea behind it is something we've already covered: rather than executing a call which may result in blocking, we instead return a promise for a future value that will eventually be satisfied.

If it helps to have an example here, consider that you are building a web application which heavily relies on data from a third party API. A common problem that's faced is having an unknown knowledge of the API server's latency at a given time so it's possible that other parts of your application may be blocked from running until a result from it is returned. Deferreds provide a better solution to this problem, one which is void of "blocking" effects and completely decoupled.

The [Promises/A](#) proposal defines a method called "then" that can be used to register callbacks to a promise and, thus, get the future result when it is available. The pseudo-code for dealing with a third party API that returns a promise may look like:

```
var promise = callToAPI( arg1, arg2, ...);

promise.then(function( futureValue ) {

    // Handle futureValue

});

promise.then(function( futureValue ) {

    // Do something else

});
```

Furthermore, a promise can actually end up being in two different states:

- Resolved: in which case data is available
- Rejected: in which case something went wrong and no value is available

Thankfully, the "then" method accepts two parameters: one for when the promise was resolved, another for when the promise was rejected. If we get back to pseudo-code, we may do things like:

```
promise.then(function( futureValue ) {

    // We got a value

}, function() {

    // Something went wrong

});
```

In the case of certain applications, it is necessary to have several results returned before your application can continue at all (for example, displaying a dynamic set of options on a screen before a user is able to select the option that interests them). Where this is the case, a method called "when" exists, which can be used to perform some action once all the promises have been fully fulfilled:

```
when(
    promise1,
    promise2,
    ...
).then(function( futureValue1, futureValue2, ... ) {
```

```

        // All promises have completed and are resolved
    });

```

A good example is a scenario where you may have multiple concurrent animations that are being run. Without keeping track of each callback firing on completion, it can be difficult to truly establish once all your animations have finished running. Using promises and "when" however this is very straightforward as each of your animations can effectively say "we promise to let you know once we're done". The compounded result of this means it's a trivial process to execute a single callback once the animations are done. For example:

```

var promise1 = $( "#id1" ).animate().promise();
var promise2 = $( "#id2" ).animate().promise();
when(
    promise1,
    promise2
).then(function() {

    // Once both animations have completed
    // we can then run our additional logic

});

```

This means that one can basically write non-blocking logic that can be executed without synchronization. Rather than directly passing callbacks to functions, something which can lead to tightly coupled interfaces, using promises allows one to separate concerns for code that is synchronous or asynchronous.

Code Organization - Deferreds

jQuery Deferreds

jQuery Deferreds

Deferreds were added as a part of a large rewrite of the Ajax module, led by Julian Aubourg following the CommonJS Promises/A design. Whilst 1.5 and above include deferred capabilities, former versions of jQuery had `jQuery.ajax()` accept callbacks that would be invoked upon completion or error of the request, but suffered from heavy coupling — the same principle that would drive developers using other languages or toolkits to opt for deferred execution.

In practice what jQuery's version provides you with are several enhancements to the way callbacks are managed, giving you significantly more flexible ways to provide callbacks that can be invoked whether the original callback dispatch has already fired or not. It is also worth noting that jQuery's Deferred object supports having multiple callbacks bound to the outcome of particular tasks (and not just one) where the task itself can either be synchronous or asynchronous.

At the heart of jQuery's implementation is `jQuery.Deferred` — a chainable constructor which is able to create new deferred objects that can check for the existence of a promise to establish whether the object can be observed. It can also invoke callback queues and pass on the success of synchronous and asynchronous functions. It's quite essential to note that the default state of any Deferred object is unresolved. Callbacks which may be added to it through `.then()` or `.fail()` are queued up and get executed later on in the process.

You are able to use Deferred objects in conjunction with the promise concept of `when()`, implemented in jQuery as `$.when()` to wait for all of the Deferred object's requests to complete executing (i.e. for all of the promises to be fulfilled). In technical terms, `$.when()` is effectively a way to execute callbacks based on any number of promises that represent asynchronous events.

An example of `$.when()` accepting multiple arguments can be seen below in conjunction with `.then()`:

```
function successFunc() {
    console.log( "success!" );
}

function failureFunc() {
    console.log( "failure!" );
}

$.when(
    $.ajax( "/main.php" ),
    $.ajax( "/modules.php" ),
    $.ajax( "/lists.php" )
).then( successFunc, failureFunc );
```

The `$.when()` implementation offered in jQuery is quite interesting as it not only interprets deferred objects, but when passed arguments that are not deferreds, it treats these as if they were resolved deferreds and executes any callbacks (`doneCallbacks`) right away. It is also worth noting that jQuery's deferred implementation, in addition to exposing `deferred.then()`, also supports the `deferred.done()` and `deferred.fail()` methods which can also be used to add callbacks to the deferred's queues.

We will now take a look at a code example that uses many deferred features. This very basic script begins by consuming (1) an external news feed and (2) a reactions feed for pulling in the latest comments via `$.get()` (which will return a promise-like object). When both requests are received, the `showAjaxedContent()` function is called. The `showAjaxedContent()` function returns a promise that is resolved when animating both containers has completed. When the `showAjaxedContent()` promise is resolved, `removeActiveClass()` is called. The `removeActiveClass()` returns a promise that is resolved inside a `setTimeout()` after 4 seconds have elapsed. Finally, after the `removeActiveClass()` promise is resolved, the last `then()` callback is called, provided no errors occurred along the way.

```
function getLatestNews() {
    return $.get( "latestNews.php", function( data ) {
        console.log( "news data received" );
        $( ".news" ).html( data );
    });
}
```

```

    });
}

function getLatestReactions() {
    return $.get( "latestReactions.php", function( data ) {
        console.log( "reactions data received" );
        $( ".reactions" ).html( data );
    });
}

function showAjaxedContent() {
    // The .promise() is resolved *once*, after all animations complete
    return $( ".news, .reactions" ).slideDown( 500, function() {
        // Called once *for each element* when animation completes
        $(this).addClass( "active" );
    }).promise();
}

function removeActiveClass() {
    return $.Deferred(function( dfd ) {
        setTimeout(function () {
            $( ".news, .reactions" ).removeClass( "active" );
            dfd.resolve();
        }, 4000);
    }).promise();
}

$.when(
    getLatestNews(),
    getLatestReactions()
)
.then(showAjaxedContent)
.then(removeActiveClass)
.then(function() {
    console.log( "Requests succeeded and animations completed" );
}).fail(function() {
    console.log( "something went wrong!" );
});

```

Code Organization - Deferreds

Deferred examples

Further Deferreds examples

Deferreds are used behind the hood in Ajax but it doesn't mean they can't also be used elsewhere. This section describes situations where deferreds will help abstract away asynchronous behavior and decouple our code.

Caching

Asynchronous cache

When it comes to asynchronous tasks, caching can be a bit demanding since you have to make sure a task is only performed once for a given key. As a consequence, the code has to somehow keep track of inbound tasks.

```
$.cachedGetScript( url, callback1 );
$.cachedGetScript( url, callback2 );
```

The caching mechanism has to make sure the URL is only requested once even if the script isn't in cache yet. This shows some logic to keep track of callbacks bound to a given URL in order for the cache system to properly handle both complete and inbound requests.

```
var cachedScriptPromises = {};

$.cachedGetScript = function( url, callback ) {
    if ( !cachedScriptPromises[ url ] ) {
        cachedScriptPromises[ url ] = $.Deferred(function( defer ) {
            $.getScript( url ).then( defer.resolve, defer.reject );
        }).promise();
    }
    return cachedScriptPromises[ url ].done( callback );
};
```

One promise is cached per URL. If there is no promise for the given URL yet, then a deferred is created and the request is issued. When the request is complete, the deferred is resolved (with `defer.resolve`); if an error occurs, the deferred is rejected (with `defer.reject`). If the promise already exists, the callback is attached to the existing deferred; otherwise, the promise is first created and then the callback is attached. The big advantage of this solution is that it will handle both complete and inbound requests transparently. Another advantage is that a deferred-based cache will deal with failure gracefully. The promise will end up rejected which can be tested for by providing an error callback:

```
$.cachedGetScript( url ).then( successCallback, errorCallback );
```

Generic asynchronous cache

It is also possible to make the code completely generic and build a cache factory that will abstract out the actual task to be performed when a key isn't in the cache yet:

```
$.createCache = function( requestFunction ) {
    var cache = {};
    return function( key, callback ) {
        if ( !cache[ key ] ) {
            cache[ key ] = $.Deferred(function( defer ) {
                requestFunction( defer, key );
            }).promise();
        }
        return cache[ key ].done( callback );
    };
};
```

Now that the request logic is abstracted away, `$.cachedGetScript()` can be rewritten as follows:

```
$.cachedGetScript = $.createCache(function( defer, url ) {
    $.getScript( url ).then( defer.resolve, defer.reject );
});
```

This will work because every call to `$.createCache()` will create a new cache repository and return a new cache-retrieval function.

Image loading

A cache can be used to ensure that the same image is not loaded multiple times.

```
$.loadImage = $.createCache(function( defer, url ) {
    var image = new Image();
    function cleanUp() {
        image.onload = image.onerror = null;
    }
    defer.then( cleanUp, cleanUp );
    image.onload = function() {
        defer.resolve( url );
    };
    image.onerror = defer.reject;
    image.src = url;
});
```

Again, the following snippet:

```
$.loadImage( "my-image.png" ).done( callback1 );
$.loadImage( "my-image.png" ).done( callback2 );
```

will work regardless of whether my-image.png has already been loaded or not, or if it is actually in the process of being loaded.

Caching Data API responses

API requests that are considered immutable during the lifetime of your page are also perfect candidates. For instance, the following:

```
$.searchTwitter = $.createCache(function( defer, query ) {
    $.ajax({
        url: "http://search.twitter.com/search.json",
        data: {
            q: query
        },
        dataType: "jsonp",
        success: defer.resolve,
        error: defer.reject
    });
});
```

will allow you to perform searches on Twitter and cache them at the same time:

```
$.searchTwitter( "jQuery Deferred", callback1 );
$.searchTwitter( "jQuery Deferred", callback2 );
```

Timing

This deferred-based cache is not limited to network requests; it can also be used for timing purposes.

For instance, you may need to perform an action on the page after a given amount of time so as to attract the user's attention to a specific feature they may not be aware of or deal with a timeout (for a quiz question for instance). While `setTimeout()` is good for most use-cases it doesn't handle the situation when the timer is asked for later, even after it has theoretically expired. We can handle that with the following caching system:

```
var readyTime;

$(function() {
    readyTime = jQuery.now();
});

$.afterDOMReady = $.createCache(function( defer, delay ) {
    delay = delay || 0;
    $(function() {
        var delta = $.now() - readyTime;
        if ( delta >= delay ) {
            defer.resolve();
        } else {
            setTimeout( defer.resolve, delay - delta );
        }
    });
});
```

The new `$.afterDOMReady()` helper method provides proper timing after the DOM is ready while ensuring the bare minimum of timers will be used. If the delay is already expired, any callback will be called right away.

One-time event

While jQuery offers all the event binding one may need, it can become a bit cumbersome to handle events that are only supposed to be dealt with once.

For instance, you may wish to have a button that will open a panel the first time it is clicked and leave it open afterward or take special initialization actions the first time said button is clicked. When dealing with such a situation, one usually ends up with code like this:

```
var buttonClicked = false;

$( "#myButton" ).click(function() {
    if ( !buttonClicked ) {
        buttonClicked = true;
        initializeData();
        showPanel();
    }
});
```

then, later on, you may wish to take actions, but only if the panel is opened:

```
if ( buttonClicked ) {

    // Perform specific action

}
```

This is a very coupled solution. If you want to add some other action, you have to edit the bind code or just duplicate it all. If you don't, your only option is to test for `buttonClicked` and you may lose that new action because the `buttonClicked` variable may be `false` and your new code may never be executed.

We can do much better using deferreds (for simplification sake, the following code will only work for a single element and a single event type, but it can be easily generalized for full-fledged collections with multiple event types):

```
$.fn.bindOnce = function( event, callback ) {
    var element = $( this[ 0 ] ),
        defer = element.data( "bind_once_defer_" + event );

    if ( !defer ) {
        defer = $.Deferred();
        function deferCallback() {
            element.unbind( event, deferCallback );
            defer.resolveWith( this, arguments );
        }
        element.bind( event, deferCallback )
        element.data( "bind_once_defer_" + event , defer );
    }

    return defer.done( callback ).promise();
};
```

The code works as follows:

- Check if the element already has a deferred attached for the given event
- if not, create it and make it so it is resolved when the event is fired the first time around
- then attach the given callback to the deferred and return the promise

While the code is definitely more verbose, it makes dealing with the problem at hand much simpler in a compartmentalized and decoupled way. But let's define a helper method first:

```
$.fn.firstClick = function( callback ) {
    return this.bindOnce( "click", callback );
};
```

Then the logic can be re-factored as follows:

```
var openPanel = $( "#myButton" ).firstClick();

openPanel.done( initializeData );
openPanel.done( showPanel );
```

If an action should be performed only when a panel is opened later on:

```
openPanel.done(function() {

    // Perform specific action

});
```

```
});
```

Nothing is lost if the panel isn't opened yet, the action will just get deferred until the button is clicked.

Combining helpers

All of the samples above can seem a bit limited when looked at separately. However, the true power of promises comes into play when you mix them together.

Requesting panel content on first click and opening said panel

Following is the code for a button that, when clicked, opens a panel. It requests its content over the wire and then fades the content in. Using the helpers defined earlier, it could be defined as:

```
$( "#myButton" ).firstClick(function() {
    var panel = $( "#myPanel" );
    $.when(
        $.get( "panel.html" ),
        panel.slideDownPromise()
    ).done(function( ajaxResponse ) {
        panel.html( ajaxResponse[ 0 ] ).fadeIn();
    });
});
```

Loading images in a panel on first click and opening said panel

Another possible goal is to have the panel fade in, only after the button has been clicked and after all of the images have been loaded.

The HTML code for this would look something like:

```
<div id="myPanel">
    
    
    
    
</div>
```

We use the data-src attribute to keep track of the real image location. The code to handle our use case using our promise helpers is as follows:

```
$( "#myButton" ).firstClick(function() {
    var panel = $( "#myPanel" ),
        promises = [];

    panel.find( "img" ).each(function() {
        var image = $( this ),
            src = element.attr( "data-src" );
        if ( src ) {
            promises.push(
                $.loadImage( src ).then(function() {
                    image.attr( "src", src );
                }, function() {
                    image.attr( "src", "error.png" );
                })
            );
        }
    });

    promises.push( panel.slideDownPromise() );

    $.when.apply( null, promises ).done(function() {
        panel.fadeIn();
    });
});
```

The trick here is to keep track of all the \$.loadImage() promises. We later join them with the panel .slideDown() animation using \$.when(). So when the button is first clicked, the panel will slide down and the images will start loading. Once the panel has finished sliding down and all the images have been loaded, then, and only then, will the panel fade in.

Loading images on the page after a specific delay

In order to implement deferred image display on the entire page, the following format in HTML can be used.


```






```

What it says is pretty straight-forward:

- Load image1.png and show it immediately for the third image and after one second for the first one
- Load image2.png and show it after one second for the second image and after two seconds for the fourth image

```

$( "img" ).each(function() {
    var element = $( this ),
        src = element.attr( "data-src" ),
        after = element.attr( "data-after" );
    if ( src ) {
        $.when(
            $.loadImage( src ),
            $.afterDOMReady( after )
        ).then(function() {
            element.attr( "src", src );
        }, function() {
            element.attr( "src", "error.png" );
        }).done(function() {
            element.fadeIn();
        });
    }
});

```

In order to delay the loading of the images themselves:

```

$( "img" ).each(function() {
    var element = $( this ),
        src = element.attr( "data-src" ),
        after = element.attr( "data-after" );
    if ( src ) {
        $.afterDOMReady( after, function() {
            $.loadImage( src ).then(function() {
                element.attr( "src", src );
            }, function() {
                element.attr( "src", "error.png" );
            }).done(function() {
                element.fadeIn();
            });
        });
    }
});

```

Here, after the delay to be fulfilled then the image is loaded. It can make a lot of sense when you want to limit the number or network requests on page load.

jQuery UI

[jQuery UI](#) is a curated set of user interface interactions, effects, widgets, and themes built on top of the jQuery JavaScript Library. Whether you're building highly interactive web applications or you just need to add a date picker to a form control, jQuery UI is the perfect choice.

jQuery UI contains many widgets that maintain state and therefore have a slightly different usage pattern than typical jQuery plugins. All of jQuery UI's widgets use the same patterns, so if you learn how to use one, then you'll know how to use all of them.

jQuery UI

Getting Started with jQuery UI

What is jQuery UI?

jQuery UI is a widget and interaction library built on top of the jQuery JavaScript Library that you can use to build highly interactive web applications. This guide is designed to get you up to speed on how jQuery UI works. Follow along below to get started.

Start by Checking Out the Demos

To get a feel for what jQuery UI is capable of, check out the [UI Demos](#).

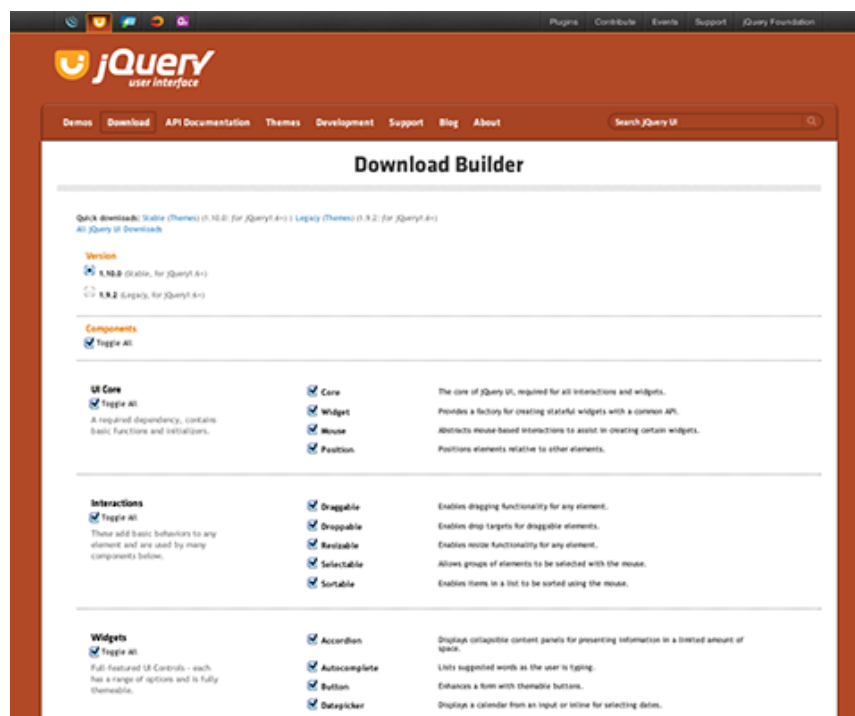
In the demos section, the navigation lists all of the interactions and widgets that jQuery UI offers. Choose an interaction or widget and you'll be presented with several demo configurations for that particular plugin. Each demo allows you to view source code, change themes, and the URL can always be bookmarked. For example, check out the [accordion widget's fill space demo page](#).

Build Your Custom jQuery UI Download

Once you have a basic understanding of what jQuery UI is and what it does, you're ready to try it out! It's time to head over to the [Download Builder](#) on the jQuery UI website to download a copy of jQuery UI. jQuery UI's Download Builder allows you to choose the components you would like to download and get a custom version of the library for your project. There are three easy steps to building your custom jQuery UI download:

Step 1: Choose Which Components You Need

The main column of the Download Builder lists all of the JavaScript components in jQuery UI categorized into groups: core, interactions, widgets, and effects. Some components in jQuery UI depend on other components. Just check the boxes for the widgets you'd like to download and any required dependencies will automatically be checked as well. The components you select will all be combined into a custom jQuery UI JavaScript file.



Step 2: Select a Theme (or Roll Your Own Custom Theme)

In the right column of the Download Builder, you'll find a field where you can choose from a number of pre-designed themes for your jQuery UI widgets. You can either choose from the various themes we provide, or you can design your own custom theme using ThemeRoller (more on that later).

Advanced Theme Settings: *The theme section of the Download Builder also offers some advanced configuration settings for your theme. If you plan to use multiple themes on a single page, these fields will*

come in handy. If you plan to only use one theme on a page, you can skip these settings entirely.

Step 3: Choose a Version of jQuery UI

The last step in the Download Builder is to select a version number. Make sure to check not only what version of jQuery UI you pick, but also the version of jQuery Core that version supports, as different versions of the library support different versions of jQuery. For more information on what's new in each version of jQuery UI, see the project's [upgrade guides](#) and [changelogs](#).

Click Download!

You're finished with the Download Builder! Click the download button and you'll get a customized zip file containing everything you selected.

Basic Overview: Using jQuery UI on a Web Page

Next, open up `index.html` from the downloaded zip in a text editor. You'll see that it references your theme, jQuery, and jQuery UI. Generally, you'll need to include these three files on any page to use the jQuery UI widgets and interactions:

```
<link rel="stylesheet" href="jquery-ui.min.css">
<script src="external/jquery/jquery.js"></script>
<script src="jquery-ui.min.js"></script>
```

Once you've included the necessary files, you can add some jQuery widgets to your page. For example, to make a datepicker widget, you'll add a text input element to your page and then call `.datepicker()` on it. Like this:

HTML:

```
<input type="text" name="date" id="date">
```

JavaScript:

```
$( "#date" ).datepicker();
```



That's It!

For demos of all of the jQuery UI widgets and interactions, check out the demos section of the jQuery UI website.

Customizing jQuery UI to Your Needs

jQuery UI allows you to customize it in several ways. You've already seen how the Download Builder allows you to customize your copy of jQuery UI to include only the portions you want, but there are additional ways to customize that code to your implementation.

jQuery UI Basics: Using Options

Each plugin in jQuery UI has a default configuration which is catered to the most basic and common use case. But if you want a plugin to behave different from its default configuration, you can override each of its default settings using "options". Options are a set of properties passed into a jQuery UI widget as an argument. For example, the slider widget has an option for orientation, which allows you to specify whether the slider should be horizontal or vertical. To set this option for a slider on your page, you just pass it in as an argument, like this:

```
$( "#mySliderDiv" ).slider({
    orientation: "vertical"
});
```

You can pass as many different options as you'd like by following each one with a comma (except the last one):

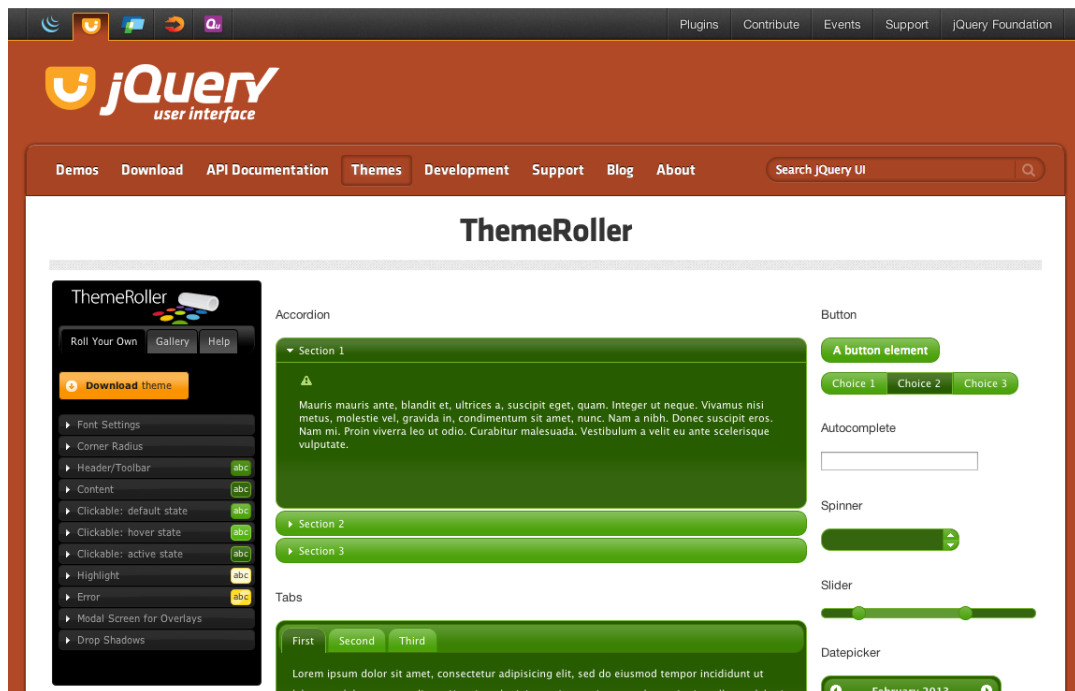
```
$( "#mySliderDiv" ).slider({
    orientation: "vertical",
    min: 0,
    max: 150,
    value: 50
});
```

Just remember to surround your options with curly brackets { }, and you're well on your way. Of course, the example above barely touches on what you can do with jQuery UI. To get detailed information on the entire set of jQuery UI widgets, visit the [jQuery UI documentation](#).

Visual Customization: Designing a jQuery UI Theme

If you want to design your own theme, jQuery UI has a very slick application for just that purpose. It's called ThemeRoller, and you can always get to it by either clicking the "Themes" link in the jQuery UI navigation, or simply going to [jQuery UI ThemeRoller](#).

ThemeRoller provides a custom interface for designing all of the elements used by jQuery UI widgets. As you tweak the "levers" in the left column, the widgets on the right will reflect your design. The Gallery tab of ThemeRoller offers a number of pre-designed themes (the same ones offered by the Download Builder) that you can tweak or download as they are.



Downloading Your Theme

When you click the "Download theme" button in ThemeRoller, you'll be directed to the Download Builder and your custom theme will be auto-selected in the Theme dropdown menu. You can configure your download package further from there. Once you download, you'll see that the `example.html` page is styled using your custom theme.

Quick Tip: If you ever need to edit your theme, simply open the CSS file and find where it says "To view and modify this theme, visit ...". That URL will open the theme in ThemeRoller for editing.

Support: Where Can I Get Help?

jQuery UI user and developer resources are kept up-to-date at the [Support Center](#).

Developers Wanted!

Want to join the jQuery UI team? We'd love your help! Visit the UI [Development Center](#) for details on how to get involved.

jQuery UI

How jQuery UI Works

jQuery UI contains many widgets that maintain [state](#) and therefore may have a slightly different usage pattern than typical jQuery plugins you are already used to. While the initialization is the same as most jQuery plugins, jQuery UI's widgets are built on top of the [Widget Factory](#), which provides the same general API to all of them. So if you learn how to use one, then you'll know how to use all of them! This document will walk you through the common functionality, using the [progressbar](#) widget for the code examples.

Initialization

In order to track the state of the widget, we must introduce a full life cycle for the widget. The life cycle starts when the widget is initialized. To initialize a widget, we simply call the plugin on one or more elements.

```
$( "#elem" ).progressbar();
```

This will initialize each element in the jQuery object, in this case the element with an id of "elem". Because we called the `.progressbar()` method with no parameters, the widget is initialized with its default options. We can pass a set of options during initialization in order to override the default options.

```
$( "#elem" ).progressbar({ value: 20 });
```

We can pass as many or as few options as we want during initialization. Any options that we don't pass will just use their default values.

The options are part of the widget's state, so we can set options after initialization as well. We'll see this later with the `option` method.

Methods

Now that the widget is initialized, we can query its state or perform actions on the widget. All actions after initialization take the form of a method call. To call a method on a widget, we pass the name of the method to the jQuery plugin. For example, to call the `value` method on our progressbar widget, we would use:

```
$( "#elem" ).progressbar( "value" );
```

If the method accepts parameters, we can pass them after the method name. For example, to pass the parameter 40 to the `value` method, we can use:

```
$( "#elem" ).progressbar( "value", 40 );
```

Just like other methods in jQuery, most widget methods return the jQuery object for chaining.

```
$( "#elem" )
    .progressbar( "value", 90 )
    .addClass( "almost-done" );
```

Common Methods

Each widget will have its own set of methods based on the functionality that the widget provides. However, there are a few methods that exist on all widgets.

option

As we mentioned earlier, we can change options after initialization through the `option` method. For example, we can change the progressbar's value to 30 by calling the `option` method.

```
$( "#elem" ).progressbar( "option", "value", 30 );
```

Note that this is different from the previous example where we were calling the `value` method. In this example, we're calling the `option` method and saying that we want to change the value option to 30.

We can also get the current value for an option.

```
$( "#elem" ).progressbar( "option", "value" );
```

In addition, we can update multiple options at once by passing an object to the `option` method.

```
$( "#elem" ).progressbar( "option", {
    value: 100,
    disabled: true
});
```

You may have noticed that the `option` method has the same signature as getters and setters in jQuery core, such as `.css()` and `.attr()`. The only difference is that you have to pass the string "option" as the first parameter.

disable

As you might guess, the `disable` method disables the widget. In the case of `progressbar`, this changes the styling to make the progressbar look disabled.

```
$( "#elem" ).progressbar( "disable" );
```

Calling the `disable` method is equivalent to setting the `disabled` option to `true`.

enable

The `enable` method is the opposite of the `disable` method.

```
$( "#elem" ).progressbar( "enable" );
```

Calling the `enable` method is equivalent to setting the `disabled` option to `false`.

destroy

If you no longer need the widget, you can destroy it and return back to the original markup. This ends the life cycle of the widget.

```
$( "#elem" ).progressbar( "destroy" );
```

Once you destroy a widget, you can no longer call any methods on it unless you initialize the widget again. If you're removing the element, either directly via `.remove()` or by modifying an ancestor with `.html()` or `.empty()`, the widget will automatically destroy itself.

widget

Some widgets generate wrapper elements, or elements disconnected from the original element. In these cases, the `widget` method will return the generated element. In cases like the `progressbar`, where there is no generated wrapper, the `widget` method returns the original element.

```
$( "#elem" ).progressbar( "widget" );
```

Events

All widgets have events associated with their various behaviors to notify you when the state is changing. For most widgets, when the events are triggered, the names are prefixed with the widget name. For example, we can bind to `progressbar`'s `change` event which is triggered whenever the value changes.

```
$( "#elem" ).bind( "progressbarchange", function() {
    alert( "The value has changed!" );
});
```

Each event has a corresponding callback, which is exposed as an option. We can hook into `progressbar`'s `change` callback instead of binding to the `progressbarchange` event, if we wanted to.

```
$( "#elem" ).progressbar({
    change: function() {
        alert( "The value has changed!" );
    }
});
```

```
});
```

Common Events

While most events will be widget specific, all widgets have a `create` event. This event will be triggered immediately after the widget is created.

jQuery UI

Theming jQuery UI

All jQuery UI plugins are designed to allow a developer to seamlessly integrate UI widgets into the look and feel of their site or application. Each plugin is styled with CSS and contains two layers of style information: standard [jQuery UI CSS Framework](#) styles and plugin-specific styles.

The jQuery UI CSS Framework provides semantic presentation classes to indicate the role of an element within a widget such as a header, content area, or clickable region. These are applied consistently across all widgets so a clickable tab, accordion, or button will all have the same `ui-state-default` class applied to indicate that it is clickable. When a user mouses over one of these elements, this class is changed to `ui-state-hover`, then `ui-state-active` when selected. This level of class consistency makes it easy to ensure that all elements with a similar role or interaction state will look the same across all widgets.

The CSS Framework styles are encapsulated in a single file called `theme.css` and this is the file modified by the [ThemeRoller](#) application. Framework styles only include attributes that affect the look and feel (primarily color, background images, and icons) so these are "safe" styles that will not affect functionality of individual plugins. This separation means that a developer can create a custom look and feel by modifying the colors and images in the `theme.css` file and know that as future plugins or bug fixes become available, these should work with the theme without modification.

Since the framework styles only cover look and feel, plugin specific stylesheets are separated. These contain all the additional structural style rules required to make the widget functional, such as dimensions, padding, margins, positioning, and floats. When downloading jQuery UI, these can be found in `jquery-ui.structure.css`.

We encourage all developers creating jQuery plugins to leverage the jQuery UI CSS Framework because it will make it much easier for end users to theme and use your plugin.

Getting started

There are three general approaches to theming jQuery UI plugins:

- **Download a ThemeRoller theme:** The easiest way to build a theme is to use [ThemeRoller](#) to generate and download a theme. This app will create a new `jquery-ui.theme.css` file and an `images` directory containing all necessary background images and icon sprites which can simply be dropped into your project. This approach will be the easiest to create and maintain but limits customization to the options provided in ThemeRoller.
- **Modify the CSS files:** To get a bit more control over the look and feel, you may choose to start with the default theme (Smoothness) or a ThemeRoller-generated theme and then adjust the `jquery-ui.theme.css` file or any of the individual plugin stylesheets. For example, you could easily tweak the corner radius for all buttons to be different than the rest of the UI components or change the path for the icon sprite to use a custom set. With a bit of style scoping, you can even use multiple themes together in a single UI. To keep maintenance simple, restricting changes to just the `jquery-ui.theme.css` file and images is recommended.
- **Write completely custom CSS:** For the greatest amount of control, the CSS for each plugin can be written from scratch without using the framework classes or plugin-specific stylesheet. This may be necessary if the desired look and feel can't be achieved by modifying the CSS or if highly customized markup is used. This approach requires deep expertise in CSS and will require manual updates for future plugins.


jQuery UI

Using jQuery UI ThemeRoller



About ThemeRoller

ThemeRoller is a web app that offers a fun and intuitive interface for designing and downloading custom themes for jQuery UI. You can find ThemeRoller in the "Themes" section of the jQuery UI site, or by following this link: [jQuery UI ThemeRoller](#)

ThemeRoller 

Roll Your Own Gallery Help

[Download theme](#)

- Font Settings
- Corner Radius
- Header/Toolbar
- Content
- Clickable: default state
- Clickable: hover state
- Clickable: active state
- Highlight
- Error
- Modal Screen for Overlays
- Drop Shadows

jQuery UI 1.5 Users: [Download theme](#)

ThemeRoller 

Roll Your Own Gallery Help

December 2008

| Su | Mo | Tu | We | Th | Fr | Sa |
|----|----|----|----|----|----|----|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | | | |

UI lightness

[Download](#) [Edit](#)

December 2008

| Su | Mo | Tu | We | Th | Fr | Sa |
|----|----|----|----|----|----|----|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | | | |

UI darkness

[Download](#) [Edit](#)

December 2008

| Su | Mo | Tu | We | Th | Fr | Sa |
|----|----|----|----|----|----|----|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | | | |

Smoothness

[Download](#) [Edit](#)

December 2008

| Su | Mo | Tu | We | Th | Fr | Sa |
|----|----|----|----|----|----|----|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | | | |

Start

[Download](#) [Edit](#)

December 2008

| Su | Mo | Tu | We | Th | Fr | Sa |
|----|----|----|----|----|----|----|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | | | |

December 2008

| Su | Mo | Tu | We | Th | Fr | Sa |
|----|----|----|----|----|----|----|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | | | |



The ThemeRoller Interface

The interface for ThemeRoller is categorized into panels for global font and corner radius settings, widget container styles, and interaction states for clickable elements, and various styles for overlays and shadows. These panels allow configuration of various CSS properties such as font size, color, and weight, background color and texture, border color, text color, icon color, corner radius, and more!

The Theme Gallery: Pre-Rolled Themes

ThemeRoller themes can be viewed via permalink URLs, and it includes a gallery of pre-designed themes to choose from. The theme gallery is accessible through the tab strip located at the top of the application interface. From the gallery, you can preview and download themes, or even choose to tweak a theme further in the "Roll Your Own" tab.

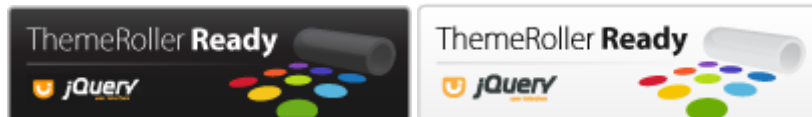
Downloading Themes

When you're done designing a theme, you can download it for use in your projects. ThemeRoller has a "Download theme" button at the top which will lead you to the download builder page. From there you can also pick the components to download with your theme. Finally the Download button at the bottom will generate a zip file containing the theme along with assets like images and any components you picked. Images included in your download will be generated to your specifications and saved as high-quality PNG files.

Installing Downloaded Themes Into Your Project

Once you've unzipped the package, you will see several `css` and `js` files, along with two folders. If you're only interested in the full theme, copy `jquery-ui.css` and the `images` folder into your project and link to the `jquery-ui.css` file from your pages.

Building Custom "ThemeRoller-Ready" Components



ThemeRoller generates a customized version of the jQuery UI CSS Framework for developing your own ThemeRoller-ready jQuery components. The classes generated by this framework are designed to accommodate common user interface design situations and include states, icons, and various helper classes as well.

For information on developing with the jQuery UI CSS Framework, visit our [Theming API documentation](#).

jQuery UI - Widget Factory

Widget Factory

The jQuery UI Widget Factory is an extensible base on which all of jQuery UI's widgets are built. Using the widget factory to build a plugin provides conveniences for state management, as well as conventions for common tasks like exposing plugin methods and changing options after instantiation.

jQuery UI - Widget Factory

Why Use the Widget Factory?

Writing jQuery plugins is as simple as adding a method to `jQuery.prototype` (more commonly seen as `$.fn`) and following some simple conventions like returning `this` for chainability. So why does the widget factory exist? And why is it hundreds of lines of code?

In this document, we'll walk through the benefits of the widget factory and find out when and why it makes sense to use it.

Stateless vs. Stateful Plugins

Most jQuery plugins are stateless; they perform some action and their job is done. For example, if you set the text of an element using `.text("hello")`, there is no setup phase and the result is always the same. For these types of plugins, it makes sense to just extend jQuery's prototype.

However, some plugins are stateful; they have full life cycles, maintain state, and react to changes. These plugins require a lot of code dedicated to initialization and state management (and sometimes destruction). This results in a lot of boilerplate for building stateful plugins. Even worse, each plugin author may manage life cycles and state differently, resulting in different API styles for different plugins. The widget factory aims to solve both problems, removing the boilerplate and creating a consistent API across plugins.

Consistent API

The widget factory defines how to create and destroy widgets, get and set options, invoke methods, and listen to events triggered by the widget. By using the widget factory to build your stateful plugins, you are automatically conforming to a defined standard, making it easier for new users to start using your plugins. In addition to defining the interface, the widget factory also implements much of this functionality for you. If you're not familiar with the API provided by the widget factory, you should read [How jQuery UI Works](#).

Setting Options on Initialization

Whenever you build a plugin that accepts options, you should define defaults for as many options as possible, then merge the user-provided options with the defaults on initialization. It's also a good idea to expose the defaults so that users can even change the default values. A common pattern in jQuery plugins looks like this:

```
$.fn.plugin = function( options ) {
    options = $.extend( {}, $.fn.plugin.defaults, options );
    // Plugin logic goes here.
};

$.fn.plugin.defaults = {
    param1: "foo",
    param2: "bar",
    param3: "baz"
};
```

The widget factory provides this functionality and even takes it a bit further. Let's see what this looks like with the widget factory.

```
$.widget( "ns.plugin", {
    // Default options.
    options: {
        param1: "foo",
        param2: "bar",
        param3: "baz"
    },
    _create: function() {
        // Options are already merged and stored in this.options
        // Plugin logic goes here.
    }
});
```

jQuery UI - Widget Factory

How To Use the Widget Factory

To start, we'll create a progress bar that just lets us set the progress once. As we can see below, this is done by calling `jQuery.widget()` with two parameters: the name of the plugin to create, and an object literal containing functions to support our plugin. When our plugin gets called, it will create a new plugin instance and all functions will be executed within the context of that instance. This is different from a standard jQuery plugin in two important ways. First, the context is an object, not a DOM element. Second, the context is always a single object, never a collection.

```
$.widget( "custom.progressbar", {
  _create: function() {
    var progress = this.options.value + "%";
    this.element
      .addClass( "progressbar" )
      .text( progress );
  }
});
```

The name of the plugin must contain a namespace, in this case we've used the `custom` namespace. You can only create namespaces that are one level deep, therefore, `custom.progressbar` is a valid plugin name whereas `very.custom.progressbar` is not.

We can also see that the widget factory has provided two properties for us. `this.element` is a jQuery object containing exactly one element. If our plugin is called on a jQuery object containing multiple elements, a separate plugin instance will be created for each element, and each instance will have its own `this.element`. The second property, `this.options`, is a hash containing key/value pairs for all of our plugin's options. These options can be passed to our plugin as shown here.

```
$( "<div></div>" )
  .appendTo( "body" )
  .progressbar({ value: 20 });
```

When we call `jQuery.widget()` it extends jQuery by adding a function to `jQuery.fn` (the system for creating a standard plugin). The name of the function it adds is based on the name you pass to `jQuery.widget()`, without the namespace - in our case "progressbar". The options passed to our plugin are the values that get set in `this.options` inside of our plugin instance. As shown below, we can specify default values for any of our options. When designing your API, you should figure out the most common use case for your plugin so that you can set appropriate default values and make all options truly optional.

```
$.widget( "custom.progressbar", {
  // Default options.
  options: {
    value: 0
  },
  _create: function() {
    var progress = this.options.value + "%";
    this.element
      .addClass( "progressbar" )
      .text( progress );
  }
});
```

Calling Plugin Methods

Now that we can initialize our progress bar, we'll add the ability to perform actions by calling methods on our plugin instance. To define a plugin method, we just include the function in the object literal that we pass to `jQuery.widget()`. We can also define "private" methods by prepending an underscore to the function name.

```
$.widget( "custom.progressbar", {
  options: {
    value: 0
  },
  _create: function() {
    var progress = this.options.value + "%";
    this.element
      .addClass( "progressbar" )
      .text( progress );
  }
});
```

```

    },
    // Create a public method.
    value: function( value ) {

        // No value passed, act as a getter.
        if ( value === undefined ) {
            return this.options.value;
        }

        // Value passed, act as a setter.
        this.options.value = this._constrain( value );
        var progress = this.options.value + "%";
        this.element.text( progress );
    },

    // Create a private method.
    _constrain: function( value ) {
        if ( value > 100 ) {
            value = 100;
        }
        if ( value < 0 ) {
            value = 0;
        }
        return value;
    }
});

```

To call a method on a plugin instance, you pass the name of the method to the jQuery plugin. If you are calling a method that accepts parameters, you simply pass those parameters after the method name.

Note: Executing methods by passing the method name to the same jQuery function that was used to initialize the plugin may seem odd. This is done to prevent pollution of the jQuery namespace while maintaining the ability to chain method calls. Later in this article we'll see alternative uses that may feel more natural.

```

var bar = $( "<div></div>" )
    .appendTo( "body" )
    .progressbar({ value: 20 });

// Get the current value.
alert( bar.progressbar( "value" ) );

// Update the value.
bar.progressbar( "value", 50 );

// Get the current value again.
alert( bar.progressbar( "value" ) );

```

Working with Options

One of the methods that are automatically available to our plugin is the `option()` method. The `option()` method allows you to get and set options after initialization. This method works exactly like jQuery's `.css()` and `.attr()` methods: You can pass just a name to use it as a getter, a name and value to use it as a single setter, or a hash of name/value pairs to set multiple values. When used as a getter, the plugin will return the current value of the option that corresponds to the name that was passed in. When used as a setter, the plugin's `_setOption` method will be called for each option that is being set. We can specify a `_setOption` method in our plugin to react to option changes. For actions to perform independent of the number of options changed, we can override `_setOptions()`.

```

$.widget( "custom.progressbar", {
    options: {
        value: 0
    },
    _create: function() {
        this.options.value = this._constrain(this.options.value);
        this.element.addClass( "progressbar" );
        this.refresh();
    },
    _setOption: function( key, value ) {
        if ( key === "value" ) {
            value = this._constrain( value );
        }
        this._super( key, value );
    },
    _setOptions: function( options ) {
        this._super( options );
        this.refresh();
    },
    refresh: function() {
        var progress = this.options.value + "%";
        this.element.text( progress );
    }
});

```



```

    },
    _constrain: function( value ) {
        if ( value > 100 ) {
            value = 100;
        }
        if ( value < 0 ) {
            value = 0;
        }
        return value;
    }
});

```

Adding Callbacks

One of the easiest ways to make your plugin extensible is to add callbacks so users can react when the state of your plugin changes. We can see below how to add a callback to our progress bar to signify when the progress has reached 100%. The `_trigger()` method takes three parameters: the name of the callback, a jQuery event object that initiated the callback, and a hash of data relevant to the event. The callback name is the only required parameter, but the others can be very useful for users who want to implement custom functionality on top of your plugin. For example, if we were building a draggable plugin, we could pass the mousemove event when triggering a drag callback; this would allow users to react to the drag based on the x/y coordinates provided by the event object. Note that the original event passed to `_trigger()` must be a jQuery event, not a native browser event.

```

$.widget( "custom.progressbar", {
    options: {
        value: 0
    },
    _create: function() {
        this.options.value = this._constrain(this.options.value);
        this.element.addClass( "progressbar" );
        this.refresh();
    },
    _setOption: function( key, value ) {
        if ( key === "value" ) {
            value = this._constrain( value );
        }
        this._super( key, value );
    },
    _setOptions: function( options ) {
        this._super( options );
        this.refresh();
    },
    refresh: function() {
        var progress = this.options.value + "%";
        this.element.text( progress );
        if ( this.options.value == 100 ) {
            this._trigger( "complete", null, { value: 100 } );
        }
    },
    _constrain: function( value ) {
        if ( value > 100 ) {
            value = 100;
        }
        if ( value < 0 ) {
            value = 0;
        }
        return value;
    }
});

```

Callback functions are essentially just additional options, so you can get and set them just like any other option. Whenever a callback is executed, a corresponding event is triggered as well. The event type is determined by concatenating the plugin name and the callback name. The callback and event both receive the same two parameters: an event object and a hash of data relevant to the event, as we'll see below. Your plugin may have functionality that you want to allow the user to prevent. The best way to support this is by creating cancelable callbacks. Users can cancel a callback, or its associated event, the same way they cancel any native event, by calling `event.preventDefault()` or returning `false`. If the user cancels the callback, the `_trigger()` method will return `false` so you can implement the appropriate functionality within your plugin.

```

var bar = $( "<div></div>" )
    .appendTo( "body" )
    .progressbar({
        complete: function( event, data ) {
            alert( "Callbacks are great!" );
        }
    })
    .bind( "progressbarcomplete", function( event, data ) {
        alert( "Events bubble and support many handlers for extreme

```

```
flexibility." );
    alert( "The progress bar value is " + data.value );
});

bar.progressbar( "option", "value", 100 );
```

Looking Under the Hood

Now that we've seen how to build a plugin using the widget factory, let's take a look at how it actually works. When you call `jQuery.widget()`, it creates a constructor for your plugin and sets the object literal that you pass in as the prototype for your plugin instances. All of the functionality that automatically gets added to your plugin comes from a base widget prototype, which is defined as `jQuery.Widget.prototype`. When a plugin instance is created, it is stored on the original DOM element using `jQuery.data`, with the plugin name as the key.

Because the plugin instance is directly linked to the DOM element, you can access the plugin instance directly instead of going through the exposed plugin method if you want. This will allow you to call methods directly on the plugin instance instead of passing method names as strings and will also give you direct access to the plugin's properties.

```
var bar = $( "<div></div>" )
    .appendTo( "body" )
    .progressbar()
    .data( "custom-progressbar" );

// Call a method directly on the plugin instance.
bar.option( "value", 50 );

// Access properties on the plugin instance.
alert( bar.options.value );
```

You can also create an instance without going through the plugin method, by calling the constructor directly, with the options and element arguments:

```
var bar = $.custom.progressbar( {}, $( "<div></div>" ).appendTo( "body" ) );

// Same result as before.
alert( bar.options.value );
```

Extending a Plugin's Prototype

One of the biggest benefits of having a constructor and prototype for a plugin is the ease of extending the plugin. By adding or modifying methods on the plugin's prototype, we can modify the behavior of all instances of our plugin. For example, if we wanted to add a method to our progress bar to reset the progress to 0% we could add this method to the prototype and it would instantly be available to be called on any plugin instance.

```
$.custom.progressbar.prototype.reset = function() {
    this._setOption( "value", 0 );
};
```

For more information on extending widgets, including how to build entirely new widgets on top of existing ones, see [Extending Widgets with the Widget Factory](#).

Cleaning Up

In some cases, it will make sense to allow users to apply and then later unapply your plugin. You can accomplish this via the `_destroy()` method. Within the `_destroy()` method, you should undo anything your plugin may have done during initialization or later use. `_destroy()` is called by the `destroy()` method, which is automatically called if the element that your plugin instance is tied to is removed from the DOM, so this can be used for garbage collection as well. That base `destroy()` method also handles some general cleanup operations, like removing the instance reference from the widget's DOM element, unbinding all events in the widget's namespace from the element, and unbinding generally all events that were added using `_bind()`.

```
$.widget( "custom.progressbar", {
    options: {
        value: 0
    },
    _create: function() {
        this.options.value = this._constrain(this.options.value);
        this.element.addClass( "progressbar" );
        this.refresh();
    },
```

```

        _setOption: function( key, value ) {
            if ( key === "value" ) {
                value = this._constrain( value );
            }
            this._super( key, value );
        },
        _setOptions: function( options ) {
            this._super( options );
            this.refresh();
        },
        refresh: function() {
            var progress = this.options.value + "%";
            this.element.text( progress );
            if ( this.options.value == 100 ) {
                this._trigger( "complete", null, { value: 100 } );
            }
        },
        _constrain: function( value ) {
            if ( value > 100 ) {
                value = 100;
            }
            if ( value < 0 ) {
                value = 0;
            }
            return value;
        },
        _destroy: function() {
            this.element
                .removeClass( "progressbar" )
                .text( "" );
        }
    });

```

Closing Comments

The widget factory is only one way of creating stateful plugins. There are a few different models that can be used and each has their own advantages and disadvantages. The widget factory solves lots of common problems for you and can greatly improve productivity, it also greatly improves code reuse, making it a great fit for jQuery UI as well as many other stateful plugins.

You may have noticed that in this article we used the `custom` namespace. The `ui` namespace is reserved for official jQuery UI plugins. When building your own plugins, you should create your own namespace. This makes it clear where the plugin came from and if it is part of a larger collection.

jQuery UI - Widget Factory

Widget Method Invocation

Widgets created with [the widget factory](#) use methods to change their state and perform actions after initialization. There are two ways widget methods can be invoked - through the plugin created by the widget factory, or by invoking the method on the element's instance object.

Plugin Invocation

To invoke a method using the widget's plugin, pass the name of the method as a string. For example, here is how you call the [dialog widget's close\(\) method](#).

```
$( ".selector" ).dialog( "close" );
```

If the method requires arguments, pass them as additional parameters to the plugin. Here is how you call [dialog's option\(\) method](#).

```
$( ".selector" ).dialog( "option", "height" );
```

This returns the value of the [dialog's height option](#).

Instance Invocation

Under the hood, every instance of every widget is stored on the element using [jQuery.data\(\)](#). To retrieve the instance object, call `jQuery.data()` using the widget's full name as the key. This is shown below.

```
var dialog = $( ".selector" ).data( "ui-dialog" );
```

After you have a reference to the instance object, methods can be invoked on it directly.

```
var dialog = $( ".selector" ).data( "ui-dialog" );
dialog.close();
```

In jQuery UI 1.11, the new `instance()` method will make this process even easier.

```
$( ".selector" ).dialog( "instance" ).close();
```

Return Types

Most methods invoked through the widget's plugin will return a jQuery object so the method call can be chained with additional jQuery methods. This is even true of methods that return undefined when invoked on the instance. This is shown in the example below.

```
var dialog = $( ".selector" ).dialog();

// Instance invocation - returns undefined
dialog.data( "ui-dialog" ).close();

// Plugin invocation - returns a jQuery object
dialog.dialog( "close" );

// Therefore, plugin method invocation makes it possible to
// chain method calls with other jQuery functions
dialog.dialog( "close" )
    .css( "color", "red" );
```

The exception to this are methods that return information about the widget. For example [dialog's isOpen\(\) method](#).

```
$( ".selector" )
    .dialog( "isOpen" )
    // This will throw a TypeError
    .css( "color", "red" );
```

This produces a `TypeError` error as `isOpen()` returns a boolean, not a jQuery object.

jQuery UI - Widget Factory

Extending Widgets with the Widget Factory

jQuery UI's widget factory makes it easy to build widgets that extend the functionality of existing widgets. Doing so allows you to build powerful widgets on top of an existing base, as well as make small tweaks to an existing widget's functionality.

Note: This article assumes some basic knowledge of what the widget factory is and how it works. If you're unfamiliar with this, read up on [how to use the widget factory](#) first.

Creating Widget Extensions

Creating widgets with the widget factory is done by passing the name of the widget and a prototype object to `$.widget()`. The following creates a "superDialog" widget in the "custom" namespace.

```
$.widget( "custom.superDialog", {} );
```

To allow for extension, `$.widget()` optionally accepts the constructor of a widget to use as a parent. When specifying a parent widget, pass it as the second argument - after the widget's name, and before the widget's prototype object.

Like the previous example, the following also creates a "superDialog" widget in the "custom" namespace. However, this time the constructor of [jQuery UI's dialog widget](#) (`$.ui.dialog`) is passed, indicating that the superDialog widget should use jQuery UI's dialog widget as a parent.

```
$.widget( "custom.superDialog", $.ui.dialog, {} );
```

Here superDialog and dialog are essentially equivalent widgets with different names and namespaces. To make our new widget more interesting we can add methods to its prototype object.

A widget's prototype object is the final argument passed to `$.widget()`. So far, our examples have been using an empty object. Let's add a method to this object:

```
$.widget( "custom.superDialog", $.ui.dialog, {
    red: function() {
        this.element.css( "color", "red" );
    }
});

// Create a new <div>, convert it into a superDialog, and call the red() method.
$( "<div>I am red</div>" )
    .superDialog()
    .superDialog( "red" );
```

Now the superDialog has a `red()` method that will change the color of its text to red. Note how the widget factory automatically sets `this` to the widget's instance object. For a full list of the methods and properties available on the instance, see [the widget factory's API documentation](#).

Extending Existing Methods

Sometimes you need to tweak or add to the behavior of existing widget methods. To do this, specify a method with the same name as the method you want to override on the prototype object. The following example overrides dialog's [open\(\) method](#). Since dialogs automatically open by default, "open" will be logged when this code runs.

```
$.widget( "custom.superDialog", $.ui.dialog, {
    open: function() {
        console.log( "open" );
    }
});

// Create a new <div>, and convert it into a superDialog.
$( "<div>" ).superDialog();
```

While this runs, there's a problem. Since we overrode the default behavior of `open()`, the dialog no longer displays on the screen.

When we place methods on the prototype object, we are not actually overriding the original method - rather, we are placing a new method at a higher level in the prototype chain.

To make the parent's methods available, the widget factory provides two methods - `_super()` and `_superApply()`.

Using `_super()` and `_superApply()` to Access Parents

`_super()` and `_superApply()` invoke methods of the same name in the parent widget. Refer to the following example. Like the previous one, this example also overrides the `open()` method to log "open". However, this time `_super()` is run to invoke dialog's `open()` and open the dialog.

```
$.widget( "custom.superDialog", $.ui.dialog, {
    open: function() {
        console.log( "open" );

        // Invoke the parent widget's open().
        return this._super();
    }
});

$( "<div>" ).superDialog();
```

`_super()` and `_superApply()` were designed to behave like the native `Function.prototype.call()` and `Function.prototype.apply()` methods. Therefore, `_super()` accepts an argument list, and `_superApply()` accepts a single array of arguments. This difference is shown in the example below.

```
$.widget( "custom.superDialog", $.ui.dialog, {
    _setOption: function( key, value ) {

        // Both invoke dialog's setOption() method. _super() requires the
        // arguments
        // be passed as an argument list, _superApply() as a single array.
        this._super( key, value );
        this._superApply( arguments );
    }
});
```

Redefining Widgets

jQuery UI 1.9 added the ability for widgets to redefine themselves. Therefore, instead of creating a new widget, we can pass `$.widget()` an existing widget's name and constructor. The following example adds the same logging in `open()`, but doesn't create a new widget to do so.

```
$.widget( "ui.dialog", $.ui.dialog, {
    open: function() {
        console.log( "open" );
        return this._super();
    }
});

$( "<div>" ).dialog();
```

With this approach you can extend an existing widget's method and still have access to the original methods using `_super()` - all without creating a new widget.

Widgets and Polymorphism

One word of warning when interacting with widget extensions and their plugins. The parent widget's plugin cannot be used to invoke methods on elements that are child widgets. This is shown in the example below.

```
$.widget( "custom.superDialog", $.ui.dialog, {} );

var dialog = $( "<div>" ).superDialog();

// This works.
dialog.superDialog( "close" );

// This doesn't.
dialog.dialog( "close" );
```

Above, the parent widget's plugin, `dialog()`, cannot invoke the `close()` method on an element that is a `superDialog`. For more on the invoking widget methods see [Widget Method Invocation](#).

Customizing Individual Instances

All the examples we have looked at so far have extended methods on the widget's prototype. Methods overridden on the prototype affect all instances of the widget.

To show this, refer to the example below; both instances of the dialog use the same `open()` method.

```
$.widget( "ui.dialog", $.ui.dialog, {
    open: function() {
        console.log( "open" );
        return this._super();
    }
});

// Create two dialogs, both use the same open(), therefore "open" is logged twice.
$( "<div>" ).dialog();
$( "<div>" ).dialog();
```

While this is powerful, sometimes you only need to change the behavior for a single instance of the widget. To do this, obtain a reference to the instance and override the method using normal JavaScript property assignment. The example below shows this.

```
var dialogInstance = $( "<div>" )
    .dialog()

    // Retrieve the dialog's instance and store it.
    .data( "ui-dialog" );

// Override the close() method for this dialog
dialogInstance.close = function() {
    console.log( "close" );
};

// Create a second dialog
$( "<div>" ).dialog();

// Select both dialogs and call close() on each of them.
// "close" will only be logged once.
$( ":data(ui-dialog)" ).dialog( "close" );
```

This technique of overriding methods for individual instances is perfect for one-off customizations.

jQuery UI - Widget Factory

Using the classes Option

As of the 1.12 release, the jQuery UI widget factory includes a means of managing CSS class names through the [classes option](#). This article will give you an overview of how the `classes` option works, and discuss what you can do with it.

Syntax overview

The `classes` option is used to map structural class names to theme-related class names that you define. To see what this means let's look at an example. The code below uses the `classes` option to create a red dialog:

```
<style>
    .custom-red { background: red; }
</style>

    var dialog = $( "<div>Red</div>" ).dialog({
        classes: {
            "ui-dialog": "custom-red"
        }
    });
</script>
```

Here, the presentational `custom-red` class name is associated with the structural `ui-dialog` class name. Now, whenever the dialog applies the `ui-dialog` class name, it will also add a `custom-red` class name. However, something other than adding the `custom-red` class has happened here, which isn't immediately obvious. This code also *removes* the existing default value which was `"ui-corner-all"`. You can associate multiple class names by including multiple space-delimited class names in the object's value. For instance the following code creates a dialog that is red and still has rounded corners:

```
<style>
    .custom-red { background: red; }
</style>

    var dialog = $( "<div>Big and red</div>" ).dialog({
        classes: {
            "ui-dialog": "ui-corner-all custom-red"
        }
    });
</script>
```

Note: To get a full list of the class names you can use with the `classes` option, check the API documentation for the jQuery UI widget you're interested in. For example, here's the list of classes for the dialog widget: <http://api.jqueryui.com/dialog/#theming>.

The `classes` option works like any other widget factory option, which means all the widget factory option mechanisms still apply. For instance, the following code uses the [option\(\) method](#) to remove all class names currently associated with the `ui-dialog` class name:

```
dialog.dialog( "option", "classes.ui-dialog", null );
```

And the following creates a [widget extension](#) that automatically associates the `custom-red` class with the `ui-dialog` class:

```
<style>
    .custom-red { background: red; }
</style>

    $.widget( "custom.dialog", $.ui.dialog, {
        options: {
            classes: {
                "ui-dialog": "ui-corner-all custom-red custom-big"
            }
        }
    });
    $( "<div>Big and red</div>" ).dialog();
</script>
```

As an added benefit, the widget factory also removes any class names specified in the `classes` option when the widget is destroyed.

Theming

As the previous examples show, the `classes` option provides a quick way to associate theme-related class names with the structural class names used within a widget. This approach works for simple cases, but it can also be used to adapt third-party themes to work with widget-factory-built widgets. For example, if you're using [Bootstrap](#) and jQuery UI together, you can use the following code to create a jQuery UI dialog that uses Bootstrap's theming:

```
$.extend( $.ui.dialog.prototype.options.classes, {  
    "ui-dialog": "modal-content",  
    "ui-dialog-titlebar": "modal-header",  
    "ui-dialog-title": "modal-title",  
    "ui-dialog-titlebar-close": "close",  
    "ui-dialog-content": "modal-body",  
    "ui-dialog-buttonpane": "modal-footer"  
});
```

For more examples of this approach, check out [Alexander Schmitz's repo](#) that adapts jQuery UI to work with Bootstrap using the `classes` option.

Conclusion

The introduction of the `classes` option takes us one step further in the split between structural and theme-related classes, making it easier than ever to make jQuery UI widgets match the look and feel of your existing site. At the same time, this allows jQuery UI to be used alongside other CSS frameworks, just like jQuery can be used alongside other JavaScript frameworks.

jQuery UI - Using jQuery UI

Using jQuery UI

In addition to being available on [CDNs](#) and [Download Builder](#), jQuery UI also integrates into a number of development environments.

jQuery UI - Using jQuery UI

Using jQuery UI with AMD

****Note:**** This documentation refers to functionality made available in jQuery UI 1.11.

As of jQuery UI 1.11, all of the library's source files support using AMD. This means that you can manage your jQuery UI dependencies without using [Download Builder](#), and load jQuery UI's source files asynchronously using an AMD loader such as [RequireJS](#).

In this article we'll walk through the process of using AMD with jQuery UI. Let's start by discussing the files we'll need.

Requirements

We'll need to download three things to get up and running: jQuery core, jQuery UI, and an AMD loader.

While any AMD loader will work, we'll use RequireJS in this article, which you can download from <http://requirejs.org/docs/download.html>. If you don't have a version of jQuery core handy, you can get it from <http://jquery.com/download/>, and you can download the jQuery UI source files from <https://github.com/jquery/jquery-ui/releases> (use the files in the ui directory). You can alternatively [download these libraries using a package manager such as Bower](#).

Directory Structure

Now that we have the files we need, we have to discuss where to place them. For this tutorial, we'll build a small application that uses the following directory structure.

```
├── index.html
├── js
│   ├── app.js
│   ├── jquery-ui
│   │   ├── accordion.js
│   │   ├── autocomplete.js
│   │   ├── button.js
│   │   ├── core.js
│   │   ├── datepicker.js
│   │   ├── dialog.js
│   │   └── ...
│   ├── jquery.js
│   └── require.js
```

As you can see, we're placing all JavaScript files in a js directory. jquery.js and require.js are direct children of js, and all of jQuery UI's files are within a jquery-ui directory. app.js will contain our application code.

With RequireJS you're free to use any directory structure you'd like, but with alternative structures you'll have to [change some configuration](#) so RequireJS knows how to find your dependencies.

Loading the Application

Now that we have the files in place, let's use them. Here are the contents of our app's index.html file.

```
<!doctype html>
<html lang="en">
<head>
  ...
</head>
<body>

<script src="js/require.js" data-main="js/app"></script>

</body>
</html>
```

require.js is loaded in a `<script>` tag, which [by convention] (<http://requirejs.org/docs/start.html>) asynchronously loads and executes the file

specified in the `data-main` attribute – in this case `js/app.js`. If you put a `console.log()` statement in `app.js`, you can verify that it loads appropriately.

```
/* app.js */
console.log( "loaded" );
```

Our boilerplate is now in place. Next, we have to load jQuery and jQuery UI.

Requiring jQuery and jQuery UI

The `require()` function is AMD's mechanism for specifying and loading dependencies; therefore, we can add one to our `app.js` file to load the necessary files. The following loads jQuery UI's autocomplete widget.

```
require([ "jquery-ui/autocomplete" ], function( autocomplete ) {
    ...
});
```

When this code executes, RequireJS asynchronously loads `jquery-ui/autocomplete.js` as well as its dependencies: jQuery core (`jquery.js`), jQuery UI core (`jquery-ui/core.js`), the widget factory (`jquery-ui/widget.js`), the position utility (`jquery-ui/position.js`), and the menu widget (`jquery-ui/menu.js`).

When all dependencies are resolved and loaded, RequireJS invokes the callback function.

Using jQuery UI's Files

All widgets built with the widget factory expose their constructor function when required with AMD; therefore we can use them to instantiate widgets on elements. The following creates a new `<input>`, initializes an autocomplete widget on it, then appends it to the `<body>`.

```
require([ "jquery-ui/autocomplete" ], function( autocomplete ) {
    autocomplete({ source: [ "One", "Two", "Three" ] }, "<input>" )
        .element
        .appendTo( "body" );
});
```

Each widget's constructor function takes two arguments: the widget's options, and the element to initialize the widget on. Each widget has a default element that is used if no element is provided, which is stored at `$.namespace.widgetName.prototype.defaultElement`. Because `$.ui.autocomplete.prototype.defaultElement` is `<input>`, we can omit the second argument in our autocomplete example.

```
require([ "jquery-ui/autocomplete" ], function( autocomplete ) {
    autocomplete({ source: [ "One", "Two", "Three" ] })
        .element
        .appendTo( "body" );
});
```

Even though we're loading jQuery UI's files with AMD, the files' plugins are still added to the global `jquery` and `$` objects; therefore you can alternatively use the plugins to instantiate widgets. The following also creates the same autocomplete.

```
require([ "jquery", "jquery-ui/autocomplete" ], function( $ ) {
    $( "<input>" )
        .autocomplete({ source: [ "One", "Two", "Three" ] })
        .appendTo( "body" );
});
```

Datepicker

Since jQuery UI's datepicker widget is the only jQuery UI widget not built with the widget factory, it does not return a constructor function when required with AMD. Because of this, it's best to stick with datepicker's plugin to instantiate datepicker instances. The following requires datepicker, then uses its plugin to instantiate a datepicker instance on a newly created `<input>`.

```
require([ "jquery", "jquery-ui/datepicker" ], function( $ ) {
    $( "<input>" )
        .appendTo( "body" )
        .datepicker();
});
```

jQuery UI - Using jQuery UI

Using jQuery UI with Bower

****Note:**** This documentation refers to functionality made available in jQuery UI 1.11.

[Bower](#) is a package manager for the Web. You can use Bower to download libraries like jQuery UI from the command line, without having to manually download each project from their respective sites.

As an example, suppose we're starting a new project and we need to use [jQuery UI's accordion widget](#). We'll create a new directory for our project, and add the boilerplate index.html shown below.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>jQuery Projects</title>
</head>
<body>

<div id="projects">
  <h3>jQuery Core</h3>
  <p>jQuery is a fast, small, and feature-rich JavaScript library...</p>
  <h3>jQuery UI</h3>
  <p>jQuery UI is a curated set of user interface interactions...</p>
  <h3>jQuery Mobile</h3>
  <p>jQuery Mobile is a HTML5-based user interface system...</p>
</div>

    $( "#projects" ).accordion();
</script>

</body>
</html>
```

This example fails with a JavaScript error because neither jQuery core nor jQuery UI are loaded. Let's load them with Bower.

Downloading jQuery UI With Bower

Libraries are downloaded with Bower using the `bower install` command. To install jQuery UI, run `bower install jquery-ui`. Doing so creates the following (simplified) directory structure.

Note: If you get an error that the `bower` command is not found, check out [Bower's installation instructions](#).

```

├── bower_components
│   ├── jquery
│   │   ├── dist
│   │   │   ├── jquery.js
│   │   │   └── jquery.min.js
│   │   └── src
│   ├── jquery-ui
│   │   ├── themes
│   │   │   ├── smoothness
│   │   │   │   ├── jquery-ui.css
│   │   │   │   └── jquery-ui.min.css
│   │   │   └── [The rest of jQuery UI's themes]
│   │   ├── ui
│   │   │   ├── accordion.js
│   │   │   ├── autocomplete.js
│   │   │   └── ...
│   │   ├── jquery-ui.js
│   │   └── jquery-ui.min.js
└── index.html
```

A couple of things happened here. First, Bower knew that jQuery UI depends on jQuery core, so it downloaded both libraries automatically. Second, all of jQuery UI's files for the latest release were conveniently placed in a `jquery-ui` directory within a newly created `bower_components` directory.

Note: If you don't want the latest version, you can optionally provide a version number to `bower install`. For instance `bower install jquery-ui#1.10.4` installs version 1.10.4 of

jQuery UI.

Now that we have the files available, we have to use them.

Using Bower Downloaded Files

We have a few different options for using the files downloaded with Bower. The easiest is to use the minified and concatenated files in our `bower_components/jquery` and `bower_components/jquery-ui` directories. This approach is shown below.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>jQuery Projects</title>
  <link rel="stylesheet" href="bower_components/jquery-
ui/themes/smoothness/jquery-ui.min.css">
</head>
<body>

<div id="projects">
  <h3>jQuery Core</h3>
  <p>jQuery is a fast, small, and feature-rich JavaScript library...</p>
  <h3>jQuery UI</h3>
  <p>jQuery UI is a curated set of user interface interactions...</p>
  <h3>jQuery Mobile</h3>
  <p>jQuery Mobile is a HTML5-based user interface system...</p>
</div>

<script src="bower_components/jquery/dist/jquery.min.js"></script>
<script src="bower_components/jquery-ui/jquery-ui.min.js"></script>

  $( "#projects" ).accordion();
</script>

</body>
</html>
```

This code successfully builds our accordion widget, but it also includes the entirety of jQuery UI when we only need the accordion widget. Since there's a lot more than an accordion widget in jQuery UI, this forces the user to download far more than they need.

Because Bower also downloaded jQuery UI's individual source files, we can alternatively use them to send the user just the accordion widget and its dependencies. The following example builds the same accordion widget taking this approach.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>jQuery Projects</title>
  <link rel="stylesheet" href="bower_components/jquery-
ui/themes/smoothness/jquery-ui.min.css">
</head>
<body>

<div id="projects">
  <h3>jQuery Core</h3>
  <p>jQuery is a fast, small, and feature-rich JavaScript library...</p>
  <h3>jQuery UI</h3>
  <p>jQuery UI is a curated set of user interface interactions...</p>
  <h3>jQuery Mobile</h3>
  <p>jQuery Mobile is a HTML5-based user interface system...</p>
</div>

<script src="bower_components/jquery/dist/jquery.js"></script>
<script src="bower_components/jquery-ui/ui/core.js"></script>
<script src="bower_components/jquery-ui/ui/widget.js"></script>
<script src="bower_components/jquery-ui/ui/accordion.js"></script>

  $( "#projects" ).accordion();
</script>

</body>
</html>
```

From here, you can hook jQuery UI's files into your own custom build system to concatenate and minify your resources for production. If you're a RequireJS user, check out our [guide on how to use jQuery UI with AMD](#).

jQuery Mobile

jQuery Mobile is the easiest way to build sites and apps that are accessible on all popular smartphone, tablet, and desktop devices. This framework provides a set of touch-friendly UI widgets and an AJAX-powered navigation system to support animated page transitions.

jQuery Mobile

Getting Started with jQuery Mobile

jQuery Mobile provides a set of touch-friendly UI widgets and an Ajax-powered navigation system to support animated page transitions. This guide will show you how you can build your first jQuery Mobile application.

Create a Basic Page Template

To get started, you can simply paste the template below in your favorite text editor, save, and open the document in a browser.

In the `<head>` of this template, a meta `viewport` tag sets the screen width to the pixel width of the device. References to jQuery, jQuery Mobile, and the mobile theme stylesheet from the CDN add all the styles and scripts. jQuery Mobile 1.4 works with versions of jQuery core 1.8 and newer.

In the `<body>`, a div with a `data-role` of `page` is the wrapper used to delineate a page. A header bar (`data-role="header"`), a content region (`role="main" class="ui-content"`) and a footer bar (`data-role="footer"`) are added inside to create a basic page (all three are optional). These `data-` attributes are HTML5 attributes used throughout jQuery Mobile to transform basic markup into an enhanced and styled widget.

```
<!doctype html>
<html>
<head>
  <title>My Page</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
href="https://code.jquery.com/mobile/[version]/jquery.mobile-[version].min.css">
  <script src="https://code.jquery.com/jquery-[version].min.js"></script>
  <script src="https://code.jquery.com/mobile/[version]/jquery.mobile-
[version].min.js"></script>
</head>
<body>
  <div data-role="page">

    <div data-role="header">
      <h1>My Title</h1>
    </div><!-- /header -->

    <div role="main" class="ui-content">
      <p>Hello world</p>
    </div><!-- /content -->

    <div data-role="footer">
      <h4>My Footer</h4>
    </div><!-- /footer -->

  </div><!-- /page -->
</body>
</html>
```

Add Content

The next step is to add content inside the content container. Any standard HTML elements – headings, lists, paragraphs, etc. can be added. You can write your own custom styles to create custom layouts by adding an additional stylesheet to the `<head>` after the jQuery Mobile stylesheet.

Make a Listview

jQuery Mobile includes a diverse set of common listviews that are coded as lists with a `data-role="listview"` added. Here is a simple linked list that has a role of `listview`. The `data-inset="true"` attribute makes the listview look like an inset module, while `data-filter="true"` adds a dynamic search filter.

```
<ul data-role="listview" data-inset="true" data-filter="true">
  <li><a href="#">Acura</a></li>
  <li><a href="#">Audi</a></li>
  <li><a href="#">BMW</a></li>
  <li><a href="#">Cadillac</a></li>
  <li><a href="#">Ferrari</a></li>
</ul>
```


Add a Slider

The framework contains a full set of form elements that are automatically enhanced into touch-friendly styled widgets. Here's a slider made with the new HTML5 input type of range, no `data-role` needed. All form elements must always be properly associated with a `<label>` and the group of form elements be wrapped in a `<form>` tag.

```
<form>
  <label for="slider-0">Input slider:</label>
  <input type="range" name="slider" id="slider-0" value="25" min="0" max="100"
/>
</form>
```

Make a Button

There are a few ways to make buttons. A common one is to turn a link into a button so it's easy to click. Just start with a link and add a `data-role="button"` attribute to it. You can add an icon with the `data-icon` attribute and optionally set its position with the `data-iconpos` attribute.

```
<a href="#" data-role="button" data-icon="star">Star button</a>
```

Choose a Theme Swatch

jQuery Mobile has a robust theme framework that supports up to 26 sets of toolbar, content, and button colors, called a "swatch". You can add a `data-theme="b"` attribute to any of the widgets on this page: page, header, list, input for the slider, or button to turn it a dark shade of grey. Different swatch letters from a-b in the default theme can be used to mix and match swatches.

If you add the theme swatch to the page, all the widgets inside the content will automatically inherit the theme.

```
<a href="#" data-role="button" data-icon="star" data-theme="a">Button</a>
```

If you would like to create a custom theme, you can use [ThemeRoller](#) that allows users to create their own theme through an easy to use drag and drop interface. You will then be able to download and use your newly created theme.

Go Forth and Build Something

This guide has provided you with a basic structure for a jQuery Mobile page and a few enhanced elements. You can explore the full [jQuery Mobile API Documentation](#) and [jQuery Mobile Demo Center](#) to learn about linking pages, adding animated page transitions, and creating dialogs and popups.

If you're more of the type who prefers actually writing JavaScript to build your apps, and you don't want to use the `data-` attribute configuration system, you can take full control of everything and call plugins directly as these are all standard jQuery plugins built with the UI widget factory. Particularly useful information for such cases can be found in the global configuration, events, and methods sections.

Finally, you can read up on scripting pages, generating dynamic pages, and building PhoneGap apps.

jQuery Mobile

Creating a Custom Theme with ThemeRoller

Theming Overview

jQuery Mobile has a robust theme framework that supports up to 26 sets of toolbar, content, and button colors, called a "swatch". The framework comes with five defined themes (swatches "a" to "e") which can be used readily, removed, or overwritten.

Default Theme Swatch Mapping for Components

If no theme swatch letter is set at all, the framework uses the "a" swatch (black in the default theme) for headers and footers and the "c" swatch (light gray in the default theme) for the page content to maximize contrast between the both.

All items in containers inherit the swatch from their parent. Exceptions to this rule are the listdivider in listviews, the header of nested list pages, and the button of split button lists, which all default to "b" (blue in the default theme). Count bubbles default to "c" (silver in the default theme).

Note that there is also a swatch named "active" (bright blue in the default theme) which is used to indicate an active selected item. See the global "Active" state further down this page for more information on the active swatch.

The page loading dialog and error message don't inherit a swatch theme. The loading dialog defaults to swatch "a" (black in the default theme) and the error message to swatch "e" (yellow in the default theme). You can configure those defaults globally.

Themes and Swatches

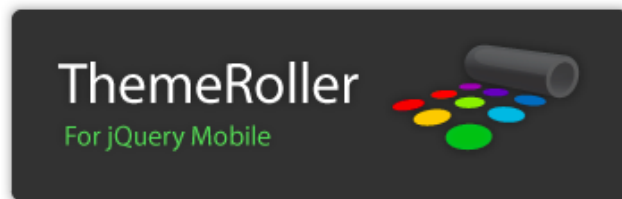
The theme system separates color and texture from structural styles that define things like padding and dimensions. This allows theme colors and textures to be defined once in the stylesheet and to be mixed, matched, and combined to achieve a wide range of visual effects.

Each theme includes several global settings, including font family, drop shadows for overlays, and corner radius values for buttons and boxes. In addition, the theme can include multiple color swatches, each with color values for bars, content blocks, buttons and list items, and font text-shadow.

The default theme includes five swatches that are given letters (a, b, c, d, e) for quick reference. To make mapping of color swatches consistent across our widgets, we have followed the convention that swatch "a" is the highest level of visual priority (black in our default theme), "b" is secondary level (blue), "c" is the baseline level (gray) that we use by default in many situations, "d" for an alternate secondary level, and "e" as an accent swatch. Themes may have additional swatches for accent colors or specific situations. For example, you could add a new theme swatch "f" that has a red bar and button for use in error situations.

Most theme changes can be done using ThemeRoller, but it is also fairly simple to manually edit the base swatches in the default theme and/or add additional swatches by editing the theme CSS file. Just copy a block of swatch styles, rename the classes with the new swatch letter name, and tweak colors as you see fit.

Creating a Custom Theme with ThemeRoller



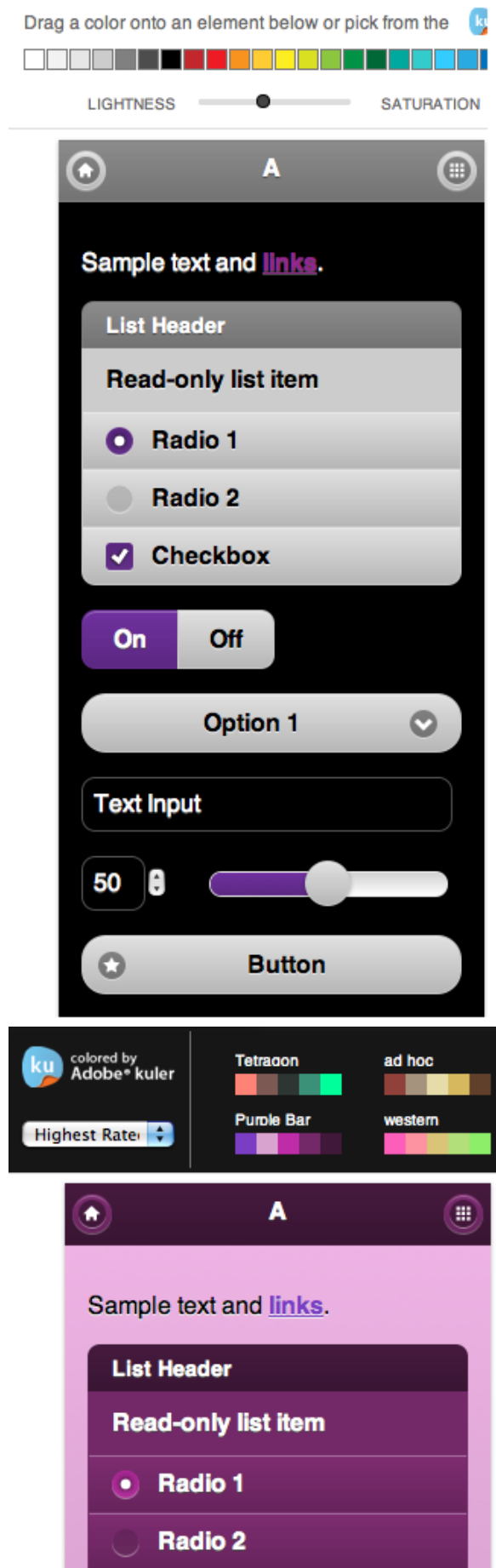
The easiest way to create custom themes is with the ThemeRoller tool. It allows you to build a theme composed of up to 26 swatches, download the newly created CSS file, and use it in your project.

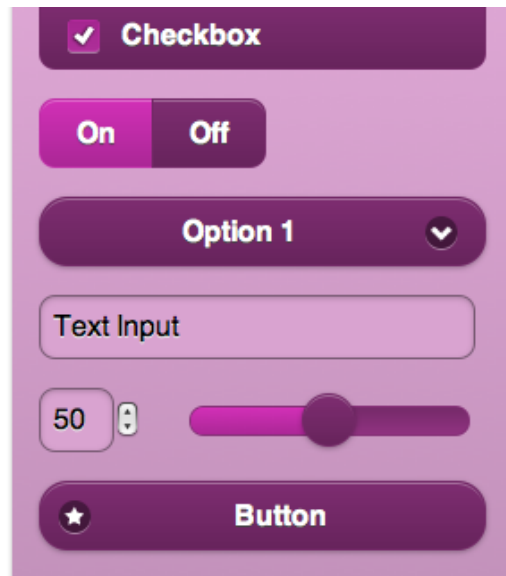
Creating the Theme Swatches

The [ThemeRoller](#) allows users to create their own theme through an easy to use drag and drop interface. By default, ThemeRoller offers three swatches (a, b and c). You can use the offered default colors, the Adobe Kuler colors, or create your own. You will create your theme by dragging the chosen color onto the chosen element in the swatch of your choice. You can add more swatches by pressing the "+" sign near the "A", "B", and "C" tabs, in the left-hand side menu.

You can further edit your swatch from the menu. For example, you can expand the various element parts and carry out detailed editing. This will allow you to change text color, text shadow size, position and color, etc. You can also edit the gradient used on each element.

Here are two examples of theme swatches created, one with the default colors, and one with the Kuler colors:





Downloading the Created Theme

Once you are satisfied with the various swatches that you have created in your theme, you can download this theme to be able to start using it in your project. You will simply need to press the "Download theme zip file button", and enter the name of your theme in the popup window. Then, press the "Download Zip" button on the download popup window, see below:

Download Theme

Theme Name

This will generate a Zip file that contains both a compressed (for production) and uncompressed (for editing) version of the theme.

To use your theme, add it to the head of your page before the jquery.mobile.structure file, like this:

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

  <title>jQuery Mobile page</title>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="css/themes/my-custom-theme.css" />
  <link rel="stylesheet" href="http://code.jquery.com/mobile/1.2.0/jquery.mobile.structure-1.2.0.min.css" />
  <script src="http://code.jquery.com/jquery-1.8.2.min.js"></script>
  <script src="http://code.jquery.com/mobile/1.2.0/jquery.mobile-1.2.0.min.js"></script>

</head>
```

Tip: To edit your theme later, use the import feature to paste in the uncompressed theme file

Close

Download Zip

Using the Downloaded Theme

The theme gets downloaded on your local machine as a zip file. This contains an `index.html` file, and a `themes` folder. The `index.html` file is an example of how you can now use your theme. The `themes` folder contains your theme CSS files, and the icons that are used by jQuery Mobile.

To start using your theme, you can either start from the provided `index.html`, or start from scratch. As explained in the theme download popup window, all you need is to add your theme to the head of your page before the `jquery.mobile.structure` file, like this:

```
<!doctype html>
<html>
<head>

  <title>jQuery Mobile page</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="css/themes/my-custom-theme.css">
  <link rel="stylesheet"
href="https://code.jquery.com/mobile/1.2.0/jquery.mobile.structure-1.2.0.min.css">
  <script src="https://code.jquery.com/jquery-1.8.2.min.js"></script>
  <script src="https://code.jquery.com/mobile/1.2.0/jquery.mobile-1.2.0.min.js"></script>

</head>
```

Final Note

You need to be aware that jQuery Mobile will default to certain swatches when none are specified. For example, page content will default to swatch "c", list dividers to swatch "b", etc. As the full jQuery Mobile CSS is replaced by your custom theme CSS and the jQuery Mobile structure CSS, the only swatches available are the ones that you have provided as part of your custom theme. Therefore, you need to either always specify a swatch letter for all your elements or their parent using for example the `data-theme` attribute, or you will need to provide a swatch in your custom theme for the possible defaults. Additionally, the error messages use the swatch "e", so this should also be specified in your theme.